ICS-01M-927

Intelligent Agents for Object Model Creation Process in Object-Oriented Analysis and Design

指導教官　　B.H. Far 助教授

平成 13 年 2 月 13 日

理工学研究科情報システム工学専攻 R3927

Romi Satria Wahono

埼玉大学工学部情報システム工学 Far 研究室

埼玉県浦和市下大久保 255

ICS-01M-927

Intelligent Agents for Object Model Creation Process in Object-Oriented Analysis and Design

By

Romi Satria Wahono

Supervisor: Behrouz Homayoun Far

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

Master of Engineering

of the

Department of Information and Computer Sciences
Graduate School of Science and Engineering
Saitama University

February 2001

Contents

List of Figures

List of Tables

Abstract

Object oriented analysis and design has now become a major approach in the design of software systems. The state of object-oriented analysis and design is evolving rapidly. There are numerous object-oriented analysis and design methods being advocated at the present time. The first object-oriented analysis and design methods appeared in the late of 1980's and early 1990's. Similar to the growth of structured methods we saw a number of methods appear, all fairly similar but with significant differences in approach and notation. Each method had a camp of supporters, who usually made a living from training and consulting based on their approach. In the 1996's, two of the major methods gurus, Ivar Jacobson and James Rumbaugh, joined a third, Grady Booch at Rational and defined a common Unified Modeling Language (UML). In addition the Object Management Group (OMG) had begun a process to come up with a standard meta-model and notation for analysis and design.

The challenges of object-oriented analysis and design are, to identify the objects and their attributes needed to implement the software, describe the associations between the identified objects, define the behavior of the objects by describing the function implementations of each object, and refine objects and organize classes by using inheritance to share common structure. The object identification and refinement process are very important process in object-oriented analysis and design, and we called this process by *object model creation process*. Researchers and software designers come to a conclusion that object model creation process is an ill-defined task, regarding of the difficulties of heuristic and there is no unified methodology for object-oriented software

analysis and design. This is mainly due to lack of formalism for object-oriented software analysis and design.

In our project, we are developing an intelligent agents system that aims to help designers while designing object-oriented software by automating the difficulties and ill-defined tasks in the object model creation process, including identification of objects, associations, attributes, behaviors, and organization of objects with inheritance. First of all, we propose formal models of the object model creation process. And then we formulate design patterns and rules for solving above problems, and store them in the agent's knowledge bases. This system was named *OOExpert*.

Chapter 1
Introduction

In this chapter, we give a brief introduction and overview to the two major topics covered in this thesis: object-oriented analysis and design, and intelligent agents. It starts with a short introduction of the object-oriented paradigm, as frameworks rely heavily on its mechanisms such as object, class, inheritance, polymorphism and so on. And then, we will take a look at object-oriented analysis and design, and its problem that motivate us to do research on this topic. We present the key attributes of intelligent agents such as autonomy, mobility, and intelligence, and also provide the benefits and taxonomy of various intelligent agents technology. The research motivations and objectives are also presented at the end of this chapter.

1.1 Overview Of Object-Oriented Analysis And Design

Object oriented analysis and design has now become a major approach in the design of software systems. The state of object-oriented analysis and design is evolving rapidly. There are numerous object-oriented analysis and design methods being advocated at the present time. As Rentsch **[Rentsch, 1982]** predicted in 1982, "My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is". Booch **[Booch, 1991]** feels that: "Because object-oriented analysis and design is a relatively young practice, a discipline for effectively applying the elements of the object model has not yet emerged".

The first object-oriented analysis and design methods appeared in the late of 1980's and early 1990's. Similar to the growth of structured methods we saw a number of methods appear, all fairly similar but with significant differences in approach and notation. Each method had a camp of supporters, who usually made a living from training and consulting based on their approach. In the 1996's, two of the major methods gurus, Ivar Jacobson and James Rumbaugh, joined a third, Grady Booch at Rational and defined a common Unified Modeling Language (UML). In addition the Object Management Group (OMG) had begun a process to come up with a standard meta-model and notation for analysis and design.

> **Definition 1.1** (*Object-Oriented Analysis and Design*): Object-oriented analysis and design is a way of thinking about problems using models organized around real-world concepts.

1.1.1 Object-Orientation Concepts

Object

As the name object-oriented implies, objects are key to understanding object-oriented

technology. We can look around we know and see many examples of real-world objects: our dog, our desk, our television set, our bicycle. These real-world objects share two characteristics: they all have *attribute* and they all have *behavior*. For example, dogs have attribute (name, color, breed, hungry) and dogs have behavior (barking, fetching, and slobbering on our newly cleaned slacks). Bicycles have attribute (current gear, current pedal cadence, two wheels, number of gears) and behavior (braking, accelerating, slowing down, changing gears).

Software objects are modeled after real-world objects in that they, too, have attribute and behavior. A software object maintains its attribute in *variables* and implements its behavior with *methods*. So we can define, an object is a software bundle of variables and related methods. We can represent real-world objects using software objects. We might want to represent real-world dogs as software objects in an animation program or a real-world bicycle as a software object in the program that controls an electronic exercise bike. However, we can also use software objects to model abstract concepts. For example, an event is a common object used in GUI window systems to represent the action of a user pressing a mouse button or a key on the keyboard.

Everything that the software object knows (attribute) and can do (behavior) is expressed by the variables and methods within that object. A software object that modeled our real-world bicycle would have variables that indicated the bicycle's current attribute: its speed is 10 mph, its pedal cadence is 90 rpm, and its current gear is the 5th gear. These variables and methods are formally known as instance variables and instance methods to distinguish them from class variables and class methods.

In many programming languages, an object can choose to expose its variables to other objects allowing those other objects to inspect and even modify the variables. Also, an object can choose to hide methods from other objects forbidding those objects from invoking the methods. An object has complete control over whether other objects can access its variables and methods and in fact, can specify which other objects have access.

> **Definition 1.2** (*Object*): Object is the principal building blocks of object-oriented programs. Each object is a programming unit consisting of attribute (instance variables) and behavior (instance methods). An object is a software bundle of variables and related methods.

Class

In the real world, we often have many objects of the same kind. For example, our bicycle is just one of many bicycles in the world. Using object-oriented terminology, we say that our bicycle object is an instance of the class of objects known as bicycles. Bicycles have some attribute (current gear, current cadence, two wheels) and behavior (change gears, brake) in common. However, each bicycle's attribute is independent of and can be different from other bicycles. When building bicycles, manufacturers take advantage of the fact that bicycles share characteristics by building many bicycles from the same blueprint. It would be very inefficient to produce a new blueprint for every individual bicycle they manufactured.

In object-oriented software, it's also possible to have many objects of the same kind that share characteristics: rectangles, employee records, video clips and so on. Like the bicycle manufacturers, we can take advantage of the fact that objects of the same kind are similar and we can create a blueprint for those objects. Software "blueprints" for objects are called classes. So, we can define a class is a blueprint or prototype that defines the variables and methods common to all objects of a certain kind. For example, we could create a bicycle class that declares several instance variables to contain the current gear, the current cadence, and so on, for each bicycle object. The class would also declare and provide implementations for the instance methods that allow the rider to change gears, brake, and change the pedaling cadence.

The values for instance variables are provided by each instance of the class. So, after we've created the bicycle class, we must instantiate it (create an instance of it) before we can use it. When we create an instance of a class, we create an object of that type and the system allocates memory for the instance variables declared by the class. Then we can invoke the object's instance methods to make it do something. Instances of the same class share the same instance method implementations (method implementations are not duplicated on a per object basis), which reside in the class itself.

In addition to instance variables and methods, classes can also define class variables and class methods. We can access class variables and methods from an instance of the class or directly from a class. We don't have to instantiate a class to use its class variables and methods. Class methods can only operate on class variable, and they do not have access to instance variables or instance methods.

The system creates a single copy of all class variables for a class the first time it encounters the class in a program. All instances of that class share its class variables. For example, suppose that all bicycles had the same number of gears. In this case defining an instance variable for number of gears is inefficient. Each instance would have its own copy of the variable, but the value would be the same for every instance. In situations such as this, we could define a class variable that contains the number of gears. All instances share this variable. If one object changes the variable, it changes for all other objects of that type.

We probably noticed that the illustrations of objects and classes look very similar to one another. And indeed, the difference between classes and objects is often the source of some confusion. In the real world it's obvious that classes are not themselves the objects that they describe, a blueprint of a bicycle is not a bicycle. However, it's a little more difficult to differentiate classes and objects in software. This is partially because software objects are merely electronic models of real-world objects or abstract concepts in the first place. But it's also because many people use the term "object" inconsistently and use it to refer to both classes and instances.

Objects provide the benefit of modularity and information hiding. Classes provide the benefit of reusability. Bicycle manufacturers reuse the same blueprint over and over again to build lots of bicycles. Software programmers use the same class, and thus the same code, over and over again to create many objects.

> **Definition 1.3** (*Class*): A class is a blueprint or prototype that defines the variables and methods common to all objects of a certain kind.

Inheritance

Each subclass inherits attribute from the superclass. Mountain bikes, racing bikes, and tandems share some attributes: cadence, speed, and the like. Also, each subclass inherits methods from the superclass. Mountain bikes, racing bikes, and tandems share some behaviors: braking and changing pedaling speed, for example.

However, subclasses are not limited to the attribute and behaviors provided to them by their superclass. What would be the point in that? Subclasses can add variables and methods to the ones they inherit from the superclass. Tandem bicycles have two seats and two sets of handle bars; some mountain bikes have an extra set of gears with a

lower gear ratio. Subclasses can also override inherited methods and provide specialized implementations for those methods. For example, if we had a mountain bike with an extra set of gears, we would override the "change gears" method so that the rider could actually use those new gears.

We are not limited to just one layer of inheritance. The inheritance tree, or class hierarchy, can be as deep as needed. Methods and variables are inherited down through the levels. In general, the further down in the hierarchy a class appears, the more specialized its behavior. Subclasses provide specialized behaviors from the basis of common elements provided by the superclass. Through the use of inheritance, programmers can reuse the code in the superclass many times. Programmers can implement superclasses called abstract classes that define "generic" behaviors. The abstract superclass defines and may partially implement the behavior but much of the class is undefined and unimplemented. Other programmers fill in the details with specialized subclasses.

> **Definition 1.4** (*Inheritance*): Inheritance is a mechanism for sharing attributes and behaviors among classes based on a hierarchical relationship.

Encapsulation

Encapsulation is the mechanism by which related data and procedures are bound together within an object. In effect, an object is software capsule that functions as a black box, responding to messages from other objects and dispatching messages of its own in ways that do not reveal its internal structure. In this way, encapsulation supports and extends the proven principle of information hiding. Information hiding is valuable because it prevents local changes from having global impact. In the case of objects, it allows the implementations of individual objects to be altered without affecting the way these objects communicate through messages.

