Software architecture of the MinneTAC supply-chain trading agent

John Collins, Wolfgang Ketter, Maria Gini, and Amrudin Agovic
Dept. of Computer Science and Engineering, University of Minnesota

Tech report #08-031, October 2008

Abstract

The MinneTAC trading agent is designed to compete in the Supply-Chain Trading Agent Competition [4]. It is also designed to support the needs of a group of researchers, each of whom is interested in different decision problems related to the competition scenario. The design of MinneTAC breaks out each basic behavior into a separate, configurable component, and allows dynamic construction of analysis and modeling tools from small, single-purpose "evaluators". The agent is defined as a set of "roles", and a working agent is one for which a component is supplied for each role. This allows each researcher to focus on a single problem and work independently, and it allows multiple researchers to tackle the same problem in different ways. A working MinneTAC agent is completely defined by a set of configuration files that map the desired roles to the code that implements them, and that set parameters for the components. We describe the design of MinneTAC, and we evaluate its effectiveness in support of our research agenda and its competitiveness in the TAC-SCM game environment.

1 Introduction

One of the more compelling application areas for autonomous agents is in electronic commerce. Decisions can be relatively clear-cut (buy or sell, set a price, submit a bid, award bids, etc.), and communications among agents and between agents and their environments can be constrained and highly scripted.

Organized competitions can be an effective way to drive development and understanding in complex domains. Since we don't fully understand how to build an automated economic agent that will operate successfully in open, real-world economic environments, we can create slightly more constrained environments and carry out competitions in those environments. Examples related to electronic commerce include the Penn-Lehman Automated Trading Project [9] and the TAC travel game [20]. Another example is the Supply-Chain Management Trading Agent Competition [4] (TAC SCM), which engages agents in simultaneous buying, selling, production scheduling, and inventory management problems.

This paper describes the design of the MinneTAC trading agent, which has competed effectively in TAC SCM for several years. We have attempted to respond both to the challenges of the game scenario as well as to the need to support multiple relatively independent research efforts that are focused on meeting one or more of those challenges. We also evaluate the success of our design both in terms of the competitiveness of the agents that have been implemented with it, and in terms of its ability to support our research agenda.

In Section 2, we review the TAC SCM game scenario, focusing on the design challenges presented by that scenario. In Section 3 we review the "non-functional" design challenges that address organizational and research needs. Section 4 describes the design of our agent, and Section 5 represents an early attempt to evaluate the success of our design. We wrap up with a description of some relevant related work in Section 6 and conclude in Section 7.

2 Overview of the TAC SCM game

In a TAC SCM game, each of the competing agents plays the part of a manufacturer of personal computers. Agents compete with each other in a procurement market for computer components, and in a sales market for customers, as shown in Figure 1. A typical game runs for 220 simulated days over about an hour of real time. Each agent starts with no inventory and an empty bank account, and must borrow (and pay interest) to build up an initial parts inventory before it can begin assembling and shipping computers. The agent with the largest bank account at the end of the game is the winner.

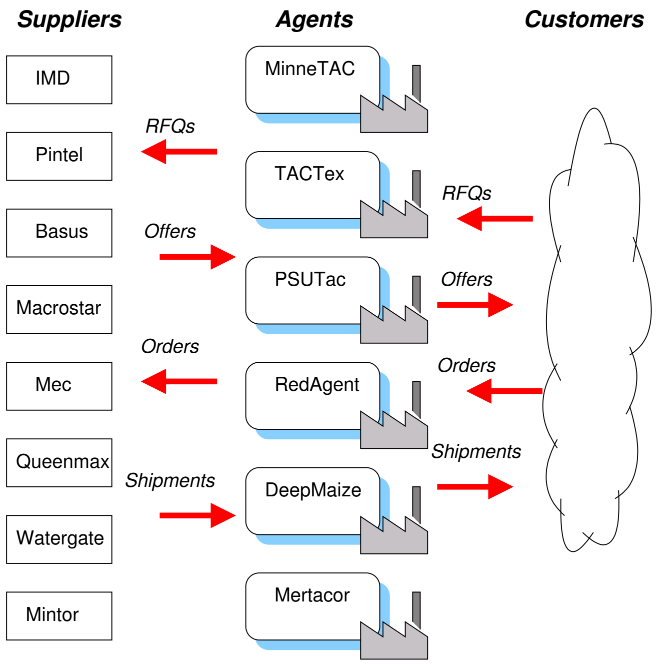

Figure 1: Schematic overview of a typical TAC SCM game scenario.

2.1 Game scenario

Customers express demand each day by issuing a set of RFQs for finished computers. Each RFQ specifies the type of computer, a quantity, a due date, a reserve price, and a penalty. Each agent may choose to bid on some or all of the day's RFQs. For each RFQ, the bid with the lowest price will be accepted, as long as that price is at or below the customer's reserve price. Once a bid is accepted, the agent is obligated to ship the requested products by the due date, or it must pay the stated penalty for each day the shipment is late. Agents do not see the bids of other agents, but aggregate market statistics are supplied to the agents periodically. Customer demand varies through the course of the game by a random walk.

Agents assemble computers from parts, which must be purchased from suppliers, and manage inventories of parts and finished goods. When agents wish to procure parts, they issues RFQs to individual suppliers, and suppliers respond with bids that specify price and availability. If the agent decides to accept a supplier's offer, then the supplier will ship the ordered parts on or after the due date (supplier capacity is variable). Supplier prices are based on current uncommitted capacity.

Once an agent has the necessary parts to assemble computers, it must schedule the assembly tasks in its finite-capacity production facility. Each computer model requires a specified number of assembly cycles. Assembled computers are added to the agent's finished-goods inventory, and may be shipped to customers to satisfy outstanding orders.

2.2 Agent decisions

An agent for the TAC SCM scenario must make four basic decisions during each simulated "day" in a competition. It must decide which customer RFQs to respond to, and set bid prices (Sales). It must decide what parts to purchase, from whom, and when to have them delivered (Procurement). It must schedule its manufacturing facility (Production). It must ship completed orders to customers (Shipping). These decisions are supported by models of the sales and procurement markets, and by models of the agent's own production facility and inventory situation. The details of these models and decision processes are the primary subjects of research for participants in TAC SCM. In particular, the Sales and Procurement markets are highly variable, and many of the important factors, such as the current capacity and outstanding commitments of suppliers, are not visible to the agents.

