CHAPTER 8

Experimentation with an Agent-Oriented Platform in JAVA

The flapping of a single butterfly's wings today produces a tiny change in the state of the atmosphere. Over a period of time, what the atmosphere actually does diverges from what it would have done. So, in a month's time, a tornado that would have devastated the Indonesian coast doesn't happen. Or maybe one that wasn't going to happen, does. —I. Stewart [Stewart 1989]

8.1 Modeling Complex Systems for Simulation Purposes

This chapter begins by showing that simulation plays an important role for understanding complex systems and framework specifications.

8.1.1 Simulation as a Solution to Understanding Complex Systems

Complex systems arise in the social, biological, physical, and behavioral sciences, in phenomena from urban traffic networks to price movements on financial markets. The frightening complexity of natural systems emphasizes simulation applications as the best actual solution to understanding and predicting their underlying phenomena. Simulation is an appropriate methodology whenever a phenomenon is not directly accessible, either because it no longer exists, or because its structure or behavior is so complex that the user cannot know what is going on [Conte-Gilbert 1995]. It may help to reformulate the system behavior model in more specific and empirically relevant terms. The boom of computer science in systems modeling during the past 20 years has greatly increased the understanding of complex systems through the use of virtual simulation. This technology is proposed by both researchers and industrial engineers, constituting a real research–industry transfer. From accumulated experience, simula-
tion is now considered to be ahead of theoretical approaches, and is becoming a promising tool in this framework.

Our framework relates to the SWARM system [Minar 1996], one of the best-known simulation platforms with agents. In SWARM the agent's behavior is defined by the emergent phenomena of the agents inside its swarm. However, a swarm does not explicitly dispose of any organizational structure, no further information being needed when the phenomenon appears; this causes the difference with our approach.

To make the best use of simulation, the first step is to aggregate data, knowledge, and hypotheses to build a model of the physical world. A model is defined as an abstraction of the real system, governed by laws and structured concepts. This model is generally expressed by mathematical relationships between variables, such as differential equations. Simulation consists of making a system abstraction (a model) evolve over time in order to try to understand the system's working and behavior and to analyze the properties of the theoretical models. Simulations run on computers provide tests to validate the theoretical model, or they may simply allow the experimenter to observe and record the target system's behavior. Results can then be processed and exploited with the help of statistical techniques to verify the given hypotheses.

8.1.2 Framework Specifications and Challenges

Our "customers" are industrial engineers and researchers who want to explore and understand the field of potential actions in a specific domain by simulation. In the domain to be investigated, the question is not what has happened, or even what might have happened, but rather what the sufficient conditions are for a given result to be obtained.

This chapter discusses the design and implementation of a development environment (sometimes called a tool or platform) that is intended to model and simulate complex systems. For instance, this kind of tool is meant for researchers who need to simulate complex physical models, without having to implement the whole application on their own. In terms of software engineering issues, this objective requires generic applicability and reusability as first objectives of the platform design, which justifies the use of object-oriented design methodology such as object modeling technique (OMT) [Rumbaugh 1991]. Indeed, our purpose is to produce a complete toolkit as a virtual laboratory for designing a large scope of dynamic systems and studying the informational structure of complex systems, and to provide generic interfaces to set and control the simulation.

The project is then dictated by the facts that (1) there are no appropriate tools (especially in simulation), (2) reducing complexity to model complex systems requires appropriate structures, and (3) intrinsic mechanisms of global behavior emergence are complex and not clearly identified. As the system complexity cannot be globally expressed, the challenge is to find a computational model able to represent and distribute complexity in individual elements, represent the system's dynamics as local interactions between agents, and provide mechanisms so that simulation results emerge by interpreting such local interactions.

The following section presents the design of a platform conforming to such specifications.
8.2 Matching the Agent Paradigm with Framework Challenges

This section gives an overview of different views, and levels of abstraction and matches the agent paradigm with framework challenges.

8.2.1 A Global View of the Platform Analysis

Figure 8.1 gives a general view of the platform analysis, which is briefly exposed here before being detailed in the following sections.

Horizontally, the platform is described following three software engineering layers:

**Language layer.** Describes Java as an object-oriented language, which has been chosen to implement the platform. The reasons for this choice are twofold. The first concerns the clarity and clearness of the language, according to the object-oriented philosophy and mechanisms (all-is-object, static and dynamic inheritance, dynamic overloading, polymorphism, and so forth), as well as its utilities, such as threads, which are useful for agent implementation. The second is Java's safety catch when using applets—that is, small, subordinate or embeddable applications to be run and used within the context of a larger application, such as a web browser [Neimeyer-Peck 1996].