Ideally, an object should not only encapsulate data and methods, it should also hide the very distinction between the two. This allows developers to change implementations from data to methods or vice versa, without affecting the way of object interacts with other objects. In practice, this is achieved by declaring all variables to be private, or hidden from view outside of the object. When another object needs to see or change the value of a variable, it does so by way of an access method. In most object languages, methods as well as data may be declared to be private. This allows the internal

operations of an object to be hidden from view. A well-designed object exposes the smallest feasible portion of its methods as public to make them available to messages from other objects. This approach offers the greatest flexibility in terms of future changes to the object.

> **Definition 1.5** (*Encapsulation*): Encapsulation is the concept of the localization of knowledge within a module. Because objects encapsulate data and implementation, the user of an object can view the object as a black box that provides services. Instance variables and methods can be added, deleted, or changed, but as long as the services provided by the object remain the same, code that uses the object can continue to use it without being rewritten.

Polymorphism

The fact that different objects can respond to the same message in different ways is known as *polymorphism*. The power of polymorphism is that it greatly simplifies the logic of programs. A requestor no longer has to use nested IF statements or complex CASE statements to call the appropriate procedure. Instead, the proper procedure is automatically invoked by sending the request to a particular object. Polymorphism is often portrayed as advanced concept in object technology, but it is really a highly intuitive mechanism. More often than not, it is the technical staff steeped in the tradition of unique functions that have difficulty with the concept. Non-programmers grasp it quite readily because it reflects the natural form of human communication.

> **Definition 1.6** (*Polymorphism*): Polymorphism means that the same behavior may behave differently on different classes.

1.1.2 Object-Oriented Analysis and Design and Its Problems

[Booch, 1991] offers precise definitions of object-oriented analysis and design:
Object-oriented analysis (OOA) is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.
Object-oriented design (OOD) is a method of design encompassing the process of object-oriented decomposition and a notation for depicting logical and physical as well

as static and dynamic models of the systems under design.

OOA involves problem definition and design focuses on solution specification. OOD transforms the problem representation into a solution representation. Figure 1.1 depicts the relationship between OOA and OOD. The problem and solution domain representations are different and smaller than real-world problem. And the solution domain includes everything in the problem domain, plus any additional constructs required by the solution. However, it is difficult to determine where OOA ends and OOD begins, because of the blundered distinction between analysis and design in the object paradigm.

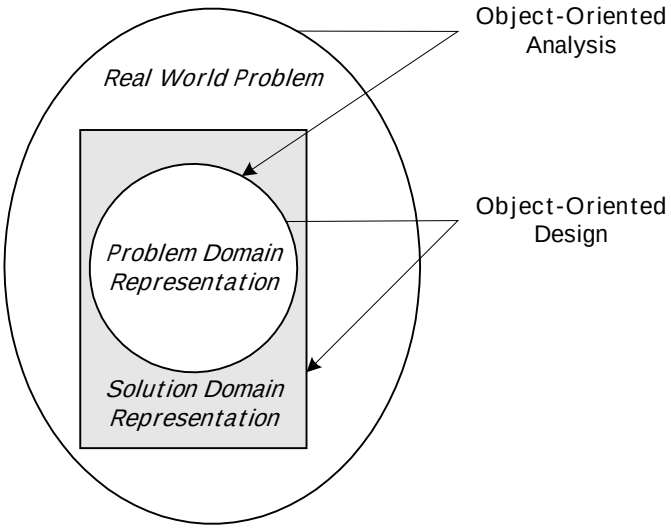

Figure 1.1: The Relationship Between Object-Oriented Analysis and Design

OOA is concerned with devising a precise, concise, understandable, ad correct model of real-world. Before building anything complex, such as a house, a computer program, or hardware-software system., the builder must understand the requirements and the real-world environment in which will exit. The purpose of OOA is to model the real-world system so that it can be understood. To do this, we must examine requirements, analyze their implications, and restate them rigorously. We must abstract important real-world features first and defer small details until later. The successful analysis model states what must be done, without restricting how it is done, and avoids implementation decisions. The result of analysis should understand the problem as a preparation for OOD.

As shown in Figure 1.2, OOAD begins with a problem statement (requirement) generated by users and possibly customer. The requirement may be incomplete of informal, and identification processes make it more precise and exposes ambiguities and inconsistencies. The requirement should not be taken as immutable but should serve as a basis for refining the real requirements. Next, the real-world system described by the requirement must be understood and identified, and its essential features abstracted into a model. Identifying objects, attributes, associations and behaviors of the object is the important step in constructing an object model. And the next step is to organize classes by using inheritance to share common structure. Inheritance can be added in two directions **[Rumbaugh et al., 1991]**: by generalizing common aspects of existing classes into a superclass (bottom up or generalization) or by refining existing classes into specialized subclasses (top down or specialization). The object identification and refinement process are called *object model creation process*. The last OOAD step is to implement class model using a programming language.

Researchers and software designers come to a conclusion that object model creation process, including object identification and refinement is an ill-defined task, regarding of the difficulties of heuristic **[Holland et al., 1996] [Kato, 1998]** and there is no unified methodology for object-oriented software analysis and design. This is mainly due to lack of formalism for object-oriented software analysis and design. So, we can make conclusion that the challenges of object-oriented analysis and design are, to identify the objects and their attributes needed to implement the software, describe the relationships between the identified objects, define the behavior of the objects by describing the function implementations of each object, and refine objects and organize classes by using inheritance to share common structure **[Beringer, 1997] [Booch, 1991] [Holland et al., 1996] [Liang et al., 1998]**. This thesis is concerning work for solving the problems on object model creation process.

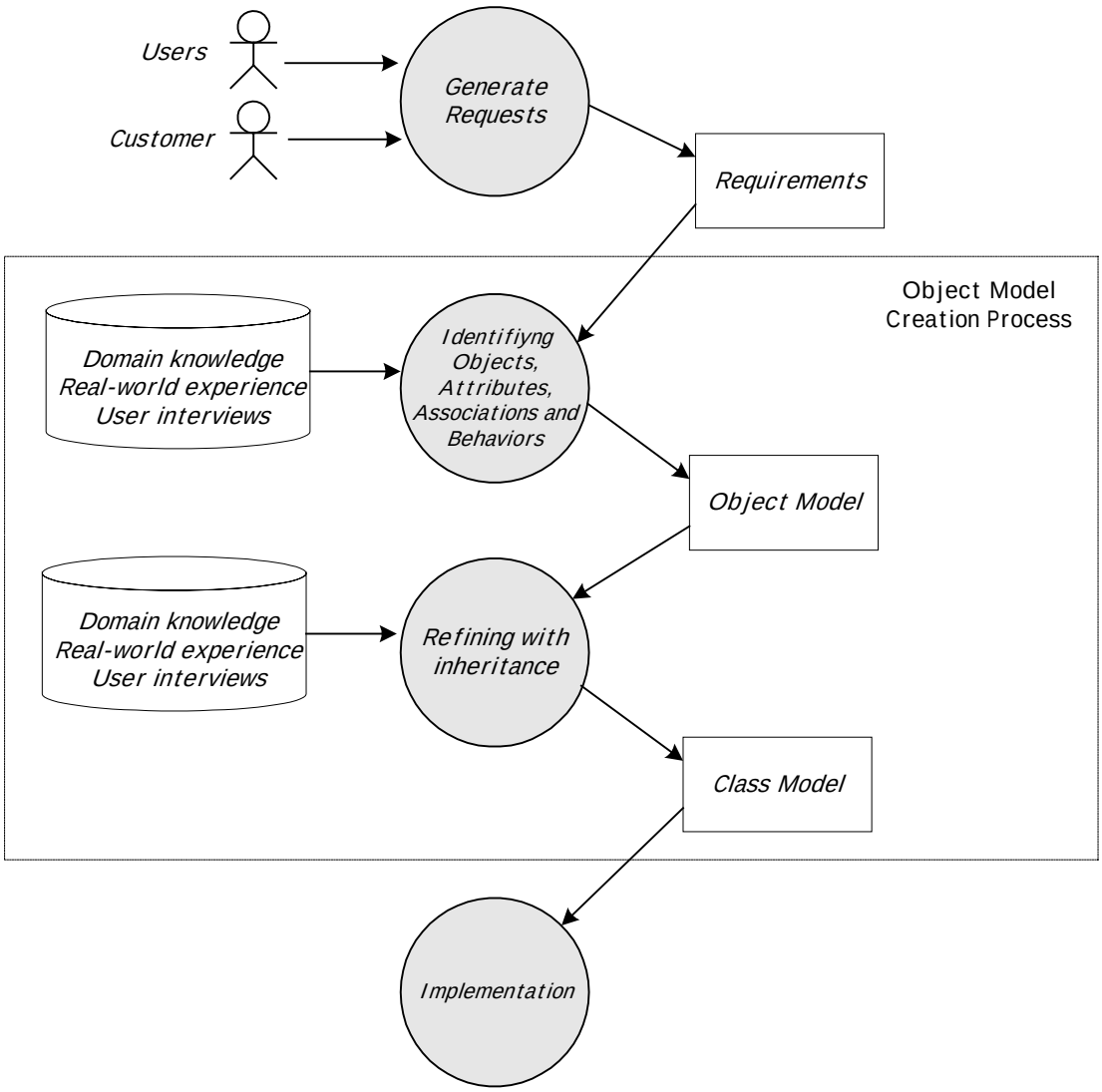

Figure 1.2: Overview of Object-Oriented Analysis and Design Process

1.2 Overview Of Intelligent Agent

1.2.1 Agent Concepts

An agent is a *physical or virtual entity*, which runs approximately as follows: **[Ferber, 1999]**

- Which is *capable of acting* in an environment,
- Which can *communicate* directly with other agents,
- Which is driven by a set of *tendencies*,

- Which *possesses resources of its own*,
- Which is *capable of perceiving* its environment,
- Which has only a *partial representation* of this environment,
- Which *possesses skills* and *can offer services*,
- Which may be able to *reproduce itself*,
- Whose *behavior* tends towards satisfying its objectives, taking account of the resources and skills available to it and depending on its perception, and the communications it receives.

A *physical entity* is something that acts in the real world; a robot, an aircraft or a car, are examples of physical entities. A software component and a computing module, on the other hand, are *virtual entities*, since they have no physical existence. Agents are capable of acting, and not just of reasoning, as in the classic AI systems. The concept of action, which is fundamental for multi-agent systems, is based on the fact that agents carry out actions, which are going to modify the agents' environment, and thus their future decision-making. They can also *communicate* with one another, and this is in fact one of the main ways in which agents interact. They are acting within an environment.

Agents are endowed with *autonomy*. This means that they are not directed by commands coming from a user or another agent, but by a set of tendencies, which can take the form of individual goals to be achieved or satisfaction or survival functions which the agent attempt to optimize. It could thus be said that motor of an agent is itself. It is the agent that is active. It can accede to or reject request coming from other agents. Autonomy is not simply behavioral, it also relates to resources. In order to act, the agent needs a certain number of resources: power, a CPU, a quantity of memory, access to certain sources of information and so on. The agent is at once partially dependent on its environment for the provision of resources, and independent of its environment to he extend that is capable of managing those resources.

Agents have only a *partial representation* of their environment, that is, they have no overall perception of what is happening. And this is actually also the case in any large-scale human endeavor in which nobody knows all the details of the project, each specialist having only a partial view corresponding to his or her area of competence.