3 Design challenges

Beyond the challenges presented by the TAC SCM problem domain, our research needs present several additional issues. The most important is that our design must support multiple independent developers pursuing their own lines of research. The TAC SCM scenario presents a number of relatively independent decision problems, and there are many possible approaches to solving them. Our design must make it relatively easy for a researcher to focus on a particular subproblem without having to worry about getting a whole agent to work correctly. In addition, we expect to continue participating in TAC SCM over several years, and we want to avoid redesign and re-implementation over that time, even though we expect significant details of the game scenario to change from one year to the next.

Decision processes may involve somewhat arbitrary parameters, and their interactions and the sensitivity of agent performance to the settings of those parameters may not be well-defined. This is true even in cases where the agent is designed specifically to minimize the number of such parameters by use of optimization methods [13]. To understand these effects, we need to be able to configure agents with different combinations of decision processes and their underlying models and parameters.

Experimental research requires data. The TAC SCM game server keeps data from each game played, which may be used to understand and compare the performance of competing agents. However, it is also necessary to integrate game data with information about the agent's internal state during the game, in order to understand the detailed performance of agent decision processes. This suggests a need for a data logging capability that can be easily configured to extract needed data from a running agent, while keeping the size of log files under control.

4 The design of MinneTAC

To address the design challenges of the MinneTAC agent, we follow a component-oriented approach [18]. The idea is to provide an infrastructure that manages data and interactions with the game server, and cleanly separates behavioral components from each other. This allows individual researchers to encapsulate agent decision problems within the bounds of individual components that have minimal dependencies among themselves. Two pieces of software form the foundation of MinneTAC: the Apache Excalibur component framework, and the "agentware" package distributed by the TAC SCM game organizers. Excalibur provides the standards and tools to build components and configure working agents from collections of individual components, and the agentware package handles interaction with the game server.

4.1 A brief overview of Excalibur

Apache Excalibur [5] is a general-purpose component framework. It is widely used as a foundation for middleware and server sofware, such as the OpenORB CORBA implementation[1] and the Cocoon web appli-

[1]OpenORB.sourceforge.net

cation framework[2], but its use in the implementation of autonomous agents is rare. It does not provide the "classic" facilities for agent design, such as knowledge representation, inter-agent communication, reasoning facilities, or a planning infrastructure. Instead, it provides a means to build complex, robust systems from sets of role-based, configurable components. This satisfies a primary goal of MinneTAC, allowing researchers to work independently on individual decision problems with minimal need for detailed coordination with each other.

Excalibur components are independent entities, in the sense that they typically have very few dependencies on each other, and minimal, well-defined dependencies on the Excalibur framework itself. Components are coarse-grained entities, each typically composed of a number of classes. Control inversion puts primary control in the Excalibur "container", which loads components, sets up logfiles, configures the components, and starts any components that run independent threads.

Each Excalibur component is designed to fulfill a specific *role*, and an Excalibur system is a set of roles, each of which is mapped to a specific Java class. A role has a name, a set of responsibilities, and a well-defined interface. An Excalibur application is composed of the Excalibur infrastructure, a container that initializes the system, and the components specified by the configuration. Configuration files specify the specific roles, the classes that satisfy those roles, and configuration parameters for those classes. The container reads the configuration files, loads the specified classes, and invokes the Excalibur interfaces on each component.

4.2 MinneTAC architecture

Following Bass et al. [1], we use the term "architecture" to refer to the set of components that make up our system, along with their properties and relationships. A MinneTAC agent is a set of components layered on the Excalibur container, as shown in Figure 2. In the standard arrangement, four of these components are responsible for the major decision processes: Sales, Procurement, Production, and Shipping. In some configurations, an LpSolver component is included to provide optimization services. All data that must be shared among components is kept in the Repository, which acts as a blackboard [3]. The Oracle hosts a large number of smaller components that maintain market and inventory models, and do analysis and prediction. The Communications component handles all interaction with the game server. By minimizing couplings between the components, this architecture completely separates the major decision processes, thus allowing researchers to work independently. Ideally, each component depends only on Excalibur and the Repository.

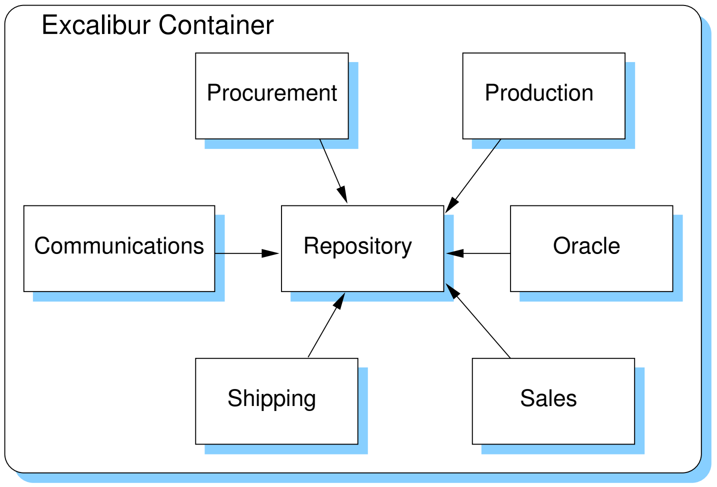

Figure 2: MinneTAC Architecure. Arrows indicate dependencies.

The agent opens four configuration files when it starts. Three of these are handled by the Excalibur

[2]cocoon.apache.org

infrastructure. The system configuration file specifies the set of roles that make up the system, along with the classes that implement those roles. The component configuration file specifies runtime configuration options for each component. For example, the Sales component may have a parameter that controls the maximum level of overcommitment of its existing inventory or capacity when it makes customer offers. The log configuration file controls the names and locations of log files that are produced by the running agent, the general format of log entries, and for each component, the level of detail to be logged. The agentware configuration file controls interaction with the game server (see Figure 8).

4.2.1 Events

A TAC Agent is basically a "reactive system" in the sense that it responds to events coming from the game server [21]. These events are in the form of messages that inform the agent of changes to the state of the world: Customer RFQs and orders, supplier offers and shipments, etc. The game is designed so that each simulated day involves a single exchange of messages; a set of messages sent from the game server to the agent, and a set returned by the agent back to the server. For example, from the standpoint of the agent, each day's incoming messages includes the set of customer RFQs for the day, and the return set of messages includes the agent's bids for those RFQs.