![Figure 8.1 A complete view of the platform.](image-url)
Agent layer. Adds the agents' capabilities to Java—that is, independence, autonomy, and evolution features.

GUI layer. Hides the complexity of languages and agents to the user by introducing interfaces needed to build applications and derive simulations with the platform. The GUI layer helps in integrating the platform as a toolkit and a virtual laboratory for complex systems simulation purposes. This chapter does not go further into this layer for reasons of space.

The platform is also vertically based on four knowledge abstraction layers:

Object layer. This is the language layer. It is composed of each Java class being used or derived to implement the subabstraction layers. Figure 8.1 illustrates some of these classes—for instance, Thread, to allow parallel tasking; Observable, to allow the use of dummy functions; or Vector, to dynamically manage a list of values.

Generic Architecture for Multiagent Simulation (GEAMAS) layer. This layer implements a computational model of agents to match the specifications required by complex systems simulation. The GEAMAS layer constitutes the heart of the system. It includes a specific architecture (composed of a macro-agent, medium agents, and microagents), the description of an agent and a society model, and self-organization mechanisms required by complex systems simulations. Figure 8.2 illustrates such a layer, which is described in the following sections. In Figure 8.2, a double arrow means that the two objects that participate in the relationship mutually need to be known to each other. A simple arrow from A to B means that A objects need to know about B objects to act, but that B objects do not require the opposite; therefore, the connection of the relationship from B to A is not to be implemented (it is represented between square brackets in Figure 8.2). Finally, if any object needs to be mutually known, the relationship is dropped (that is the case between the Structure and Agent classes, information being assumed to transit between Agent and Society).

Application layer. This layer can be seen as the result of the GEAMAS layer applied on a specific domain. An application is then understood as a real-world translation in terms of agents following the model implemented in the GEAMAS layer. Building an application then consists of describing the real world by deriving some or all classes of the previous layer by using GUIs. Examples of applications might be the description of a volcano in geophysics or of the evolution of a financial market in economics. The Application layer is a set of classes, whose definition is provided by the system editor.

Simulation layer. This is the instance level of the application. It defines the set of instances of the previous classes. The instantiation of such classes is graphically set with the help of the application editor, which proposes some fill-in-the-blanks windows to set each parameter value.

The next section details the design of the GEAMAS layer, which gives birth to agents as the basic units to model real-world components in our platform.
8.2.2 The GEAMAS Layer: Macro, Micro, and Medium

The architecture is based on three abstraction levels. Each successive level represents a higher level of abstraction. Each abstraction level describes a degree of knowledge complexity and applies a model of agents, through the expression of knowledge, behavior, and evolution capabilities. Each level is independent, executing processes asynchronously and representing a higher level of abstraction than the one below it. Furthermore, the architecture implements the recursion property to greatly reduce the design of the system, by applying the same agent-model to coarse-grain and fine-grain components. To move up and down levels, two fundamental mechanisms are introduced: Decomposition and Recomposition. Decomposition allows the transfer of information to lower levels, until Recomposition allows the transfer of information back to higher levels.

**Macrolevel**

The macrolevel describes a coarse-grain agent as a related element in which global solutions will be observed. It builds on the underlying agents’ organization in a society. In multiagent systems, such an organization has no reason to be if it is unable to represent interaction facilities between agents. The society is therefore organized as a network of acquaintances, managing agents in the system to match global specifications. The role of the whole system is expressed through an external behavior, and driven by an input/output interface with the external world (software or human).

**Microlevel**

The microlevel corresponds to fine-grain agents. It describes the microlevel as reactive agents. Each reactive agent is described by interactions, microbehaviors, and evolution capabilities. At runtime, agents interact in a concurrent way with their surroundings. The microlevel masks the complexity of the whole system, which is encapsulated in agents describing low-level and basic actions. Such reactive agents are often called cells in some related areas (biology or medicine, for instance), or microagents, in a more general way. A microagent is reactive, as it generates behavior without explicit representation of the domain or global task constraints, microagents just interact with their surroundings and evolve over time by updating their internal state. They get close to each other and form a compact group. They act in response to events that are too fine-grained to be understood (or explained) by a macroagent. Interactions between microagents are thus provided by signals which do not bear semantics, their meaning depending on the interpretative ability of the receiver.

**Medium Level**

However, as we have seen before, a complex system should integrate the nonpredictable dimension: The critical state of the system leads to catastrophes. Our assumptions are then to consider the critical state as assigned to particular states of microagents, and the
Experimentation with an Agent-Oriented Platform in JAVA

avalanche to a spatiotemporal aggregation, in some particular context and environment, and under specific conditions, of microagents taking part in the phenomenon. The result of this aggregation is then an emergent structure created by self-organization in the multiagent system.