The agent is thus a kind of organism, whose *behavior*, which can be summarized as communicating, acting, and perhaps reproducing, is aimed at satisfying its needs and

attaining its objectives, on the basis of all the other elements (perceptions, representations, actions, communications and resources) which are available to it.

Agents also must have *intelligence*. Intelligence is the degree of reasoning and learned behavior: the agent's ability to accept the user's statement of goals and carry out the task delegated to it. At a minimum, there can be some statement of preferences, perhaps in the form of rules, with an inference engine or some other reasoning mechanism to act on these preferences. Higher levels of intelligence include a user model or some other form of understanding and reasoning about what a user wants done, and planning the means to achieve this goal. Further out on the intelligence scale are systems that learn and adapt to their environment, both in terms of the user's objectives, and in terms of the resources available to the agent. Such a system might, like a human assistant, discover new relationships, connections, or concepts independently from the human user, and exploit these in anticipating and satisfying user needs.

1.2.2 Multi-Agent System Concepts

Figure 1.3 gives an illustration of the concept of a multi-agent system (MAS). The concept of multi-agent systems can be defined as applied system that comprising the following elements: **[Ferber, 1999]**

- An environment (*E*), that is, a space, which generally has a volume.
- A set of objects (*O*). These objects are situated; that is to say, it is possible at a given moment to associate any object with a position in E. These object are passive, that is, they can be perceived, created, destroyed and modified by the agents.
- An assembly of agents (*A*), which are specific objects ($A \subseteq O$), representing the active entities of the system.
- An assembly of relations (*R*), which link objects and thus agents to each other.
- An assembly of operations (*Op*), making it possible for the agents of *A* to perceive, produce, consume, transform and manipulate objects form *O*.

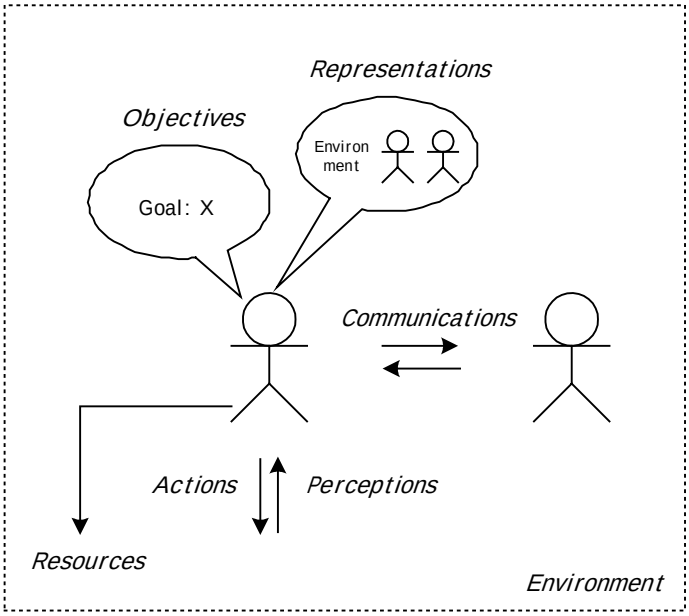

Figure 1.3: An Agent Interacting With Its Environment And The Other Agents

A special case exists for systems in which $A = O$ and E is equal to the empty assembly. In this case, the relations (R) define a network; each agent is directly linked to an assembly of other agents, which are called its *acquaintances*. These systems, which can be referred to as *purely communicating* MASs, are very common in distributed artificial intelligence. Their preferred area is cooperation between software modules, the function of which is to resolve a problem or to draw up an expert's report on the basis of specialized modules, as in the case of distributed control system where E is defined by the structure of the underlying network. These systems are characterized by the fact that interactions are essentially intentional communications and that he working mode resembles that of a social organism.

When the agent are *situated*, E is generally a metric space, and the agents are capable of perceiving their environment, that is, of recognizing the objects situated in the environment through the functioning of their perceptive capabilities; and of acting, that is, of transforming the state of the system by modifying the positions of, and the relationships existing between, the objects.

We shall see that most reactive MASs consider that the concept of environment is fundamental to the coordination of actions between several agents. For example, in a universe of robots, the agent (A) are the robots, E is the Euclidian geometrical space in

which the robots move, and *O* is obviously made up of agents, but also of physical objects placed here and there which the robots have to avoid, pick up and manipulate. The operation (*Op*) are the actions that the robots can take in moving themselves, moving the other objects or communicating, and *R* is the assembly of relations that link certain agents to others, such as acquaintance relationships (certain agents know some of the others) and communication relationship (agents can communicate with certain agents, but not necessarily with all of them).

1.2.3 Benefits of Agents

The user benefits of an agent lie in its task skills. Table 1.1 outlines the benefits of agents in broad functional categories **[Caglayan et al., 1997]**.

Feature	Advantage	Benefit
Automation	Perform repetitive task	Increased productivity
Customization	Customize information interaction	Reduced overload
Notification	Notify user of events of significance	Reduced workload
Learning	Learn users behavior	Proactive assistance
Tutoring	Coach user in context	Reduced training
Messaging	Perform task remotely	Off-line work

Table 1.1: Features, Advantages, and Benefits of Agent Technology

Automation

The automation benefits of an agent are particularly applicable for automating:

- Repetitive behavior of a single user
- Similar behavior of a group of user
- Repetitive sequential behavior of a number of users in a workflow thread

Repetitive behavior can be either time based or event based. A time-based task is something that user does at a particular time, like visiting a particular web site every morning at nine o'clock. An event-based task is something that user does in relation to another task. For example, opening your clock desk accessory before you log into an online database is an event-based task. The repetitive behavior of a single user is particularly suitable for agent-based automation when this repetitive behavior is dissimilar across the general user population. This situation makes it very hard to come up with a design compromise that would suit the whole population. On servers, similar

of a group of users can be benefit from agent automation. The personal and workgroup productivity benefits of agent automation can be significant.

Customization

An agent provides customization benefits by representing information that matches a user's personal information and interaction style preferences. The customization benefits can be discussed by noting where the agent model fits into traditional broadcast and publishing models. In the broadcast model, the service providers broadcast the same information to everyone. Users sample the information within the inherent time constraints to everyone. In such a model, an agent between a user and broadcasters can provide customization benefits by listening to the information broadcast on behalf of the user, finding relevant information matching the user's interest, and presenting the filtered information to the user. There are three basic architecture choices in the implementation of such a model. These agents can be implemented either at the broadcast site or at the user end or in the middle as a broker agent that serves multiple broadcasters and users.

Notification

An agent providing notification services to a user can produce significant reduced-workload benefits by freeing the user to monitor events of personal significance. For instance, such an agent can monitor web sites of interest for changes, and report them to a user.

Learning

An agent with learning capability can learn tasks that can be automated or preferences that can be used for customization:

- Learning and offering to automate the repetitive tasks of a single user, thus relieving the user of the need to toil with what, when, and how to automate.
- Learning the similar attributes of a group of users to customize information based on group characteristic.
- Learning similar behavior of a group of users to provide workgroup productivity enhancement.
- Learning and offering to automate recurrent sequential behavior of a group of users in a workflow thread, thus relieving the workgroup of repetitive tasks.

Tutoring

An agent with tutoring capability can coach a user in context thanks to its event monitoring and inferencing capabilities, thus reducing the training requirements.

Messaging

A messaging agent enables users to accomplish tasks off-line at remote sites. Mobile agents are examples of messaging agents that can transport themselves from place to place to interact with other agents to perform tasks on behalf of a user.

1.2.4 Areas of Agent Application

Agent technology is rapidly breaking out of universities and research labs, and is beginning to be used to solve real-world problems in range of industrial and commercial applications. Fielded applications exist today, and new systems are being developed at an increasingly rapid rate. The main areas in which agent-based applications have been reported are as follows: manufacturing, process control, telecommunication systems, air traffic control, traffic and transportation management, information filtering and gathering, electronic commerce, business process management, entertainment and medical care **[Jennings et al., 1998]**.

Industrial Applications

Industrial applications of agent technology were among the first to be developed, and today, agents are being applied in a wide range of industrial systems:

- **Manufacturing**: Parunak **[Parunak, 1987]** describes the *YAMS* system (Yet Another Manufacturing System), which applies the Contract Net Protocol to manufacturing control. The goal of *YAMS* is to efficiently manage the production process at these plants. This process is defined by some constantly changing parameters, such as products to be manufactured, available resources, time constraints, and so on. In order to achieve this enormously complex task, *YAMS* adopt multi-agent approach, where each factory and factory component is represented as an agent. Each agent has collection of plans, representing its capabilities. Other systems in this area include those for: configuration design of manufacturing product, collaborative design, scheduling and controlling manufacturing operations, controlling a manufacturing robot, and determining production sequences for a factory.

- **Process Control**: Process control is a natural application for agents, since process controllers are themselves autonomous reactive system. It is not surprising, therefore that a number of agent-based process control applications should have been developed. The best known of these is *ARCHON*, a software platform for building multi-agent systems, and associated methodology for building applications with this platform **[Jennings et al., Dec 1996]**. Other agent-based process control systems have been written for monitoring and diagnosing faults in nuclear power plants, spacecraft control, climate control, and steel coil processing.

- **Telecommunications**: Telecommunication systems are large, distributed networks of interconnected components which need to be monitored and manage in real-time. In what is a fiercely competitive market, telecommunication companies and service providers aim to distinguish themselves from their competitors by providing better, quicker or more reliable services. To achieve this differentiation, they are increasingly turning to state-of-art software techniques including agent-based approaches. In one such application, negotiating agents are used to tackle the feature interaction problem. As new features are being added to the phone network at an ever increasing rate, it is becoming correspondingly more difficult to determine which features interact with, and are inconsistent with, which other features. Therefore, the traditional approach for analyzing services at design time and hard wiring in solutions for all possible interaction permutations is doomed to failure. Given this situation, Griffeth and Velthuijsen **[Griffeth et al., 1994]** decided to adopt a different strategy and tackle the problem on an as-needed basis at run-time. They did this by employing negotiating agents to represent the different entities who are interested in the set up of call. Other problems for which agent-based systems have been constructed include: network control, transmission and switching, service management and network management.

- **Air Traffic Control**: Ljunberg and Lucas **[Ljunberg et al., 1992]** describe a sophisticated agent-realized air traffic control system known as *OASIS*. In this system, which is undergoing field trials at Sydney airport in Australia, agents are used to represent both aircraft and the various air-traffic control systems in operation. The agent metaphor thus provides a useful and natural way of modeling real-world autonomous components. As an aircraft enters Sydney airspace, an agent is allocated for it, and the agent is instantiated with the information and goals corresponding to the real-world aircraft.

Commercial Applications

While industrial applications tend to be highly-complex, bespoke systems which operate in comparatively small niche areas, commercial applications, especially those concerned with information management, tend to be oriented much more towards the mass market.