More specifically, Figure 3 shows the communication activity for a game day. The general pattern is that the game server sends out a set of messages representing supplier and customer activity, as well as inventory and bank-account status data, the agent deliberates for some time, and then the agent responds with a set of messages that respond to the customer RFQs, and supplier offers for the current day. The agent must also specify the production and shipping schedules for the following day, and it may issue additional supplier RFQs each day. The length of a simulation day is fixed by the server; in the standard tournament configuration, days are 15 seconds long.

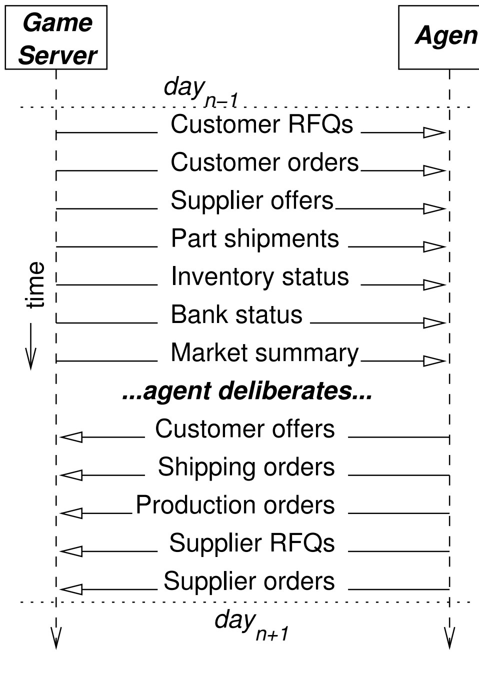

Figure 3: One day of communications activity between the game server and an agent.

As shown in Figure 3, the agent does not need to react to individual messages from the server. Instead, it waits until after all the day's messages have been received, and then considers all of them together. In fact, there is one additional end-of-data message not shown in the figure, which contains no data but simply tells the agent that the day's input messages are complete. MinneTAC handles all data messages by storing them in the Repository. When the end-of-data message is handled by the Repository, it notifies the other components that the day's data input is complete. Components use this notification as the signal to perform their deliberations and compose the day's return messages. In this interaction, the Repository acts as a Subject and the other components as Observers in the *Observer* pattern [6].

An important limitation of the Observer pattern is that the sequence of notifications is not controlled, although in most implementations it is repeatable. But the order of event processing is important for the MinneTAC decision processes. For example, it greatly simplifies the Sales decision process to know that the current day's Shipping decisions have already been made. To allow event sequencing without introducing new dependencies, two events are generated by the Repository for each day of a game. The *data-available* event is a signal to read the incoming messages and do basic data analysis. The subsequent *decision* event is a signal to make the daily decisions and post the outgoing messages back in the Repository. The decision event itself provides an additional level of sequence control by allowing components to "refuse" the event until one or more other components (identified by role names) have made their decisions. Components that have refused the event will receive it again once all other components have had an opportunity to process it. To ensure that Sales decisions are made after Shipping decisions, Sales must refuse to accept the decision event until after it sees "shipping" among the roles that have already processed it. No additional dependencies are introduced by this mechanism, since the role names are simply added to the event object itself, and the names come from a configuration file, not the code.

4.2.2 Evaluators

As indicated earlier in this section, a goal of the MinneTAC design is to minimize coupling between the various components. How, then, do they communicate, if they cannot depend on one another? One possible approach is the one used by the RedAgent team at McGill University [10], in which the components communicate through internal auction-based markets. Our approach is to use *evaluations* that are accessible through the various data elements in the Repository. The general idea is that when a component needs to make a decision, it will inspect the available data and run some utility-maximizing function. The available data consists of any data it maintains internally, and the data in the repository. Any data reductions or analyses that are performed on Repository data can be encapsulated in the form of Evaluations, and made available to other components. These analyses are implemented by the Oracle component through a configurable set of *evaluators*.

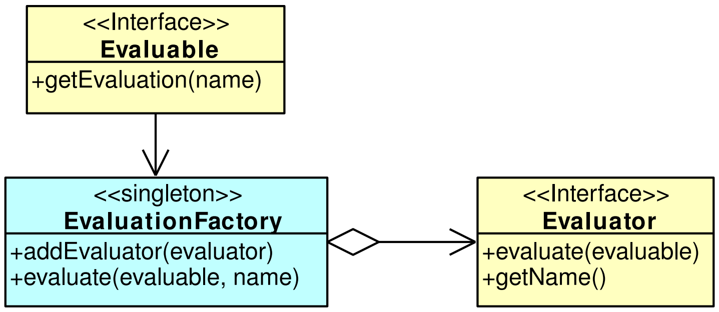

Figure 4: Evaluables and Evaluators.

All the major data elements in the Repository (such as RFQs, offers, orders, products, components, etc.) are Evaluable types. As shown in the UML class diagram in Figure 4, each Evaluable can be associated with some number of associated Evaluations. Each Evaluation has a name as well as a value. Also associated with each Evaluable is an EvaluationFactory, which maintains a mapping of Evaluation names to Evaluator instances, and is responsible for producing Evaluations when they are requested. It does this by inspecting the name of the requested Evaluation, looking up the Evaluator associated with that name, and invoking the *evaluate* method on the associated Evaluator, as shown in the UML sequence diagram of Figure 5.