Therefore, the main issue is to look for a computational model that can represent and characterize self-organization from local interactions between agents. Considering a two-level architecture in the system makes the catastrophe-modeling task difficult for two reasons. On one hand, as a microlevel is described by fine-grain agents, the organization of new structures cannot be processed at such a microscopic level; it does not have enough knowledge to do so. Beyond this microscopic dimension, emergent structures can arise only in one macroscopic level, disposing of missing knowledge. On the other hand, it is often impossible for a macrolevel to characterize and correctly identify the whole set of emergent structures. As the complex system is unpredictable and badly understood, trying to explain emergence mechanisms at the global level of the system fails. As a matter of fact, a significant number of microagents could intervene in the whole emergence process, making the system too complex to be analyzed at a macrolevel.

It is then elegant to introduce an intermediate-grain level between the macrolevel and the microlevel, embedded as suborganizations. Such an intermediate level is called the medium level, and the suborganizations, medium agents. The medium level is designed before the simulation begins by giving suborganizations structure and behavior. This point of view enforces the necessity of finding the most efficient abstraction when designing this level, which is totally dependent of the context tackled [Marcenac-Calderoni 1997]. However, medium agents are not dynamically predefined in the system, because they spontaneously appear during the simulation as the result of self-organization. When self-organization arises (meaning that the system’s critical state is reached), a new medium agent will be created within the system to model the catastrophe.

This approach seems handy, because it allows consideration of the catastrophe as an entire entity able to compute its own properties. Moreover, it is at the right place to distribute new computed values after the critical point is reached in order to constrain the affected part of the system. Medium agents’ properties are computed and set when the organization is built. Such properties are then reintroduced in underlying lower-level agents as constraints to apply in their own structures, a mechanism called back-propagation. In this way, a medium agent plays its causal role for the rest of the simulation, by constraining a part of the system.

8.2.3 A Detailed View of the Society Model

The macrolevel manages agents in the system in order to match global specifications of the application; it is inscribed in a model called the society model. The society model describes the role, the interface, and the whole organization of the system: The system’s role is expressed through behaviors (global events and global constraints); the interface with the external world (software or human) describes input and output parameters for simulation needs; and the organization is managed through a structure of interrelated agents.
Society Knowledge

Input parameters govern the whole system and describe all agents’ states. In a sense, input parameters restrict the system extent by constraining agents’ states. Output parameters express the role of the system. Such parameters are used to measure the simulation results of the system outputs. They express the external behavior of the system to be observed by the user that constitutes the system’s goal. They are managed by a graphical user interface (GUI) described by the GUI editors of the Application layer.

Society Structure

The structure of the society describes how agents are organized, and is then responsible for the collective of agents representing interagent connections (as interaction possibilities between agents). This organization is defined as the description of agents’ dynamic relationships used to structure interactions within the system. The structure is then organized as a network of acquaintances, forming a competence network, wherein agents correspond to nodes, and interaction possibilities (acquaintances relationships) correspond to edges. The choice of a network as a data structure to organize agents is justified by its flexibility and its general nature of adaptability to several situations [Marcenac 1996]. In addition, it allows the description of some real-world topologic representations, where the geometry can be implicitly represented. Indeed, the number of acquaintances of each agent determines the topology, and constitutes a very interesting feature, showing how a real world could be designed as a three-dimensional network. This parameter is called the agents’ connectivity. As connections determine the ability of an agent to communicate, a part of the communication protocol is then defined by setting this parameter. This number of connections describes the influence an agent can have on another, and defines a spatial and geometric disposition of the agents.

In addition, some more information has been added in the diagram in Figure 8.3.

First, edges can be strengthened to take the signal intensity between two agents into account. This feature allows a certain priority of messages among agents to be expressed by the application designer. Indeed, the intensity of a signal emitted by an agent decreases as a function of the distance between agents, and agents’ behavior is

![Figure 8.3 Model of agents’ organization in the society.](image)
strongly dictated by their relative position in the topological structure. In our model, agents evolve in a world whose structure is defined by the agents’ communication network, and an agent can interact solely with its immediate neighborhood. Agents’ behavior and state are therefore influenced by the actions of their neighbors. When an agent acts or changes state, some of its neighbors may react more or less strongly according to the signal intensity.