- **Distributed Project Management**: For effective collaborative working between the parties in a construction project team, it is essential that enabling information and communications technologies are available. The problems posed by the use of heterogeneous software tools are well known and need to be overcome by the adoption of new approaches. One approach, which has significant potential for use in the construction industry, involves the use of distributed artificial intelligence, which is commonly implemented in the form of intelligent agents. **[Anumba et al., 1997]** is intended to serve as a useful decision support system for designers, and should allow faster, better, and more economic, collaborative design of buildings. Other applications in this area include *RAPPID* project **[Parsons et al., 1999]**, *PROCESSLINK* project **[Petrie et al., 1999]** which has research goal to enable multidisciplinary design engineers to track and coordinate their design decisions with each other, even when not co-located or working with the same software, and *OOExpert* project **[Romi et al., March 1999] [Romi et al, June 1999]** that concern work on building intelligent agents for object model creation process in object-oriented analysis and design.

- **Information Management**: The lack of effective information management tools has given rise to what is colloquially known as the information overload problem. We can characterize the information overload problem in two ways:
  1. *Information Filtering*: Everyday we are presented with enormous amounts of information, only a tiny proportion of which is relevant or important. We need to able to sort the wheat from the chaff, and focus on the information we need.
  2. *Information Gathering*: The volume of information available prevents us from actually finding information to answer specific queries. We need to be able to obtain information that meets our requirements, even if this information can only be collected from a number of different sites.

  One important contributing factor to information overload is almost certainly that an end user is required to constantly direct the management process. But there is in principle no reason why such searches should not be carried out by agents, acting

autonomously to search the web on behalf of some user. The idea is so compelling that many projects are directed at doing exactly this **[Maes, 1994] [Sycara et al., 1996]**. Other application in this area include: a personal assistant that learns user interests and on the basis of these compiles a personal newspaper, a personal assistant agent for automating various user tasks on a computer desktop, a home page finder agent, a web browsing assistant and expert locator system.

- **Electronic Commerce**: Currently, commerce is almost entirely driven by human interactions; humans decide when to buy goods, how much they are willing to pay, and so on. But in principle, there is no reason why some commerce cannot be automated. By this, we mean that some commercial decision-making can be placed in the hands of agents. Although widespread electronic commerce is likely to lie some distance in the future, an increasing amount of trade is being undertaken by agents. As an example, **[Chaves et al., 1996]** describes a simple electronic marketplace called Kasbah. This system realizes the marketplace by creating buying and selling agents for each good to be purchased or sold respectively. Commercial transactions then take place by the interactions of these agents. Other commerce applications include: an agent which discovers the cheapest CDs, a personal shopping assistant able to search online stores for product availability and price information, a virtual marketplace for electronic commerce, and several agent-based interactive catalogues.

- **Business Process Management**: Company managers make informed decisions based on a combination of judgment and information from many departments. Ideally, all relevant information should be brought together before judgment is exercised. However obtaining pertinent, consistent and up to date information across a large company is a complex and time-consuming process. For this reason, organizations have sought to develop a number of IT systems to assist with various aspects of the management of their business process. Project *ADEPT* **[Jennings et al., 1996]** tackles this problem by viewing a business process as a community of negotiating, service providing agents. Each agent represents a distinct role or department in the enterprise and is capable of providing one or more services. Other applications in this area include a system for supply chain management, a system for managing heterogeneous workflows and a system of mobile agents for inter-organizational workflow management.

Entertainment Applications

The leisure industry is often not taken seriously by the computer science community. Leisure applications are frequently seen as somehow peripheral to the serious application of computers. And yet leisure applications such as computer games can be extremely challenging and lucrative. Agents have an obvious role in computer games, interactive theatre, and related virtual reality applications: such systems tend to be full of semi-autonomous animated characters, which can naturally be implemented as agents.

- **Games**: Grand and Cliff **[Grand et al., 1998]** built the highly successful Creatures game using agent techniques. Creatures provides a rich, simulated environment containing a number of synthetic agents that a user can interact with in real-time. The agents are intended to be sophisticated pets whose development is shaped by their experiences during their lifetime. **[Wavish et al., 1996]** also described applications of agent technology to computer games.

- **Interactive Theater and Cinema**: By interactive theatre and cinema, we mean a system that allows a user to play out a role analogous to those played by real, human actors in plays or films, interacting with artificial, computer characters that have the behavioral characteristics of real people. Agents that play the part of human in theatre-style applications are often known as believable agents--software programs "that provide the illusion of life, thus permitting an audience's suspension of disbelief". A number of projects have been set up to investigate the development of such agents **[Trapl et al., 1997] [Lester, 1997]**.

Medical Applications

Medical informatics is a major growth area in computer science: new applications are being found for computers everyday in the health industry. It is now surprising, therefore, that agents should be applied in this domain. Two of the earliest applications are in the areas of patient monitoring and health care **[Hayes et al., 1989] [Huang et al., 1995]**.

1.3 Research Motivations And Objectives

The challenges of object-oriented analysis and design are, to identify the objects and their attributes needed to implement the software, describe the associations between the

identified objects, define the behavior of the objects by describing the function implementations of each object, and refine objects and organize classes by using inheritance to share common structure **[Beringer, 1997] [Booch, 1991] [Holland et al., 1996] [Liang et al., 1998]**. The object identification and refinement process are very important process in OOAD, and we called this process by *object model creation process* (Figure 1.2). Researchers and software designers come to a conclusion that object model creation process is an ill-defined task, regarding of the difficulties of heuristic **[Holland et al., 1996] [Kato, 1998]** and there is no unified methodology for object-oriented software analysis and design. This is mainly due to lack of formalism for object-oriented software analysis and design.

In our project, we are developing an intelligent agents system that aims to help designers while designing object-oriented software by automating the difficulties and ill-defined tasks in the object model creation process, including identification of objects, relationships, attributes, behaviors and organization of objects with inheritance. We formulate design patterns and rules for solving the above problems, and store them in the knowledge bases. This system is named *OOExpert* **[Romi et al., March 1999] [Romi et al, June 1999]**.

1.4 How This Thesis Is Organized

The remainder of this thesis is organized into five chapters: introduction, object model creation process and its computational model, system architecture and design, implementation, and conclusion.

Chapter 1: Introduction

In this chapter, we give a brief introduction and overview to the two major topics covered in this thesis: object-oriented analysis and design, and intelligent agents. It starts with a short introduction of the object-oriented paradigm, as frameworks rely heavily on its mechanisms such as object, class, inheritance, polymorphism and so on. And then, we will take a look at object-oriented analysis and design, and its problem that motivate us to do research on this topic. We present the key attributes of intelligent agents such as autonomy, mobility, and intelligence, and also provide the benefits and taxonomy of various intelligent agents technology. The research motivations and objectives are also presented at the end of this chapter.

Chapter 2: Object Model Creation Process and Its Computational Model

In this chapter, we focus on object model creation process and why it has the capacity to play a key role in object-oriented analysis and design. However, building software engineering tools, and defining repository requires quantitative approach, because everything must be clear and unambiguous. One way to ensure clarity of ideas is through mathematical formalism. This chapter is an initial attempt to produce such formalism for object model creation process used to represent the result of our works. It presents a basic ontology for expressing our concepts and their relationships using set of theory and functions. In this chapter, we explain our concepts, idea and approach toward well-defined object model creation process and its computational model.

Chapter 3: System Architecture and Design

In this chapter, we focus on how the problems on object model creation process introduced and formalized in the previous section can be designed to be a software system. In our research, object model creation process is viewed as a society of software agents that interact and negotiate with each other. We also construct the *OOExpert* agent framework so that inter-agent communication can be supported as well as the mobility of our agents across network. Finally, we explain system design and architecture of each *OOExpert* agent, including requirements acquisition agent, object identification agent, attribute identification agent, association identification agent, behavior identification, and object refinement agent.

Chapter 4: Implementation

In this chapter, we focus on how the problems on object model creation process introduced, formalized, and designed in the previous section can be implemented to be a software system. It starts with an explanation about why Java is used as programming language to implement *OOExpert* agents. However, There are specific features of Java, which support intelligent agent paradigm: autonomy, intelligence and mobility. How the *OOExpert* agents work is also presented at the end of this chapter.

Chapter 5: Conclusion

At this point we have described and addressed the problem of object model creation process in object-oriented analysis and design. Furthermore, We have defined and formalized our approach to overcome above problems. We also have designed and

implemented our idea to be a software system, that we called it *OOExpert*. The final step will be to summarize the argument presented in this thesis and reflect on it.

Chapter 2
Object Model Creation Process and Its Computational Model

In this chapter, we focus on object model creation process and why it has the capacity to play a key role in object-oriented analysis and design. However, building software engineering tools, and defining repository requires quantitative approach, because everything must be clear and unambiguous. One way to ensure clarity of ideas is through mathematical formalism. This chapter is an initial attempt to produce such formalism for object model creation process used to represent the result of our works. It presents a basic ontology for expressing our concepts and their relationships using set of theory and functions. In this chapter, we explain our concepts, idea and approach toward well-defined object model creation process and its computational model.

2.1 Priming the Object Model Creation Process

As shown in Figure 2.1, OOAD begins with a problem statement (requirement) generated by users and possibly customer. The requirement may be incomplete, informal, and identification processes make it more precise and expose ambiguities and inconsistencies. The requirement should not be taken as immutable but should serve as a basis for refining the real requirements. Next, the real-world system described by the requirement must be understood and identified, and its essential features abstracted into a model. Identifying objects, attributes, associations and behaviors of the object is the important step in constructing an object model. And the next step is to organize classes by using inheritance to share common structure. Inheritance can be added in two directions **[Rumbaugh et al., 1991]**: by generalizing common aspects of existing classes into a superclass (bottom up or generalization) or by refining existing classes into specialized subclasses (top down or specialization). The object identification and refinement process are together called object model creation process.