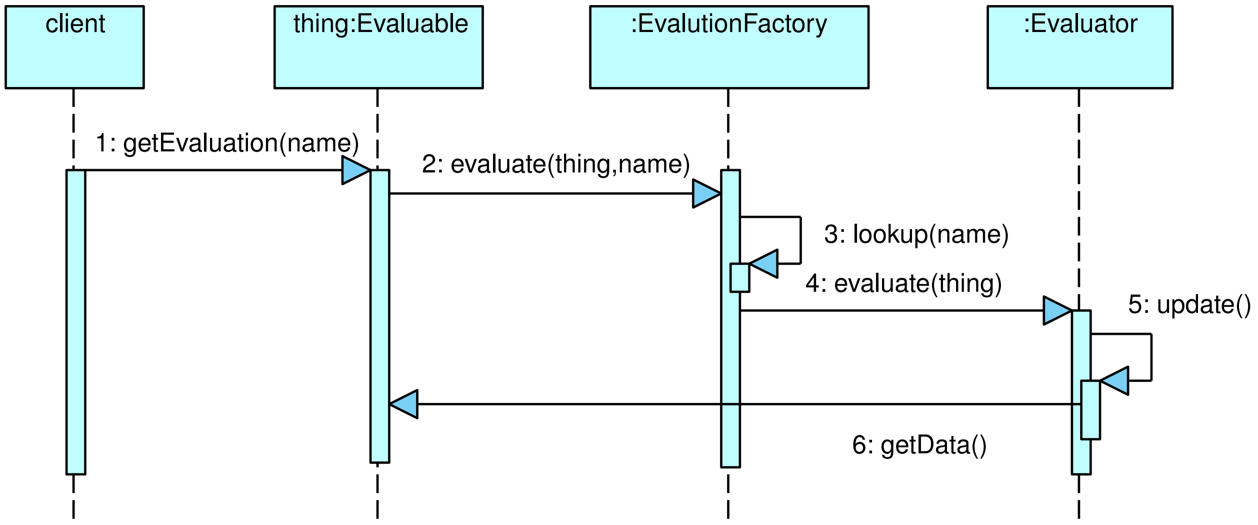

Figure 5: Processing an evaluation.

Evaluators are hosted by the Oracle component, which is responsible for loading and configuring Evaluators. Evaluators are registered with the Repository when they are configured, thus making them known to the EvaluationFactory. Because a given Evaluation may be requested multiple times during the agent's decision process, most Evaluators cache their results as long as they have reason to believe they remain valid. In most cases, Evaluator results remain valid until the next data-available event arrives.

Evaluations are *composable*. Evaluators implement back-chaining by requesting other Evaluations in the process of producing their results, and therefore many Evaluations are composed from other, presumably simpler, Evaluations. Evaluators are hosted by the Oracle component, which is responsible for loading and configuring Evaluators. Evaluators are registered with the Repository when they are configured, thus making them known to the EvaluationFactory. Because a given Evaluation may be requested multiple times during the agent's decision process, most Evaluators cache their results as long as they have reason to believe they remain valid. In most cases, Evaluator results remain valid until the next data-available event arrives.

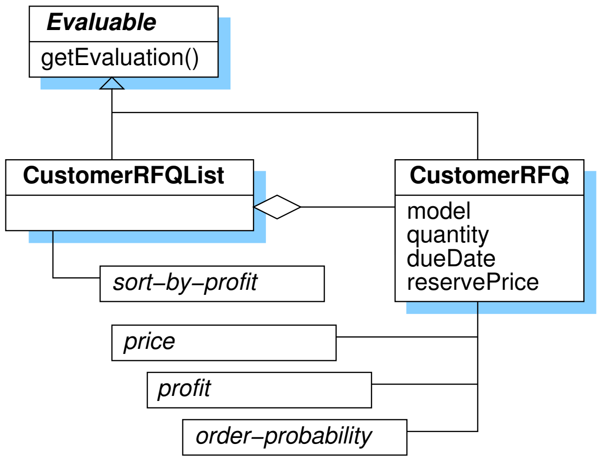

Figure 6: RFQ evaluation example.

Figure 6 shows a simple example of some Evaluable instances and a set of Evaluations that might be associated with them. The *price* evaluation might combine parts cost information with an estimate of current market conditions. The *profit* evaluation would need parts cost information and *price*. The *sort-by-profit* evaluation would need the *profit* evaluations on the individual RFQs.

4.3 MinneTAC components

A typical MinneTAC agent consists of the 7 components shown in Figure 2. Here we provide more detail on the three "core" components, the Repository, Communications, and Oracle components. The non-core components should be thought of as "role interfaces" since multiple implementations exist for each of them. To flesh out this concept, we also provide an in-depth look at one of our Sales component implementations, the price-driven sales manager used in the 2005 and 2006 competitions.

4.3.1 Repository

The Repository is the one component that is visible to all the other components, as required by the MinneTAC architecture. At the beginning of each day of the game, new incoming messages are deposited into the Repository. The event subsystem described in Section 4.2.1 is then used to notify other components to perform their analyses and decisions. The decision components retrieve data and evaluations from the repository, and record their decisions back into the repository. Finally, the resulting decisions are retrieved from the Repository by the Communications component and returned to the server.

Events are generated in response to state transitions. Figure 7 shows the state transitions and associated events in the Repository. The three events in the daily cycle have specific meanings, and are used to split daily processing into three distinct phases:

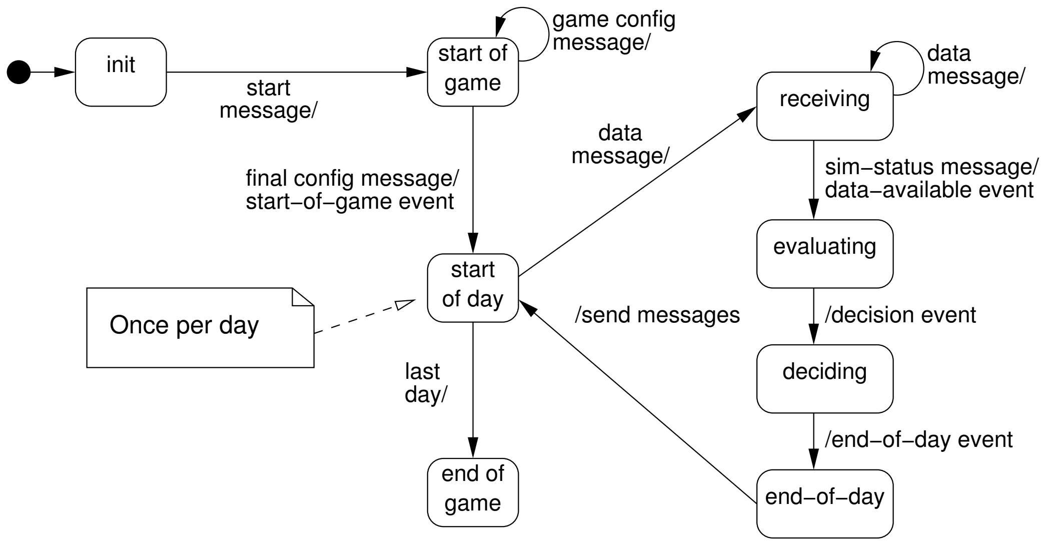

Figure 7: States and transitions in the Repository component.

data-available Components (including evaluators) are notified that the day's input from the server is complete. This is the time to read the data and do preliminary analyses that do not require interaction with evaluators.

decision Components are notified that their final decisions are needed. At this point, all preliminary data processing has been done, and so the evaluator chains can run without interruption.

end-of-day Some evaluators need to keep track of evaluation results from the previous day. This event is the signal that all evaluation results have already been computed and can be retrieved and stored without fear of recursively running loops in the evaluator chains.