Second, some agents are shown as able to receive external solicitations. This kind of agent will be considered first when an external event has to be performed by the system. In descriptive real worlds, such as physics, for instance, such agents generally model the system’s side. Finally, along the same idea, some agents are shown as able to provide results, modeling, for instance, the opposite side of the system. This information allows the system to compute simulation results or adapt its behavior (see Figure 8.3).

**Society Behavior**

The behavioral part of the society describes what the system has been designed for. It includes the following:

**Global events description.** This describes events being performed by the system. Each external event is first interpreted by the society that controls the event. As the system’s behavior is nonlinear, this event cannot be performed at the global level. The external event is therefore decomposed on underlying agents able to receive external solicitations, according to the general decomposition mechanisms described following.

**Global constraints definition.** This defines parameters that could constrain the whole structure by applying specific behaviors. They are composed of general strategies and laws according to the domain tackled. For instance in a financial market, there can be a law prohibiting certain types of trades when the Dow Jones Industrial Average has declined more than 50 points from its previous close (this rule was instituted by the New York Stock Exchange shortly after Monday, October 19, 1987, to prevent the kind of panic seen on that Black Monday).

**Acquaintance establishment.** When simulation begins, each agent looks for acquaintances in the organization according to the network topology, so agents might communicate with neighbor ones.

**General Mechanisms**

What is not shown in Figure 8.2 is a set of global mechanisms that assure general tasks at the society level. Two kinds of mechanisms are identified: first, a set of decomposition mechanisms that allows a global input event to be distributed over the network in order to be performed by lower-level agents; second, the part of the self-organization mechanisms needed to create, manage, and preserve consistency of medium agents during phenomena emergence.

**Decomposition Mechanisms**

The Decomposition process is defined to propagate a message to underlying agents. Decomposition allows information to be transferred in lower levels of the architecture.
When receiving a Decomposition message, the agent gets information given by its society. Two kinds of information can be addressed by such a message. First, a Decomposition message could specify that an external event is to be performed by lower-level agents. For instance, the external perturbation has to be distributed between all concerned sublevel agents, such as between the society and the microagents. This type of message helps microagents to be ready to perform the event. The event is thus divided up, and subevents are transferred to the microagents that are best able to process them. Conversely, there may be laws or knowledge on a macrolevel that must be observed by microagents—for instance, knowledge that constrains microagents’ movements. A society is the right place to express such knowledge; it may constrain part of the state and behavior of its subordinate microagents, by acting on either the world structure or on individual microagents.

A process is also defined to find the handling underlying agents that can perform an event, according to agents defined as able to receive external solicitations in the organization. It defines a kind of process that is responsible for localizing the necessary agents to perform a kind of event and that remains coherent with the current goal of the society.

Finally, a filtering function receives an external event in input and provides discretized subevents in output. Such a function is application-relative, as the method of transforming an event in such tiny simulated events cannot be generic in any case. For instance, in a geophysical application, such as modeling volcanic eruptions, external perturbations are caused by specific behaviors; for instance, to simulate external fluid feeding, such as from the earth’s core, magma volumes are injected within the system. When drained, the society modeling the edifice has to distribute the external injection to microagents modeling magma lenses. The event is divided in order to model the fluid flow: If a volume of magma $V$ is transferred to a lens during the simulation, the input volume is discretized in multiples $n$ of $V$, and an amount of magma $nV$ is injected into each microagent. From a programming point of view, $nV$ values represent time units required to recompute the internal pressure between two messages. Indeed, microagents modify their state rapidly, in order to maintain the correct pressure value and be ready for the arrival of new disturbances. At the injection of the next $nV$, the microagent is then ready to process the new computation.

**Global Self-Organization Mechanisms**

When self-organization is required to represent emergent phenomena, a global mechanism is used to create a medium agent, aggregating some microagents or adding them to an existing suborganization. If a suborganization already exists, the underlying medium agent managing the suborganization is advised, and the new microagent is integrated within the existing structure. Note that if the two microagents are members of two different suborganizations, they can be integrated on both suborganizations to take different emergent phenomena into account. If no medium agent already exists, the global creation mechanism creates a new medium agent responsible for the emergent structure. This agent will then be populated step by step. Finally, note that such an encapsulated microagent can leave the structure at any time, if its state evolves again. If only one agent remains in the structure, the suborganization is deleted.