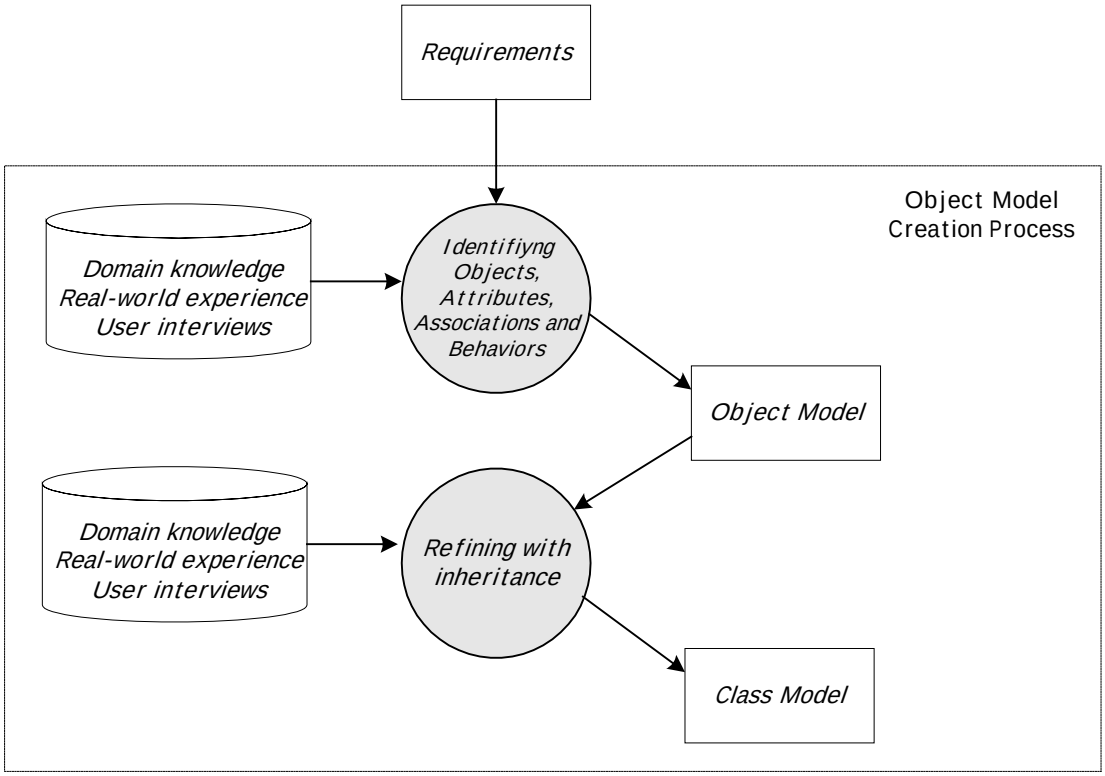

Figure 2.1: Object Model Creation Process

Once developers have established the requirements for software, they can begin the recursive object model creation process. The process can be summarized in the following steps:

1. Identifying Objects
2. Identifying Attributes
3. Identifying Associations
4. Identifying Behaviors
5. Object Refinement with Inheritance

The object model creation process starts with the specification phase. In this phase the developers work to understand exactly what the objects will look like in finished. The result of the specification phase is a description of all external views of the completed object. After the specification phase, come to the identification phase. During the identification phase the developer describes how the object works in terms of subordinate objects, including object attributes, associations and behaviors. The identification phase also identifies the subordinate objects for use in following phase. Identifying the subordinate objects sets the stage for the recursive phase of the object model creation process. The final phases of the object model creation process focus on object refinement.

The object model creation process relies on abstraction and expert perspective. Each phase in sequence, applies perspectives to produce artifacts. The full set of artifacts constitutes the complete product. Because the object model creation process needs requirements as its input, the first step in the creation of a program is to identify the program requirements. Given the program requirements, the developer begins the recursive object mode creation process using the program as the highest-level object. Decomposing the starting point operations provides insight into identifying and creating objects.

> **Definition 2.1** (*Object Model Creation Process*): Object model creation process is a main process of object-oriented analysis and design process, which starts with identification of objects, behaviors, attributes, and associations from requirements, and ends with object refinement with inheritance process.

2.2 Requirements Specification and Its Computational Model

2.2.1 Requirements Specification and Acquisition Concepts

Requirement acquisition is considered one of the most important activities in software development. Most faults found during testing and operation result from poor understanding or misinterpretation of requirements. In spite of progress in analysis techniques, CASE tools support, prototyping, and early verification and validation technique, software development still suffers from poor requirements acquisition.

In the traditional approach to software analysis, system analyst interview end users to capture requirements. We propose an approach where end users take an active role in analysis by specifying requirements using *Object Based Formal Specification* (OBFS). We use OBFS to guide end users in describing their problem. This approach will be first important step for solving the difficulties and ill-defined tasks in the object model creation process, including identification of objects, associations, attributes, behaviors and organization of objects with inheritance. This approach also takes advantage of end users' domain knowledge.

Requirements Specification

A requirement is a desired relationship among phenomena of the environment of a system, to be brought about by the software system that will be constructed and installed in the environment.

A specification describes system behavior sufficient to achieve the requirement. A specification is a restricted kind of requirement. All the environment phenomena mentioned in a specification are shared with the system. The phenomena constrained by the specification are controlled by the system, and the specified constraints can be determined without reference to the future. Specifications are derived from requirements by reasoning about the environment, using properties that hold independently of the behavior of the system **[Jackson et al., 1995]**.

In other words, we can say that the difference between requirements and specification is that requirements refer to the entire system to be realized, whereas a specification refers only to the part of the system to be implemented by software.

Jackson and Zave **[Jackson et al., 1995]** consider specifications as special kind of requirements. A requirement is a specification if all actions constrained by the requirement are controlled by the software system, and all information it relies on is shared with the software system and refers only to the past, not the future. Requirements (and thus specifications) do not talk about the state of the software system. In contrast to this view, we consider a specification to be a model of the software system to be built in order to satisfy the requirements.

The software requirements specifications process consists of three steps:

1. Requirements capture and analysis
2. Requirements definition and specification
3. Requirements validation

The origin of most software system is the need of a client /user who desires a new software system. The final output of this process is a requirements document, which defines the system to be developed **[Jalote, 1997]**.

Formal Methods and Formal Specification

Formal methods used in developing software systems provide frameworks for specifying, developing, and verifying systems in a systematic manner rather than ad hoc manner. Formal methods are used to reveal ambiguity, incompleteness, and inconsistency in a software system. System designer use formal methods to specify desired behavioral and structural properties **[Ralston et al., 1993]**. Formal methods can be used in all phases of software's development and present an opportunity to develop new techniques to improve software production. One tangible product to applying a formal method is a formal specification.

Formal requirement specifications have the additional advantage over informal requirement specifications because they are amenable to machine analysis and manipulation. The greatest benefit of applying a formal requirement specification is that system designers gain a deeper understanding of the specified system, because they have forced to be more abstract and precise about desired properties. Another important

application of formal requirement specification is that they can be used as a base to reason about the behavior of system's components.

Below, there are some applications of formal specifications and the formal methods that support each software development phase: **[Iglewski et al., 1997]**

- **Requirement Analysis**. This step clarifies in the informally stated requirements, help clear up vague ideas, reveals contradictions (or inconsistencies), ambiguities and incompleteness.
- **Software Design**. This step is used during modular decomposition and refinement to record design decisions and assumptions. It provides the implementer the information needed to construct the modules without knowledge of its clients. The implementation can be replaced by more efficient one, without affecting the interface and the client's code.
- **Software Verification**. This step is the process of showing that a system satisfies its specification. This process is impossible without a formal specification. It is important to realize that although the entire system may never be completely verified, a smaller, critical piece often can be.
- **Software Validation**. Formal methods can aid in system testing and debugging. Specifications can be used to generate complete test cases.
- **Software Documentation**. A specification serves as a description of the software. It is used for a communication between a client and a specifier, between a specifier and an implementer, and among the implementation team.
- **Software Analysis and Evaluation**. To learn from experience of building prototype software, developers should perform a critical analysis of its functionality and performance after this prototype has been built. Recently, significant research has been carried out in specifying a software, which is already built, running, and used. Some of these exercises revealed serious bugs in published algorithms and design. As expected, most formal specifications revealed unstated assumptions, inconsistencies, and unintentional incompleteness of software.

The usefulness of formal requirement specification is more and more accepted by researcher and practical software engineers. But formal requirement specification techniques still suffer from two drawbacks.

First, research spends more effort to develop new languages than provide

methodological guidance for using existing ones. Often, users of formal techniques are left alone with a formalism for which no explicit methodology has been developed.

Second, formal requirement specification techniques are not well integrated with the analysis phase of software engineering. Often, formal requirement specifications begin with very short description of the system to be implemented, and detail is added during the development of the formal requirement specification. Such a procedure does not adequately take into account the need to thoroughly analyze the system to be implemented and the environment in which it will operate before a detailed requirement specification is developed.

2.2.2 Models for Requirements Specification

Figure 2.2 shows our strategy to formulate requirement specification for solving the object model creation process. We propose an approach where end users take an active role in analysis by specifying requirements using *Object-Based Formal Specification* (*OBFS*). We use OBFS to guide end users in describing their problem. This approach will be the first important step for solving the difficulties and ill-defined tasks in the object model creation process, including identification of objects, associations, attributes, behaviors and organization of objects with inheritance.

OBFS is composed of *Description Statements* (*DS*), *Collaborative Statements* (*CS*), *Attributive Statements* (*AS*), *Behavioral Statements* (*BS*), and *Inheritance Statements* (*IS*).

$$OBFS = DS \oplus CS \oplus AS \oplus BS \oplus IS$$

Each OBFS statement consists of Subject (*S*), Verb (*V*), and Object (*O*) as well as the English (*E*) natural language.

$$DS = \{requirementID, requirementName, Language, Description\}$$
$$CS = \{(S_1, V_1, O_1)_{cs}, (S_2, V_2, O_2)_{cs}, (S_3, V_3, O_3)_{cs}, ...\}$$
$$AS = \{(S_1, V_1, O_1)_{as}, (S_2, V_2, O_2)_{as}, (S_3, V_3, O_3)_{as}, ...\} \quad \text{and} \quad \forall OBFS \in E$$
$$BS = \{(S_1, V_1, O_1)_{bs}, (S_2, V_2, O_2)_{bs}, (S_3, V_3, O_3)_{bs}, ...\}$$
$$IS = \{(S_1, V_1, O_1)_{is}, (S_2, V_2, O_2)_{is}, (S_3, V_3, O_3)_{is}, ...\}$$