The Repository plays the part of the Blackboard in the *Blackboard* pattern [3], and the remainder of the components, other than the Communications component, act as Knowledge Sources. However, the Control element of the Blackboard pattern is replaced by the Event and the Evaluable/Evaluator mechanisms.

4.3.2 Communications

In order to participate in a trading game, the agent must be able to communicate with the game server. The basic communication behaviors are implemented in the *agentware* package, provided by the game organizers. A common way to construct an agent for TAC SCM is to extend and subclass the elements in the agentware code directly. However, our team felt that this was a risky approach, since changes to the communication protocol could ripple through to changes in the agentware, necessitating new rounds of code-extraction and disruptions to our agent implementation. Our approach is therefore to simply "wrap" the entire agentware package with an Excalibur component that has responsibility for communications with the game server. The Communications component acts as an *Adapter* as described in [6]. We show this approach schematically in Figure 8. The result is that we avoid modifying the agentware package, and we are always able to use the latest version of the agentware by downloading it and importing it into our environment.

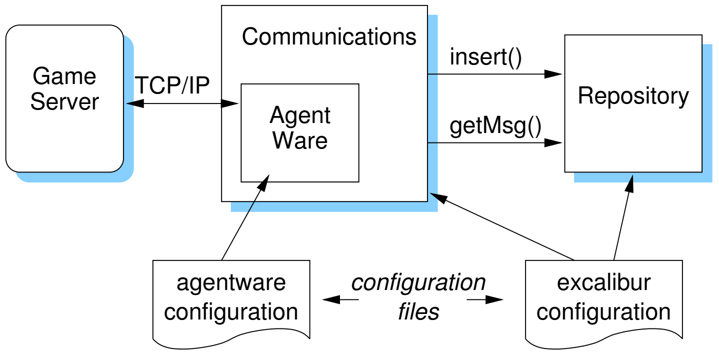

Figure 8: Communications component wraps the Agentware package.

4.3.3 Oracle

The Oracle component is essentially a meta-component, since its only purpose is to provide a framework for a set of small configurable components that may be used to perform analysis and prediction tasks. Most of these are Evaluators, but a few other types are supported as well. The Oracle itself simply reads its configuration data, and uses it to create and configure instances of Evaluators and other subclasses of ConfiguredObject. The top-level classes within the Oracle component are shown in Figure 9.

ConfiguredObject is an abstract class that has a name and an ability to configure itself, given an XML clause. The Oracle creates ConfiguredObject instances and keeps track of them by mapping their names to instances. When it starts, the Oracle processes a configuration clause that typically includes at least two subclauses. The first is a `<setup>` clause, which is processed at the time the Oracle is created, during agent initialization. At this time, objects can be created that do not need access to game parameters. A typical example is a model element that must read its initialization data from a file or database that has been created off-line, perhaps by analyzing prior games using machine learning techniques [11]. This must be done before the game begins simply because of the time required to set up these models; once the game begins, the agent must complete its work in less than 15 seconds each day. Other clauses are processed by the Oracle after the start-of-game event has been received, thus allowing objects to access game parameters

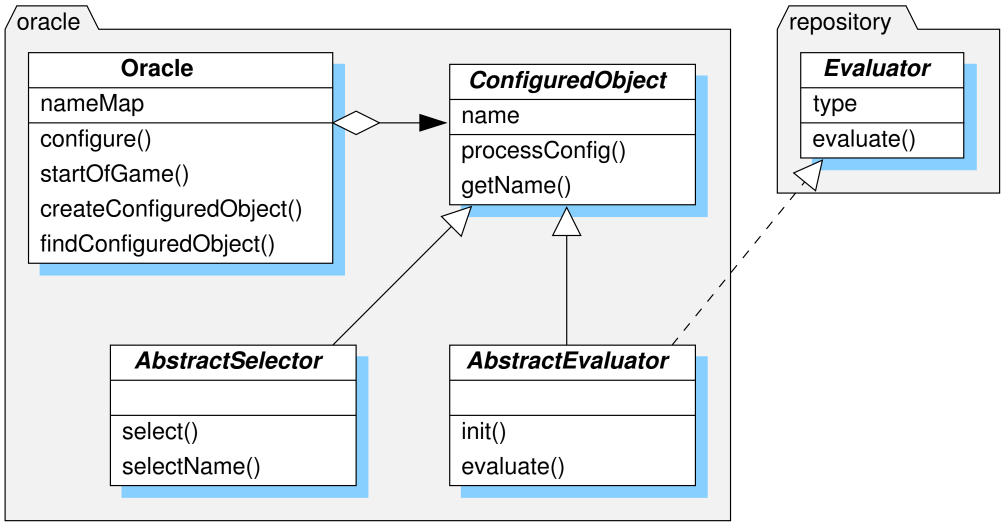

Figure 9: Principal classes in the Oracle component.

from the Repository when they are created. For example, many evaluators need to initialize themselves using data from the server's Component Catalog or Bill of Materials, which are sent to the agent at the start of a game.

Figure 10 illustrates the configuration clause for an Evaluator called "order-probability" that estimates the sales order probability for each product (the *order-probability* evaluator in Figure 11) by combining a median price estimate with a slope estimate. By convention, the output of any evaluator that promises to estimate order probability is an object called a `Pricer` that has two methods: `getPrice()` returns the predicted median price, and `getPriceForProbability(p)` returns the price corresponding to the given probability p. Inputs to this evaluator are two other evaluators, named "median-price" and "slope-estimate."

```
<evaluator class="edu.umn.cs.tac.oracle.eval.LinearOPEstimator"
           name="order-probability">
    <input median="median-price"
           slope="slope-estimate" />
</evaluator>
```

Figure 10: Configuration clause for an order-probability estimator that uses a median price and a slope estimate as data sources

The most common subclasses of ConfiguredObject are Evaluators and Selectors. We have discussed Evaluators at length in Section 4.2.2, and we shall see an extended example of their use in the next section. A Selector is simply a switch that can be used to select different models or evaluators in different game situations. For example, the early part of a game is typically characterized by customer prices that start high and fall rapidly as agents acquire parts and begin building up inventories. Later in the game, prices are less predictable and more sophisticated models may be useful. A simple DateSelector can be used to switch between pricing models at a particular preset date, or a more sophisticated Selector could be used to switch models once the initial price decline bottoms out. An interesting subtype of Selector is Mixer, which blends one model into another over a period of time, thereby eliminating sharp transitions. This can be important when feedback loops are being used to track prices, as described in the following section.