Beyond this mechanism, backpropagation involves a medium agent to act as a feedback on each member of its structure. At the design phase of the application, the
designer gives the void structure of a medium agent defining properties that are independent of the microagents' properties. During the simulation, when the medium agent is entirely populated, the global mechanism computes appropriate values for the new agent's state. Setting new properties in a medium agent becomes easy at this point, mainly because the global creation mechanism can pick up information from lower levels. This set of new properties defines a kind of constraint or restriction governing the new structure. This constraint acts as a global law for all underlying microagents belonging to the structure, and backpropagation is the way to inform such microagents. Therefore, when the emergent structure backpropagates a constraint, it forces the behavior of underlying agents by applying some of its properties to them. This view is very close to reality: For instance, during an eruption each magma lens releases energy and moves into the system. The remaining energy is then low, and the microagents should be properly reinitialized so that the whole system behavior might be modified accordingly.

8.2.4 A Detailed View of the Agent Model

The agent model is the most important part of the platform, as it defines the agent as the only unit to be handled when modeling. It describes the agent's knowledge, role, independence and autonomy capabilities, and local mechanisms for self-organization purposes. Knowledge is given by states; role is expressed though internal and external behaviors; asynchronous part is ensured by two mailboxes (In and Out); and autonomy is managed by a kind of metacontrol of the agent's behavior, named consciousness.

Agent's Knowledge

The agent's state is represented with a state vector, from which each coordinate describes an internal property. A subset of these internal properties, called state parameters, is limited by thresholds giving the agent's critical state. A specific behavior is then associated with each threshold. Thresholds and state parameters are set during the designing phase of the application. The agent's stability is defined as a satisfaction state with regard to its role. An agent is then stable when state parameter values do not reach the thresholds.

Agent's Behavior

The agent's behavior represents what the agent is able to do during its life. In our model, the agent's behavior can be internal or external. Internal behaviors allow agents to work without external solicitations. They describe a life cycle of the agent and perform actions continually by updating the state vector. Internal behaviors ensure the independent part of the agent. For this reason, the agent's behavior should be implemented as a Thread (in the Java sense, that is, as a parallel process). External behaviors are provided by the agent when receiving external solicitations (such as events). For instance, one of the scenarios for such an external behavior might be: (1) update the state vector, (2) compute the rest of the constraints to propagate in the neighborhood,
(3) choose agents to which to propagate the event, and (4) propagate the event in the chosen agents.

**Agent’s Asynchronous Part**

The asynchronous part of the agent is derived from the use of two mailboxes, In and Out. The In class stores input messages that should be performed by the agent, ordered by intensity. The Out class stores each message to be propagated in the agent’s neighborhood after being locally performed. The mechanism used to set the appropriate behavior in a neighbor agent is not given by a traditional message, but is based on the observer mechanism in Java. Each In mailbox observes the agent’s acquaintances’ Out mailboxes. When an object B wants to propagate an event to its neighborhood, it leaves the message in the Out box. The Out box is observed by the In boxes of all acquaintances (agent A for instance), which automatically catch the message. This mechanism is similar to those of demon functions used in knowledge representation systems in the 1980s. Figure 8.4 illustrates such a mechanism.

**Agent’s Autonomous Part**

The autonomous part of the agent is managed by a kind of metacontrol that controls the agent’s behavior and determines which action to invoke. This mechanism allows maintaining room for decision making, because the agent’s behavior is dynamically self-adapted during the simulation. This characteristic then authorizes an agent to reproduce adequate behaviors—that is, behaviors that are the best adapted to the environment and to the current situation. Such a mechanism is called Consciousness. As it constitutes the overcontrol of the agent, this process should itself be independent in order to be able to stop the agent’s current behavior if some conditions change. This is why Consciousness inherits from Thread.

![Diagram](image_url)

**Figure 8.4** Observers and asynchronous message passing.
Recomposition

Recomposition is the bottom-up mechanism that is at the root of emergent behaviors of the system. It transfers information on the agent's stability from microlevel to macrolevel. The values of state parameters evolve during the simulation, based on multiple local interactions between microagents. When one threshold is reached, the agent is assumed to be unstable, and information is then transferred to the agent's society with a Recomposition message. Therefore, a Recomposition message is provided by microagents to alert the society that something unusual is happening. Shifting from the microlevel up to the upper level, the society collects data on microbehaviors, combines them to determine macrobehavior, and then adapts itself to the situation. Recomposition then requires that the society have some abilities to interpret information at low levels. A higher-level behavior then emerges from such an ability. In such cases, the dynamic of low-level agents governs emerging behaviors.

Local Self-Organization Mechanisms

Agents interact according to the organizational structure described at the macrolevel. If some agents are looking for cooperation to act together in a common purpose, others, on the contrary, will get clash. In this way, some groups of agents will be self-organized through interactions, forming new organizational structures. Self-organization then requires the following mechanisms:

A global mechanism to create and manage the new structure within the system. This mechanism was previously presented in the society model.