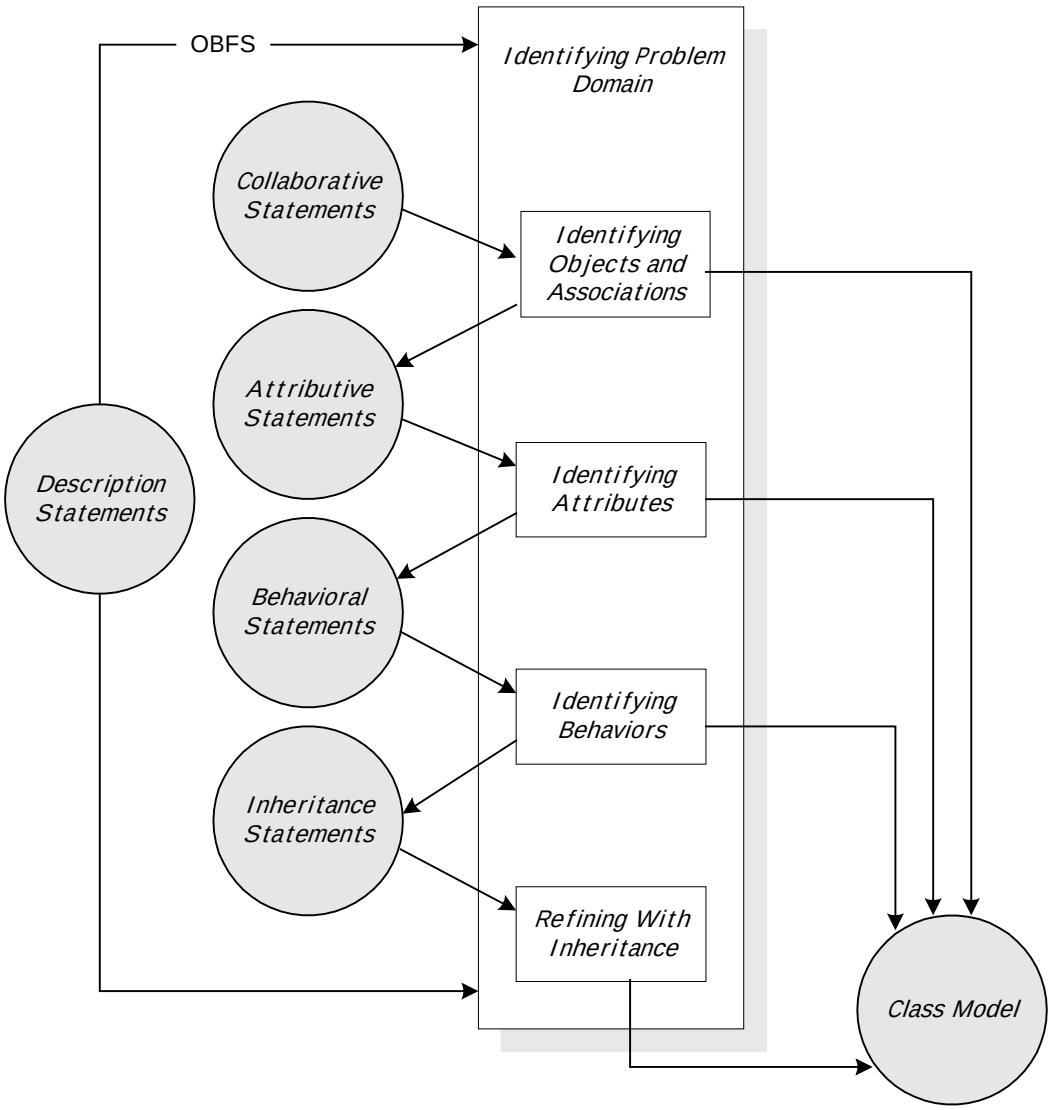

Figure 2.2: Object-Based Formal Specification

Definition 2.2 (*Object-Based Formal Specification (OBFS)*): Object-Based Formal Specification (*OBFS*) is a semi-formal requirements template used to reveal ambiguity, incompleteness, and inconsistency in an object-oriented software system, and to guide end users take an active role while describing their problem statements. OBFS is composed of description statements (*DS*), collaborative statements (*CS*), attributive statements (*AS*), behavioral statements (*BS*), and inheritance statements (*IS*).

Description Statements (DS)

Description statements are used to guide for writing an overview of the system that we want to build. Description statements contain four kinds of elements: *Requirement ID*, *Requirement Name*, *Language*, and *Description* (Figure 2.3). The description statements should state what is to be done and not how it is to be done. It should be a statement of needs, not a proposal for a solution.

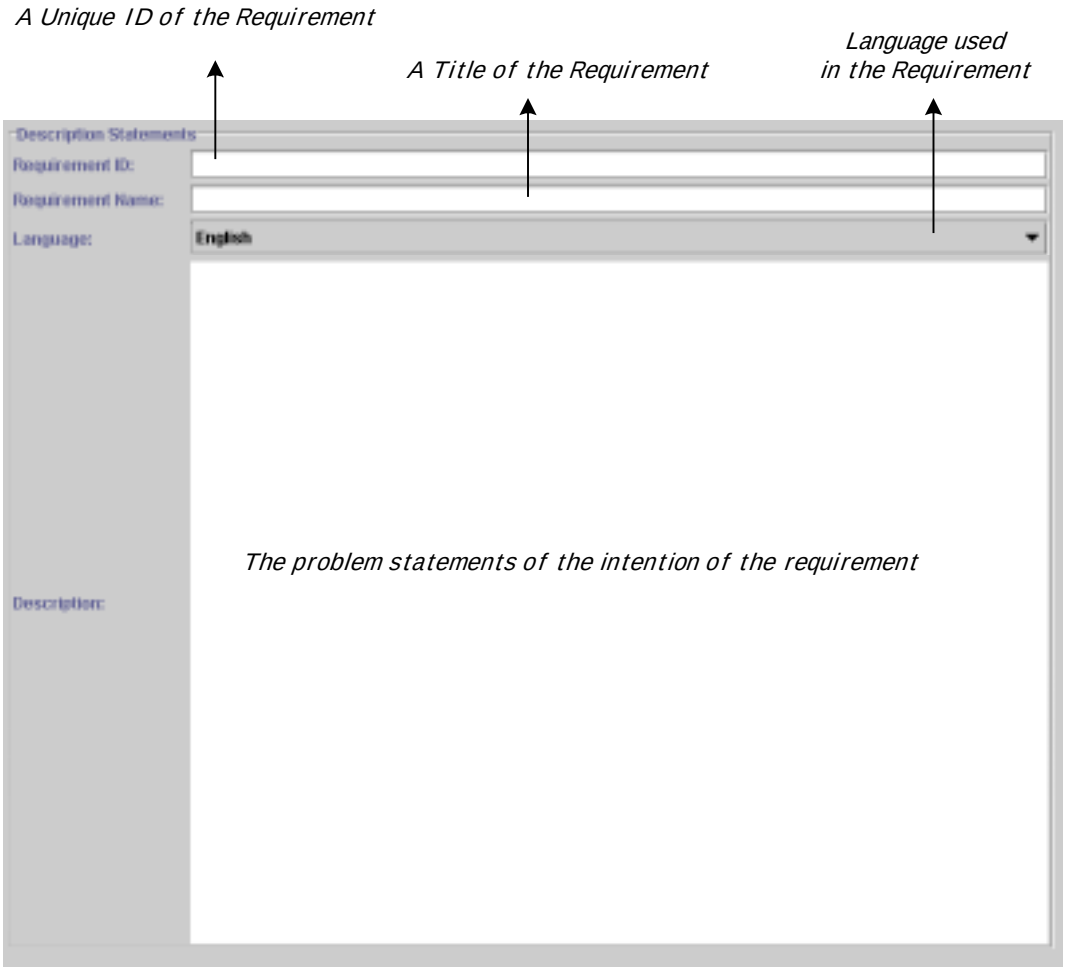

Figure 2.3: Description Statements Shell

Definition 2.3 *(Description Statements (DS))*: A description statement is a requirement statement used to write an overview of the system that we want to build, which consists of *Requirement ID*, *Requirement Name*, *Language*, and *Description*.

Collaborative Statements (CS)

Collaborative statements are used to identify objects, and association between objects. The first step in object model creation process is to identify relevant object and its association from the application domain. Objects include physical entities and all objects must make sense in the application domain. All objects are explicit in the collaborative statements, and objects are corresponding to nouns that identified from collaborative statements.

Any dependency between two ore more objects in the collaborative statements is an object association. A reference from one object to another is also an association. Associations show dependencies between objects at the same level of abstraction as the objects themselves. Associations can be implemented in various ways, but such implementation decisions should kept out of the analysis model to preserve design freedom. Associations often correspond to verbs or verb phrases. These include physical location (next to, part of, contained in), directed actions (drives), communication (talks to), ownership (has, part of), or satisfaction of some condition (works for, married to, manages).

Collaborative Statement (CS) consists of Subject (*S*), Verb (*V*), and Object (*O*) as well as the English (*E*) natural language.

$$CS = \{(S_1, V_1, O_1)_{cs}, (S_2, V_2, O_2)_{cs}, (S_3, V_3, O_3)_{cs}, ...\} \quad \text{and} \quad \forall CS \in E$$

S_{cs} and O_{cs} will be identified as a tentative object (OBJ_t), and V_{cs} will be identified as a tentative association (ASS_t) in terms of object-oriented paradigm.

$$\forall CS \in E \; [S_{cs} \Rightarrow OBJ_t] \quad \text{and} \quad \forall CS \in E \; [O_{cs} \Rightarrow OBJ_t]$$

$$\forall CS \in E \; [V_{cs} \Rightarrow ASS_t]$$

Definition 2.4 *(Collaborative Statements (CS))*: A collaborative statement is a requirement statement used to identify objects, and association between objects, which consists of subject (*S*), verb (*V*), and object (*O*) as well as the English (*E*) natural language.

Attributive Statements (AS)

Attributive statements are used to identify object attributes. Attributes are properties of individual objects. Attributes usually correspond to nouns followed by possessive phrases, and sometimes are characterized by adjectives or adverbs. Attributive statement must contain properties of each object identified at the previous step.

Attributive Statement (AS) consists of Subject (*S*), Verb (*V*), and Object (*O*) as well as the English (*E*) natural language.

$$AS = \{(S_1, V_1, O_1)_{as}, (S_2, V_2, O_2)_{as}, (S_3, V_3, O_3)_{as}, ...\} \quad \text{and} \quad \forall AS \in E$$

O_{as} will be identified as a tentative attribute (ATT_t) in the term of object-oriented paradigm. S_{as} is identified and refined objects (OBJ) from tentative object (OBJ_t) as the final result of object identification's process.

$$\forall AS \in E \; [O_{as} \Rightarrow ATT_t]$$

$$\forall AS \in E \; [S_{as} = OBJ]$$

> **Definition 2.5** *(Attributive Statements (AS))*: An Attributive statement is a requirement statement used to identify object attributes, which consists of subject (*S*), verb (*V*), and object (*O*) as well as the English (*E*) natural language.

Behavioral Statements (BS)

Behavioral statements are used to identify object behaviors. Behavior is how an object acts and reacts, in terms of its state changes and message passing **[Booch, 1991]**. Behavioral statement must contain behaviors of each object identified at the previous step.

Behavioral Statement (BS) consists of Subject (*S*), Verb (*V*), and Object (*O*) as well as the English (*E*) natural language.

$$BS = \{(S_1, V_1, O_1)_{bs}, (S_2, V_2, O_2)_{bs}, (S_3, V_3, O_3)_{bs}, ...\} \quad \text{and} \quad \forall BS \in E$$

O_{as} will be identified as a tentative behavior (BEH_t) in the term of object-oriented paradigm. S_{bs} is identified and refined objects (OBJ) from tentative object (OBJ_t) as the final result of object identification's process.

$$\forall BS \in E\ [O_{bs} \Rightarrow BEH_t]$$

$$\forall BS \in E\ [S_{as} = OBJ]$$

Definition 2.6 *(Behavioral Statements (BS))*: A Behavioral statement is a requirement statement used to identify object behaviors, which consists of subject (S), verb (V), and object (O) as well as the English (E) natural language.

Inheritance Statements (IS)

Inheritance statements are used to organize classes by using inheritance, to share common object attributes and behaviors. Inheritance provides a natural classification for kinds of objects and allows for the commonality of objects to be explicitly taken advantage of in modeling and constructing object systems. Inheritance is a relationship between classes where one class is the parent class of another. Inheritance statement provides sentences that have is-a-kind-of relationship. For example, mountain bikes, racing bikes, and tandems are all different kinds of (is-a-kind-of) bicycles.