4.3.4 Sales

To illustrate the power of Evaluators, we show in Figure 11 the evaluation chain that is used to produce sales quotas and set prices in a relatively simple MinneTAC configuration. Each of the cells in this diagram is an Evaluator. A version of the Sales component called PriceDrivenSalesManager is conceptually very simple – it places bids on each customer RFQ for which the randomized-price evaluator returns a non-zero value. The core of this chain is the allocation evaluator, which composes and solves a linear program each game day that represents a combined product-mix and resource-allocation problem that maximizes expected profit. The objective function is

$$\Phi = \sum_{d=0}^{h} \sum_{g \in \mathcal{G}} \Phi_{d,g} A_{d,g} \tag{1}$$

where Φ is the total profit over some time horizon h, $\mathcal{G}$ is the set of goods or products that can be produced by the agent, $\Phi_{d,g}$ is the (projected) profit for good g on day d, and $A_{d,g}$ is the allocation or "sales quota" for good g on day d. The constraints are given by evaluators *available-factory-capacity*, the current day's *effective-demand*, projected *future-demand*, and by Repository data, such as existing and projected inventories of parts and finished products, and outstanding customer and supplier orders. Predicted profit per unit for each product type is the difference between *median-price* and *cost-basis* for those products.

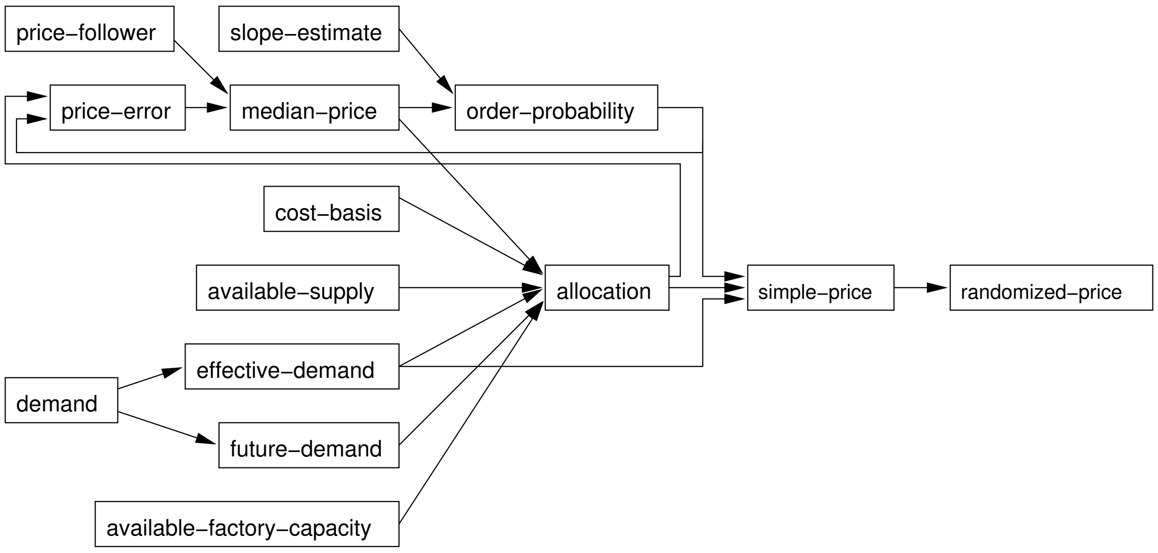

Figure 11: Evaluator chain for sales quota and pricing.

The output of the allocation evaluator is sales quotas for each product. Given a sales quota for a given product and a representation of an order-probability function, the simple-price evaluator computes a price that is expected to sell the desired quota, assuming that price is offered on all the demand for that product. In other words, if the quota is 25 units and the demand is for 100 units, simple-price computes a price that is expected to be accepted by only 25% of the customers. Since there is some uncertainty in the predictions of price and order probability, randomized-price adds a slight variability to offer prices. This improves the information content and reduces variability of the returned orders.

Market prices are tracked by the *price-follower* evaluator, which observes the daily high prices reported by the server. The price-follower implements a straightforward double-exponential smoothing function

$$\begin{aligned} price_d^{sm} &= \alpha y_d + (1-\alpha)(price_{d-1}^{sm} + trend_{d-1}) && (2) \\ trend_d &= \gamma(price_d^{sm} - price_{d-1}^{sm}) + (1-\gamma) trend_{d-1} && (3) \end{aligned}$$

where y_d is the observed high price for a given product on day d (the highest price at which the product was sold on the previous day), $price_d^{sm}$ is the smoothed price for the current day d, $trend_d$ is the trend for the current day, and α and γ are the smoothing parameters. The resulting smoothed price estimate is too high to support sales, since it is tracking the daily high price, and it depends strongly on the detailed behavior of other agents. Therefore, we use a feedback mechanism to adjust our price estimates, as shown graphically in Figure 12. Each day d, the *order-probability* evalutor generates a pricing function $P_d(order|price)$, and $price^{est}$ is the price computed for yesterday's sales quotas Q_{d-1}. Yesterday's observed price $price^{obs}$ is computed by applying yesterday's pricing function to today's customer orders O_d. The correction computed by *price-error* is the difference between estimated and observed prices

$$err_p = price^{obs} - price^{est} \tag{4}$$

The *median-price* evaluator then computes a median price

$$price^{med} = price^{sm} + err_p \tag{5}$$

for the current day, giving the corrected output of *price-follower*.

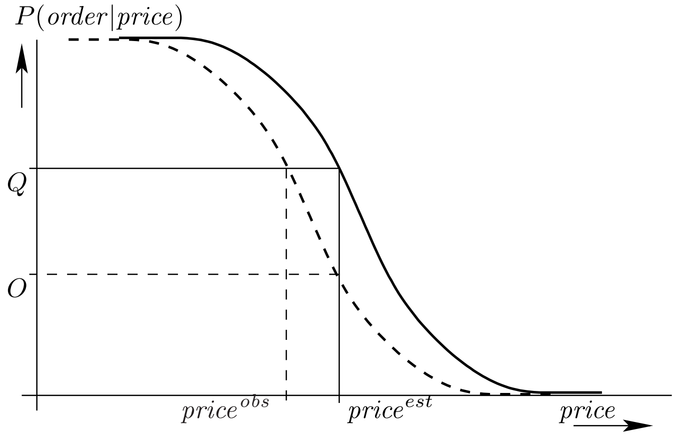

Figure 12: Estimating actual market price $price^{act}$, given sales quota Q, order volume O and an estimate of the order probability function P.