A local detection mechanism, distributed over microagents, to detect and identify an effective cooperation between interacting agents, leading to an emergent phenomenon. At the time an interaction occurs, an agent A alerts an agent B to look for its intentions. Then, a local observer mechanism compares agent A's intentions with those of agent B. As the similarity is detected, cooperation is established, and the agents can be considered to have organized for a common purpose. From this self-organization, an organizational structure emerges locally that has never been described in the system before. This structure is semantically based on the similarity between agents.

A mechanism in charge of ending the self-organization process when the emergent phenomenon stops. This mechanism, usually called an observer, acts as an end-detection mechanism by examining the neighborhood to look for potential agents that are not stable enough to propagate the phenomenon again. The end-detection mechanism computes the number of microagents remaining unstable. Thus, when the counter registers nil, the end of propagation is detected. In traditional multiagent systems, the programmed observer is generally implemented as a global loop inserted over the multiagent system or as a part of the society, to inspect the stability of underlying agents forming the structure. In our architecture, the programmed observer is distributed and locally defined, to remain close to the agent paradigm.

The next section describes some elements of the implementation, as well as hints to better design applications with the platform.
8.3 Java Implementation and Application Generation

The following section outlines the Java classes that have been implemented according to the design steps presented in Section 8.2 and shows how to develop applications with the platform.

8.3.1 Java and Agents

To translate the design into implementation accurately, some simple rules have been followed by programmers. Each inheritance link has been implemented using the Java inheritance keyword extends, and multi-inheritance has been avoided. As a consequence, the concept of Interface in Java has been reserved for defining parsers between each layer in the platform rather than for solving multi-inheritance issues. An exception has been made for the observer mechanism, which requires the use of intrinsic interfaces. Each relationship size has been implemented by holding each size arrow and looking at its connections (see Figure 8.5):

Case 1,1. A parameter is added in the needed class.

Case 1,n. A container is created (Vector or List) and a parameter is added to refer to the container created.

Figure 8.6 gives a complete illustration of the Java class hierarchies defined to implement the whole platform according to the design.

![Figure 8.5 Implementation of 1,1 and 1,n.](image-url)
8.3.2 Developing Applications with the Platform

After collecting a number of observations of the real world, empirical pattern rules are constructed which constitute the starting point of the real-world theory to be modeled. This description of the field to be investigated is called the application. Simulations are then derived from the application, and can be considered as application instances. To describe an application, the user must adapt an abstracted model of the real world to the three-level architecture. This foundation is inherent to the problem to be solved, namely modeling and reducing complexity. Modeling then consists of describing all levels in the architecture by applying the agent model and the society model at each appropriate level. After modeling, simulations are instanced from the application so built, and the real-world theory is tested against observation. If results are not satisfying, the theory can then be easily refined by deriving a new simulation from the application and repeating the experiment.
This section offers some hints in order to give the user the means to design an application in the best way. When using object-oriented analysis methods such as Coad and Yourdon’s [Coad-Yourdon 1991] or OMT [Rumbaugh 1991], the first step is to understand the real world by looking at “business objects,” which is particularly difficult when evolving applications in an unfamiliar environment. Designing with agents does not defy the law: The first step is to look at the real world to find software components that have enough suitable characteristics to be agents in the system and determine the logical aspect of the system, according to external specifications.

Then, the designing task is quite well structured, because the designer drives through a three-level architecture, and a model for both the agent and the society is provided. The designer has to match the model for the three levels and define the right level of complexity at each level. To find the adequate separation, the methodology we propose is based on the distribution of roles in the system. Each agent plays a role for the whole application, and bears semantics. Each role played by the agent in the application is described by a set of internal and external characteristics. These characteristics define the agent model, expressed by knowledge, internal and external behaviors, and consciousness, which have to be designed for each agent type. In addition, the designer has to look at what level of activities and functions performed by an agent should be described. This determines the right level of complexity and granularity. Finding the limit of complexity to be introduced is the main issue when modeling complex systems.

To address such an issue, three steps should be taken. First of all, begin to build the microlevel. This level inscribes only the agent model (no lower level, so no society to describe). Behaviors are quite simple, and can easily be split up into tiny pieces. Because a microagent can not be decomposed again, it describes the most tiny grain. The right level of granularity is reached when the agent cannot be described in any further detail, following the triad knowledge-behavior-consciousness. At this step, the stability of a reactive agent also has to be determined. Because the stability is given by some parameter thresholds, such an agent is stable when the values of associated parameters do not reach the thresholds. These values will then be computed according to messages handled by the microagent during the simulation. The computation depends on the complexity of the network described in the organization, and the neighboring agents’ states.