Inheritance Statement (IS) consists of Subject (S), Verb (V), and Object (O) as well as the English (E) natural language.

$$IS = \{(S_1, V_1, O_1)_{is}, (S_2, V_2, O_2)_{is}, (S_3, V_3, O_3)_{is}, ...\} \quad \text{and} \quad \forall IS \in E$$

O_{is} will be identified as a tentative superclass (SCL_t) in the term of object-oriented paradigm. S_{is} is identified and refined objects (OBJ) from tentative object (OBJ_t) as the final result of object identification's process.

$$\forall IS \in E\ [O_{is} \Rightarrow SCL_t]$$

$$\forall IS \in E\ [S_{is} = OBJ]$$

Definition 2.7 *(Inheritance Statements (IS))*: An Inheritance statement is a requirement statement used to organize classes by using inheritance, and to share common object attributes and behaviors, which consists of subject (*S*), verb (*V*), and object (*O*) as well as the English (*E*) natural language.

2.3 Object Identification and Its Computational Model

2.3.1 Object Identification Concepts

As shown in Figure 2.4, object identification begins by listing candidate objects found in the written requirements specification. The next step is to identify relevant objects from the application domain. Objects include physical entities and all objects must make sense in the application domain. All objects are explicit in the requirements specification, and objects are corresponding to nouns that identified from requirements specification. The next step is discard unnecessary and spurious objects according to the following criteria: redundant objects, irrelevant objects, vague objects, attributes, operations, roles, and implementation construct.

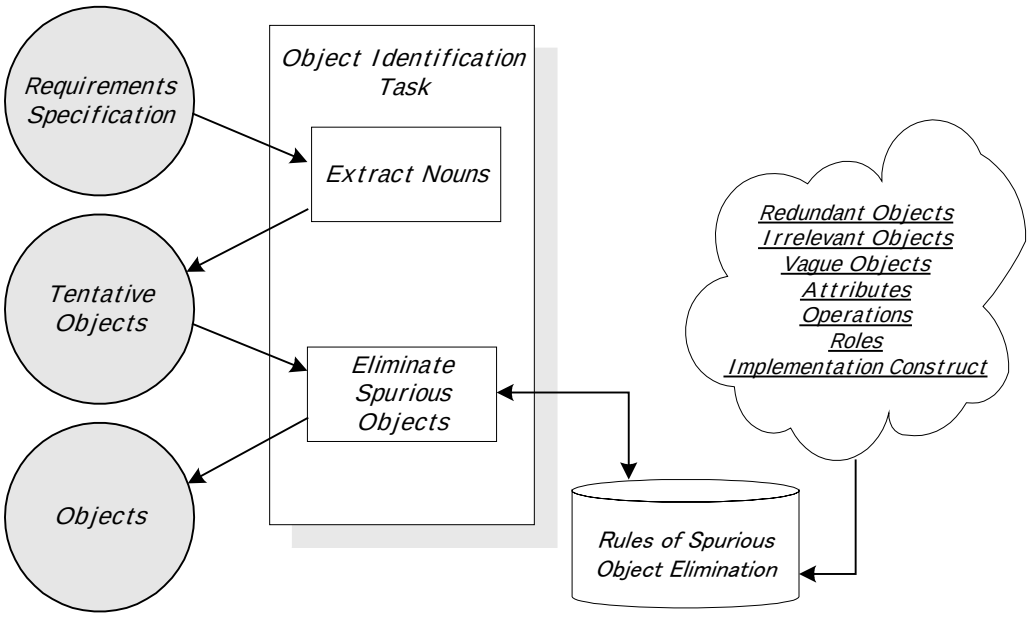

Figure 2.4: Object Identification Process

2.3.2 Models for Object Identification

Figure 2.5 shows our strategy for solving the object identification process. We use *collaborative statements* from OBFS to guide end users in describing their problem. The first step in object identification process is to extract *S* and *O* written in the collaborative statements to be *tentative objects* (OBJ_t).

$$\forall CS \in E\ [S_{cs} \Rightarrow OBJ_t] \quad \text{and} \quad \forall CS \in E\ [O_{cs} \Rightarrow OBJ_t]$$

The next step is to eliminate spurious objects and propose relevant objects using Rule-Based Reasoning (RBR) and Case-Based Reasoning (CBR) paradigms. In the RBR, the system will discard unnecessary and incorrect objects according to the following criteria: *redundant objects* (OBJ_{red}), *attributes* (OBJ_{att}), *behaviors* (OBJ_{beh}), and *not noun objects* (OBJ_{non}).

$$\forall OBJ \in E\ [\neg OBJ_{red} \wedge \neg OBJ_{att} \wedge \neg OBJ_{beh} \wedge \neg OBJ_{non} \Rightarrow OBJ]$$

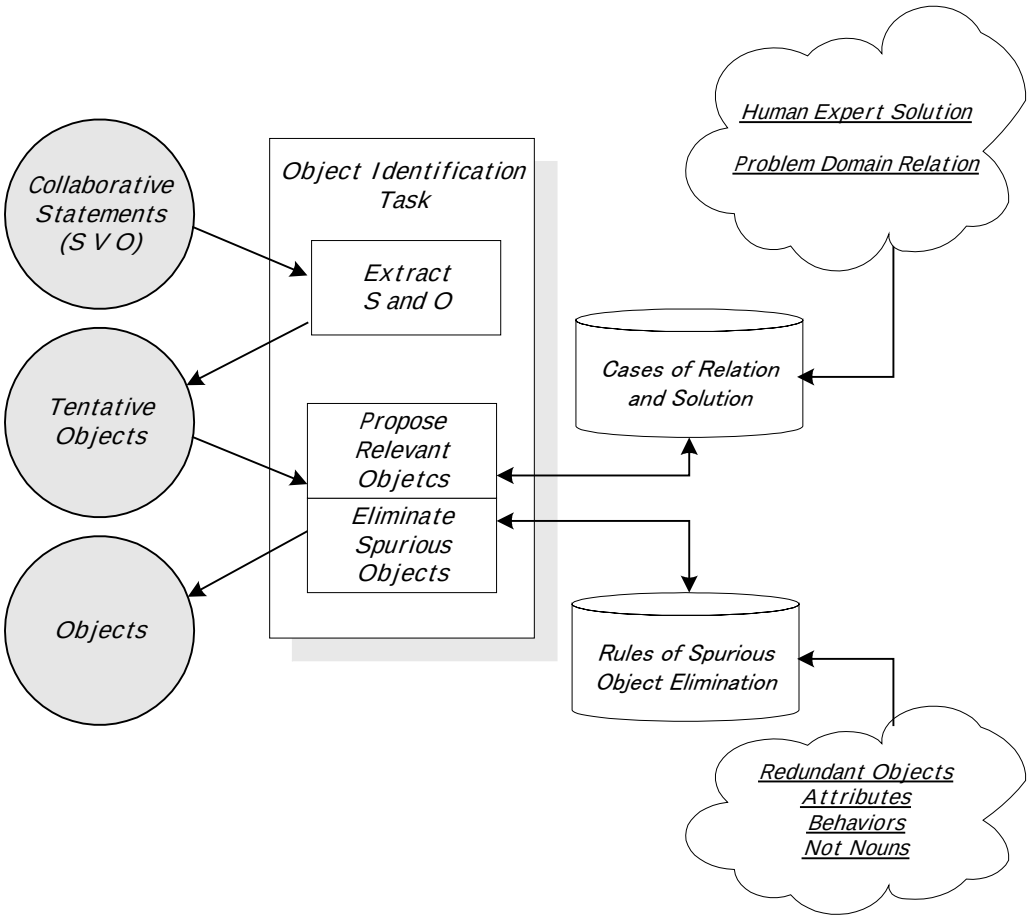

Figure 2.5: Proposed Approach for Object Identification Process

In other hand, CBR is based on psychological theories of human cognition. We collect design rules from human experts, and store/index them in the case-base. It rests on the intuition that human expertise does not depend on rules or other formalized structures, but on experiences. Human experts differ from novices in their ability to relate problems to previous ones, to reason based on analogies between current and old problems, to use solutions from old experiences, and to recognize and avoid old errors and failures. Two kinds of case-base indexed in our approach are: *Human Expert Solution* (*HES*) and *Problem Domain Relation* (*PDR*). The final result of the object identification process is a relevant object (*OBJ*).

2.4 Attribute Identification and Its Computational Model

2.4.1 Attribute Identification Concepts

As shown in Figure 2.6, attribute identification begins by listing candidate attributes found in the written requirements specification. The next step is to identify relevant attributes from the application domain. Attributes are properties of individual objects, such as name, weight, velocity, or color. And, attributes are corresponding to nouns followed by possessive phrases that identified from requirements specification. Adjectives often represent specific enumerated attribute value, such as red, on, or expired. Unlike objects and associations, attributes are less likely to be fully described in the requirements statement. We must draw on our knowledge of the application domain and the real world to find them. Fortunately, attributes seldom affect the basic structure of the problem. The next step is discard unnecessary and spurious attributes according to the following criteria: objects, qualifiers, names, identifiers, link attributes, internal values, and discordant attributes.

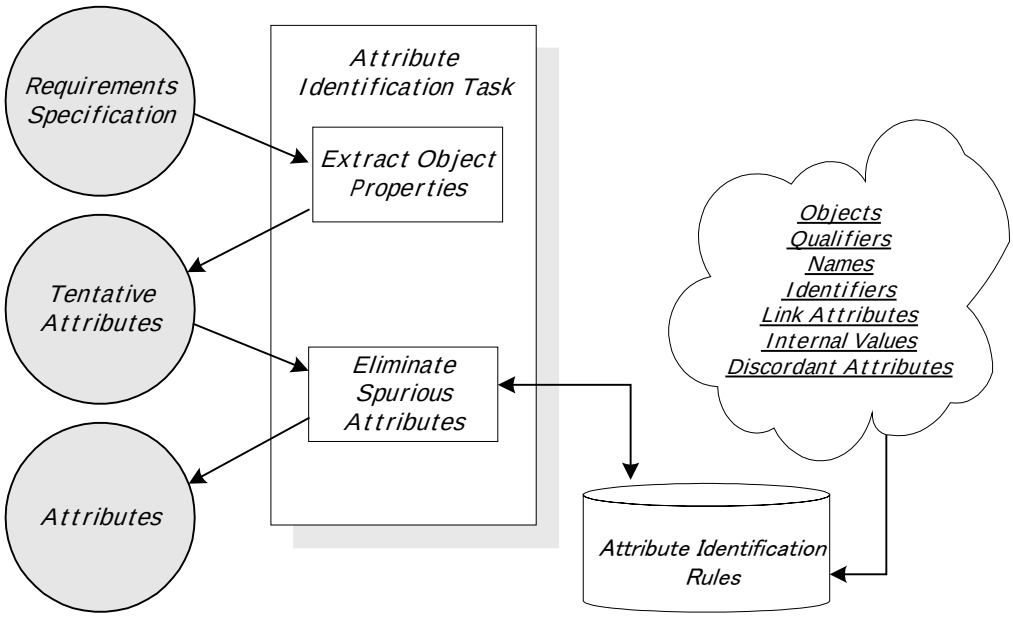

Figure 2.6: Attribute Identification Process

2.4.2 Models for Attribute Identification

Figure 2.7 shows our strategy for solving the attribute identification process. We use *attributive statements* from OBFS to guide end users in describing their problem. The first step in attribute identification process is to extract *O* written in *the attributive statements* to be tentative attribute (ATT_t).

$$\forall AS \in E\ [O_{as} \Rightarrow ATT_t]$$

The next step is to eliminate spurious attributes and propose relevant attributes using Rule-Based Reasoning (RBR) and Case-Based Reasoning (CBR) paradigms. In the RBR, the system will discard unnecessary and incorrect attributes according to the following criteria: *redundant attributes* (ATT_{red}), *objects* (ATT_{obj}), and *behaviors* (ATT_{beh}).

$$\forall ATT \in E\ [\neg ATT_{red} \wedge \neg ATT_{obj} \wedge ATT_{beh} \neg \Rightarrow ATT]$$

In other hand, CBR is based on psychological theories of human cognition. We collect design rules from human experts, and store/index them in the case-base. It rests on the intuition that human expertise does not depend on rules or other formalized structures, but on experiences. Human experts differ from novices in their ability to relate problems to previous ones, to reason based on analogies between current and old problems, to use solutions from old experiences, and to recognize and avoid old errors and failures. Two kinds of case-base indexed in our approach are: *Human Expert Solution* (*HES*) and *Problem Domain Relation* (*PDR*). The final result of the attribute identification process is a relevant attribute (*ATT*).