Another compelling example of the power of evaluators in the design of MinneTAC is the sales pricing and forecasting model based on Gaussian Mixture Models described in [11]. The elements of this model replace the *price-follower*, *median-price*, and *order-probability* evaluators shown in Figure 11, and the *slope-estimate* evaluator is omitted in that configuration. There are also two ConfiguredObject types that load training data for this model when the agent starts. This is the configuration that ran in the 2006 TAC SCM tournament; the details are beyond the scope of this paper.

5 Evaluation

The software architecture of MinneTAC is strongly focused on strict control of dependencies, and on flexible configuration. We evaluate the success of this design by asking two questions: (1) Does the agent perform well, and to what extent does the design affect agent performance? (2) Does the design meet the "usability" challenges described in Section 3?

5.1 Performance

There are two measures of agent performance that could be affected by the design. One is related to overhead: Does the design impose an unacceptable runtime overhead? The other is how well the agent performs in competition against other agents that have been implemented with different designs.

The Excalibur framework does indeed impose some overhead when the agent starts up, since it must read configuration files, find and load code for components, and set up and configure the components. However, once the agent is running, there is essentially no overhead imposed by the framework. Event processing and evaluation is done by direct lookup, since event handlers and Evaluators are registered when components are loaded. We have run 6 MinneTAC agents (with a simple Sales component) on the same desktop machine (a 1 GHz Pentium), and all 6 agents are able to complete their daily decision procedures in less than 1 second. A Sales component that relies on solving a linear program each round takes longer, but its performance is almost entirely determined by the time required to compose and solve the linear program.

MinneTAC has done reasonably well in the official TAC SCM tournaments since 2003. In 2005 and 2006 it was a finalist. Each year, MinneTAC has been fielded with a new implementation of at least one of the decision components (Sales, Procurement, Production, and Shipping), and several others have been implemented but have not been entered into the competition. The ease with which these new components could be implemented and configured into the agent is a testament to the design we describe here.

5.2 Usability

Mitch Kapor says that software design bridges the world of people and human purpose with the world of technology [8]. It's easy to understand what that means when the artifact is a desktop application intended to be used directly by people. But in fact, the first user of a software design is the programmer who implements it, and the programmer who must change it later is perhaps even more strongly impacted. From the implementer's standpoint, good design is easy to understand, easy to implement correctly, and easy to maintain. From the maintainer's standpoint, a good design is one that is easily understood, and that can sustain change without losing integrity.

MinneTAC is an autonomous agent. It exists in the game environment and needs no user interface, other than the configuration files it reads and the logfiles it produces. The principal usability criterion is whether researchers can effectively work on the various decision problems independently, and whether they can extract the data they need to analyze performance and confirm or refute hypotheses.

There is considerable evidence that our design has met its goals.

- Inexperienced student programmers have been able to contribute significant functionality without needing to understand the entire system. Examples include two different Shipping components, two different Production components, five different Sales components, six different Procurement components, and over 80 different Evaluators, written by at least 22 students over a period of four years.

- The standardized log-message format produced by the Excalibur infrastructure makes data extraction relatively easy, even though MinneTAC generates approximately 5Mb of data for a typical game. A wide variety of analyses have been carried out with this data. An example of such an analysis is given in [12], where we were able to show that the original design of the game gave a large advantage to the agent that won a procurement lottery on the first day of the game.

- One student implemented a user interface that takes full advantage of the component-oriented architecure of MinneTAC and its reliance on Evaluators to do much of the analysis. It provides several types of plotting windows. Its layout and the content of the windows are entirely determined by its configuration, which specifies the plot types and which evaluators are to be plotted.

- Selectors and Mixers (see Section 4.3.3) are very recent additions to the MinneTAC design, and they add considerable expressive power for constructing models that can learn and adapt to market conditions. Once the need was clear, they were added and tested in less than four hours, and required no other changes in the rest of the system.

- MinneTAC is an open-source project, available at `http://tac.cs.umn.edu`. The source release includes the full infrastructure, and relatively simple examples of each of the decision components, a few evaluators, and a sample set of configuration files. It has been downloaded about 30 times per month since its initial release in April 2005. There have also been over 500 binary downloads of the 2005 and 2006 competition versions of MinneTAC, which include some relatively complex evaluators that are not sufficiently documented or tested for release to the public.

6 Related Work

Most agent design efforts have focused on either the autonomous behavior aspects of agency, or on interaction among agents. Norman *et al.* [15] describe agent societies that model organizational structures and automate business processes. These ADEPT agents negotiate over service agreements that can involve many parties and many dimensions. JADE [14] is an agent framework that has been used to build trading agents, and could have been used for MinneTAC. However, its primary emphasis is on building multi-agent systems that comply with FIPA specifications for inter-agent communications, and with flexible deployment in a network environment. This is not a requirement for the TAC SCM domain. The MinneTAC design is *compositional* in the sense of Brazier *et al.* [2], but not hierarchically so. The DESIRE method from Brazier *et al.* does not seem applicable to the MinneTAC situation, since we are dealing with a single agent in an existing environment, and the blackboard approach used in MinneTAC is not easily modeled with DESIRE. RETSINA [17] suggests both a multi-agent architecture with a variety of agent roles, and an architecture for individual agents that provides communications, planning, scheduling, and execution monitoring. This architecture could probably be adapted to the TAC SCM domain, but its planning and communication capabilities would not be especially useful. Vetsikas and Selman [19] show a method for studying design tradeoffs in a trading agent. This approach could likely be used effectively in MinneTAC.

A few of the other participants in TAC SCM have described their agent designs. He *et al.* [7] have adopted a design consisting of three internal “agents” to handle Sales, Procurement, and Production/Shipping. Sales decisions use a fuzzy logic module. Some algorithmic detail is given, but there is little further detail on the architecture of the agent. TacTex05, the winner of the 2005 competition [16] is based on two major modules, a Supply Manager that handles procurement, and a Demand Manager that handles sales, production, and shipping. These modules are supported by a supplier model, a customer demand model, and a pricing model that estimates sales order probability.