The second step is to design the macrolevel. As the microlevel matches the agent model, a macroagent matches the society model. So the designer must define the input and output parameters of the system and the agents’ organization (connectivity and intensity), as well as global events and constraints to be considered. Finally, the last step is to design the medium level. Two remarks could help the designer at this step: On one hand, a medium agent should apply both the agent and society models to respect the recursion property; on the other hand, a medium agent will be dynamically created by self-organization mechanisms. The medium level is then designed before the simulation begins by giving suborganizations structure and behavior according to agent and society models. However, medium agents are not dynamically predefined in the system because they spontaneously appear during the simulation as the result of self-organization.
8.4 A Real Example: Agricultural Biomass Management

Agricultural biomass management is becoming an important stake today, driven by environmental and pollution considerations. The objective of this application is a better understanding and management of stockbreeding operation effluents on Reunion Island, in the Indian Ocean, by using and comparing several simulation approaches. The system to carry out must, a priori, be centered on the simulation of matter flow, information flow, and financial flow. This will be achieved on the basis of the system structure (physical, technical, and geographical), of actor behavior, and of decisional choice. The model must make analyses possible. These analyses should point out relevant issues to the question of what would occur if one gathered a certain subset of stockbreeders around a certain composting platform, and those stockbreeders adopted a certain rule of management.

8.4.1 Agents and Situated Objects

Figure 8.7 represents the model structure on the basis of interactions between two Exploitation agents. In terms of agent classes, we thus distinguish Conveying agents (carriers) and Exploitation agents (made up of breeding and culture agents). Exploitation agents are equipped with an internal function of coordination and external communication. The internal function is determined by the role, an important notion of the application. The role describes either an offer or a demand.

These agents operate in an environment described as a whole of located objects. The distinction takes place on the basis of capacity to act. Whereas the agents are dynamic entities and have the capacity to act on other agents and their environment, the objects located are static entities that impose constraints on agents' actions, and can be modi-
fied only by them. The main identified objects located describe parcels, livestock buildings, and transportation communication channels (the roadway system). The application is then implemented following two main modules, biomas.agent (regrouping agents) and biomas.environment (which describes situated objects).

The environment is implemented as simple Java classes, but each agent is implemented as a set of Java classes, following the generic structure defined in the GEAMAS level. Each agent is then faithful to the structure defined in Section 8.3, and is then a micro-, medium-, or macro-agent: biomas is the macroagent, which describes the agents' society; ExploitationGroup is the medium agent; and breeding, culture, and conveying are described as microagents. These agents are simply inherited from the generic classes of the GEAMAS kernel, thanks to the user interface.

After defining Exploitation agents' roles in terms of offers and demands, negotiation between Conveying and Exploitation agents is quite simple. This scenario is illustrated in Figure 8.8.

Simulations have actually been performed by our team since the beginning of 1998. For further analysis, a graph plotter has been developed in Java and interfaced with GEOMAS outputs. For each simulation, the plotter prints graphs, and this is intended to quickly compare the results between different simulation methodologies. This application can be downloaded from our web site, www.univ-reunion.fr/~biomas/demo.

### 8.5 Summary

This chapter presents a generic platform to develop simulation applications of complex systems. The complexity of the systems tackled needs to consider the result of the
program execution as provided by a set of interactions among agents. Knowledge di-
vision, which helps to reduce a design's intricacy when tackling complex systems, 
emphasizes the use of the agent paradigm. The twisting specifications of the platform 
involve representing complexity in individual elements, the system's dynamics as 
local interactions between agents, and the result of simulation as the emergence of phe-
nomena by interpreting such local interactions. These challenges justify the need for an 
OMT approach to conduct a strong analysis and design. This software engineering 
point of view has led the project team to develop a specific architecture that allows 
distributing the complexity in independent and autonomous agents, as well as the focusing 
of a model of both the agents and the society. 
This approach has proved to be very helpful in understanding and analyzing the 
behavior of geophysical or environmental simulations in life-sized contexts. Examples 
of such applications are presented in the chapter and are available from our Web pages. 
The global system behavior has been reproduced from a model of very simple micro-
agents' behavior. This global behavior has previously been unattainable with more 
classical approaches. The architecture therefore validates both the methodology and 
the approach for complex systems modeling and simulation. We are now convinced 
that the architecture is appropriate for simulating other natural systems that exhibit 
similar behavior, and we actually aim to experiment with the approach in the social 
behavior framework.

8.6 References

social theory. Artificial Societies—The Computer Simulation of Social Life, pp. 1–18. Lon-
based simulation. Proceedings of Modeling Autonomous Agents in a Multi-Agent World 
[Marcenac 1996] Marcenac, P. Emergence of behaviors in natural phenomena agent-
The ongoing development of communication technologies (such as the Internet and Intranets) provides users with powerful distributed and concurrent systems. Because such systems are complex to design and implement, we can rely on object-oriented technology. Hence, these systems may be based on several cooperative objects that are gathered into object-oriented application frameworks [Cotter-Potel 1995; Fayad-Schmidt 1997]. Furthermore, in order to enable distributed and concurrent systems to perform in a coherent and efficient manner, distributed artificial intelligence (DAI) technology can be used. The aim of DAI is to study a broad range of issues related to the distribution and coordination of knowledge and actions in environments involving multiple entities. These entities, called agents, are viewed as complex objects [Shoham 1993]. A particular type of agent, the software agent, has recently attracted much attention. From this emerges the idea of substituting objects by software agents inside a framework [Maamar 1997]. The result is what we call software agent-oriented frameworks.

SB4.1 Object-Oriented Application Frameworks

When several objects evolve in the same environment, it is important to initially define their behaviors in order to avoid conflicting situations. Therefore, several questions have to be answered: How to create objects? What kinds of information can be exchanged by objects? What types of communication channels can handle these exchanges? And, how can objects be associated to achieve common goals?

Authors in [Cotter-Potel 1995; Fayad-Schmidt 1997; Lewis 1995] introduce the notion of object-oriented application frameworks. A framework is viewed as an organized environment for running a collection of objects. It is also defined as a set of related classes which a designer can specialize and/or instantiate to implement specific applications.

An environment can be based on several distributed object-oriented application frameworks. To set up such an environment, it is important to follow a design approach—that is, an approach that allows designers to identify and characterize the appropriate frameworks with regard to the domain to be analyzed and to the application to be developed. The proposed approach is to decompose a problem in such a way that a collection of frameworks can work together in order to provide the solution [Cotter-Potel 1995]. However, this seems too simple, because, in fact, several issues need to be addressed. These issues include how the needed frameworks (numbers, types, and localizations) are determined, how their objects are defined, how their responsibilities are assigned, and how their functionalities are specified. Moreover, it isn't easy for designers to develop such frameworks and to monitor their evolution, particularly in distributed and heterogeneous contexts. One solution to overcome these issues is to expand object-oriented frameworks through the use of software agents.
SB4.2 Software Agent-Oriented Frameworks

Designing software agent-oriented frameworks is a recursive activity that is applied in its
turn to frameworks, teams of agents, and software agents [Meamar 1997].

A software agent-oriented framework offers a set of services that can be requested
either by users or by other frameworks. If a framework cannot carry out a service on its
own, it can require complementary services from other frameworks. A framework is
based on a supervisor agent and one or several teams of software agents. A bank of soft-
ware agents contains several agents having different functionalities that are specific to
the application to be developed.

Teams of software agents are set up by the framework and are structured differently
according to their responsibilities. A team is also characterized by a supervisor agent and
a set of roles that agents must fulfill according to their abilities. To fulfill their goals,
agents need to interact and exchange information.

Services provided by a framework satisfy specific users' needs, such as an informa-
tion retrieval. When a service is invoked by a user, the framework's supervisor agent
activates a realization scenario that specifies the interactions that are allowed between
frameworks, teams of agents, and software agents and the operations required to carry
out the service. According to this realization scenario, the framework supervisor creates
teams that will play roles specified in the scenario. At their own level, team supervisor
agents activate realization scenarios in order to coordinate the activities of their soft-
ware agents.

A realization scenario is composed of the following elements: software agents' types
and roles, scheduling sequences indicating in which order agents should perform their
activities, the list of internal and/or external information resources that provide knowl-
dge required by the agents to achieve their activities, and, finally, an evaluation proce-
dure used by the supervisor agent to monitor the operations of the other agents.

SB4.3 Summary

Recall the main characteristics of software agent-oriented frameworks: The frameworks
can interact in order to define a global behavior that is an outcome of such interactions.
The frameworks can be adapted in terms of components (types of teams and agents to
integrate) and functionalities (types of services to offer). The frameworks can play sev-
eral roles according to the application to be developed. Finally, in the frameworks, soft-
ware agents can be specified by knowledge structures and operating mechanisms. The
first category defines agents' knowledge base and the second category describes agents' behavior.

Continues
SB4.4 References