Ultimately, the TAC SCM problem domain does not require the sort of flexible cognitive and social elements of these more “traditional” agent designs. Instead, our focus has been on separating the decision tasks and supporting research needs, and we have found the component-oriented model to be ideal.

7 Conclusions and Future Work

Experimental work with multi-agent systems requires an implementation. Often, the design qualities that best support experimental work are different from those normally considered “ideal” in industry. In complex economic scenarios such as the Supply Chain Trading Agent Competition, the desired design qualities include clean separation of infrastructure from decision processes, ease of implementation of multiple decision processes, clean separation of different decision processes from each other, and controllable generation of experimental data. In a competition environment, the ability to compose multiple agents with different combinations of decision process implementations makes it possible to test hypotheses about the effectiveness of competing decision models.

We show one way to construct such an agent, using a readily-available component framework. The framework provides the ability to compose agent systems from sets of individual components based on simple configuration files. We also show that two basic mechanisms, event distribution with the Observer pattern and our Evaluator-Evaluation scheme, permit an appropriate level of component interaction without introducing unnecessary coupling among components.

There are many possible extensions to the basic design we show here. One that we are currently pursuing is to add an “executive” component that would allocate “resources” to competing implementations of basic decision processes within a single agent. This would allow a high degree of adaptability in the game environment, where the level of demand can fluctuate greatly, and where the actions of other agents can have a significant impact on the markets.

8 Acknowledgement

We would like to thank the organizers of the TAC SCM game for a stimulating research problem. Partial support for this research is gratefully acknowledged from the National Science Foundation under award NSF/IIS-0414466.

References

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[2] Frances M. T. Brazier, Catholijn M. Jonker, and Jan Treur. Principles of component-based design of intelligent agents. *Data and Knowledge Engineering*, 41(1):1–28, April 2002.

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: a System of Patterns*. Wiley, 1996.

[4] John Collins, Raghu Arunachalam, Norman Sadeh, Joakim Ericsson, Niclas Finne, and Sverker Janson. The supply chain management game for the 2006 trading agent competition. Technical Report CMU-ISRI-05-132, Carnegie Mellon University, Pittsburgh, PA 15213, November 2005.

[5] Apache Software Foundation. Apache excalibur. http://excalibur.apache.org/, 2006.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[7] Minghua He, Alex Rogers, Xudong Luo, and Nicholas R. Jennings. Designing a successful trading agent for supply chain management. In *Proc. of the Fifth Int'l Conf. on Autonomous Agents and Multi-Agent Systems*, pages 1159–1166, May 2006.

[8] Mitchell Kapor. A software design manifesto. In Terry Winograd, editor, *Bringing Design to Software*, pages 1–9. ACM Press, 1996.

[9] Michael Kearns and Luis Ortiz. The Penn-Lehman Automated Trading Project. *IEEE Intelligent Systems*, pages 22–31, 2003.

[10] Philipp Keller, Felix-Olivier Duguay, and Doina Precup. Redagent-2003: An autonomous market-based supply-chain management agent. In *Proc. of the Third Int'l Conf. on Autonomous Agents and Multi-Agent Systems*, pages 1180–1187, New York, NY, USA, July 2004. ACM, ACM Press.

[11] Wolfgang Ketter, John Collins, Maria Gini, Alok Gupta, and Paul Schrater. Identifying and forecasting economic regimes in TAC SCM. In Han La Poutré, Norman Sadeh, and Sverker Janson, editors, *Agent-Mediated Electronic Commerce: Designing Trading Agents and Mechanisms*, volume 3937 of *Lecture Notes in Artificial Intelligence*, pages 113–125. Springer-Verlag, 2006.

[12] Wolfgang Ketter, Elena Kryzhnyaya, Steven Damer, Colin McMillen, Amrudin Agovic, John Collins, and Maria Gini. Analysis and design of supply-driven strategies in TAC-SCM. In *Workshop on Trading Agent Design and Analysis at the Third Int'l Conf. on Autonomous Agents and Multi-Agent Systems*, pages 44–51, New York, July 2004.

[13] Christopher Kiekintveld, Michael P. Wellman, Satinder Singh, Joshua Estelle, Yevgeniy Vorobeychik, Vishal Soni, and Matthew Rudary. Distributed feedback control for decision making on supply chains. In *Fourteenth International Conference on Automated Planning and Scheduling*, Whistler, BC, Canada, June 2004. AAAI, AAAI Press.

[14] P. Moraitis, E. Petraki, and N. Spanoudakis. Engineering JADE agents with the Gaia methodology. In R. Kowalszyk, J. Miller, H. Tianfield, and R. Unland, editors, *Agent Technologies, Infrastructures, Tools, and Applications for e-Services*, volume 2592 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, 2003.

[15] Timothy J. Norman, Nicholas R. Jennings, Peyman Faratin, and E. H. Mamdani. Designing and implementing a multi-agent architecture for business process management. In M. J. Wooldridge, J. P. Müller, and N. R. Jennings, editors, *Intelligent Agents III*, volume 1193 of *Lecture Notes in Artificial Intelligence*, pages 261–275. Springer-Verlag, Berlin, 1997.

[16] David Pardoe and Peter Stone. Tactex-05: A champion supply chain management agent. In *Proc. of the Twenty-First Nat'l Conf. on Artificial Intelligence*, pages 1389–1394, Boston, Mass., July 2006. AAAI.

[17] Katia Sycara and Anandeep S. Pannu. The RETSINA multiagent system: towards integrating planning, execution, and information gathering. In *Proc. of the Second Int'l Conf. on Autonomous Agents*, pages 350–351, 1998.

[18] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, 1998.

[19] Joannis A. Vetsikas and Bart Selman. A principled study of the design tradeoffs for autonomous trading agents. In *Proc. of the Second Int'l Conf. on Autonomous Agents and Multi-Agent Systems*, 2003.

[20] Michael P. Wellman, Peter R. Wurman, K. O'Malley, R. Bangera, S. Lin, D. Reeves, and William E. Walsh. Designing the market game for a trading agent competition. *IEEE Internet Computing*, 5(2):43–51, 2001.

[21] R. J. Wieringa. *Design Methods for Reactive Systems*. Morgan Kaufmann, San Francisco, 2003.