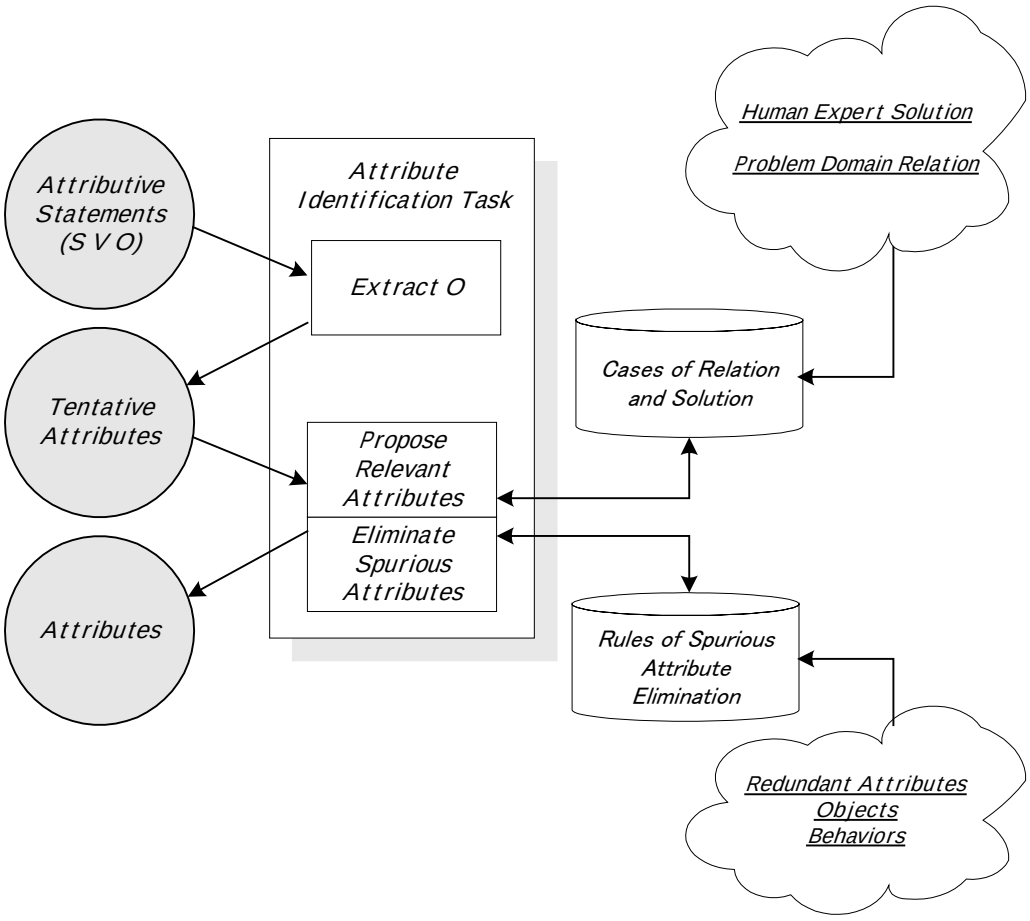

Figure 2.7: Proposed Approach for Attribute Identification Process

2.5 Association Identification and Its Computational Model

2.5.1 Association Identification Concepts

As shown in Figure 2.8, association identification begins by listing candidate associations found in the written requirements specification. The next step is to identify relevant associations from the application domain. All associations are explicit in the requirements specification, and associations are corresponding to static verbs or verb phrases that identified from requirements specification. The next step is discard unnecessary and spurious associations according to the following criteria: associations

on eliminated objects, irrelevant associations, actions, ternary associations, derived associations, misnamed associations, role names, qualified associations, multiplicity, and missing associations.

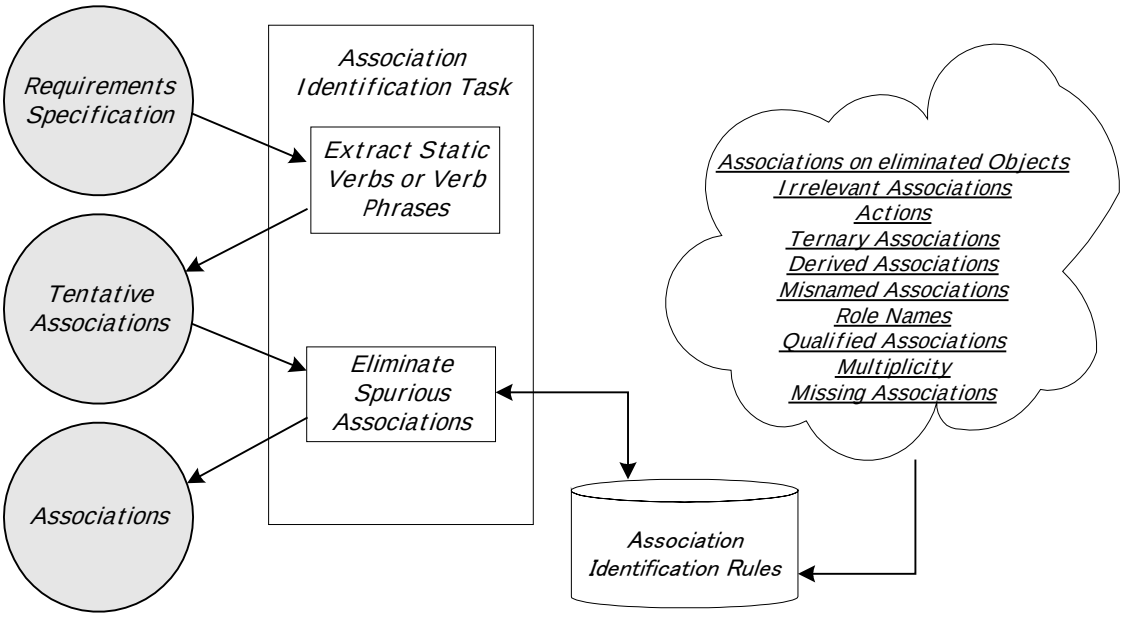

Figure 2.8: Association Identification Process

Any dependency between two ore more objects is an object association. A reference from one object to another is also an association. Associations show dependencies between objects at the same level of abstraction as the objects themselves. Associations can be implemented in various ways, but such implementation decisions should kept out of the analysis model to preserve design freedom. Associations often correspond to *static verbs* or *verb phrases*. These include physical location (next to, part of, contained in), directed actions (drives), communication (talks to), ownership (has, part of), or satisfaction of some condition (works for, married to, manages).

We can summarize, there are four general categories of inter-object associations. These are:

1. The *Is-a-kind-of* association
2. The *Uses* association
3. The *Consists-of* association
4. The *Contains* association

The classic *Is-a-kind-of* object association indicates set membership. For example, a

queen *is a kind of* chess piece, or a dog *is a kind of* animal. The object-oriented paradigm represents *Is-a-kind-of* association through inheritance. Inheritance implies that descendant object receive the attributes of their base classes.

The *Uses* association is a client-server association. In a *Uses* association, one object *uses* another object to accomplish some task. In a database application, a database object might store and retrieve data records for an application object. A client is the application that *uses* the server or the database object. Objects can be both clients and servers. While the database object is the server for the application, the database object might be a client of a file system server. It is important to focus on identifying the associations between objects and not just the objects.

The *Consists-of* association occurs when an object is composed of other objects. A car *consists of* an engine, four wheels, a body, and an electrical system. Consists-of associations are generally static. This means that a car always has four wheels and not sometimes two or 10.

Finally, the *Contains* association describes a potentially transient association such as cards in a poker hand or items in a box. The transient ness of the *Contains* association distinguishes it from the *Consists-of* association. For example, objects may be put into or taken out of a box, but in normal use, the engine never leaves the car. The *Contains* association also represents set of membership, but the transient nature of the *Contains* association distinguishes it from the *Is-a-kind-of* association. An item may come and go as a member of the set of things a box *contains*, but a dog will always be a member of the set of animals. The transient nature of the associations between the objects should help categorize the association.

To apply abstraction in the recursive analysis and design process, software designers must first concentrate on *Consists-of* association, i.e., on viewing programs as objects within objects. *Consists-of* associations are the binding links between levels of abstraction. A book consists of a table of contents, chapters, appendices, and index. The abstraction called the book is, bound by the *Consists-of* association to the lower level abstraction's table of contents, chapters, appendices, and index. For the recursive analysis and design process to be successful, we must also recognize *Contains* associations as link between levels of abstraction. Objects are potentially dynamic sets that decompose differently depending upon the changing constitution of the set. For

example, the set of windows that makes up a screen of a Graphical User Interface (GUI) application may change as the application runs. The GUI screen might be considered as abstraction of the set of windows it contains. This contains association binds the higher-level screen abstraction to the lower level windows' abstractions. Once software designers understand what the subordinate objects are, it becomes important to organize how the subordinate objects interact. As subordinate objects work to perform the functions of their superior object, they will require each other's assistance. These assisting associations are the *Uses* associations.

2.5.2 Models for Association Identification

Figure 2.9 shows our strategy for solving the association identification process. We use *collaborative statements* from OBFS to guide end users in describing their problem. The first step in association identification process is to extract *V* written in the collaborative statements to be *tentative associations* (ASS_t).

$$\forall CS \in E\ [V_{cs} \Rightarrow ASS_t]$$

The next step is to eliminate spurious associations and propose relevant associations using Rule-Based Reasoning (RBR) and Case-Based Reasoning (CBR) paradigms. In the RBR, the system will discard unnecessary and incorrect associations according to the following criteria: *redundant associations* (ASS_{red}), *attributes* (ASS_{att}), *behaviors* (ASS_{beh}), and *not verb associations* (ASS_{nov}).

$$\forall ASS \in E\ [\neg ASS_{red} \wedge \neg ASS_{att} \wedge \neg ASS_{beh} \wedge \neg ASS_{nov} \Rightarrow ASS]$$

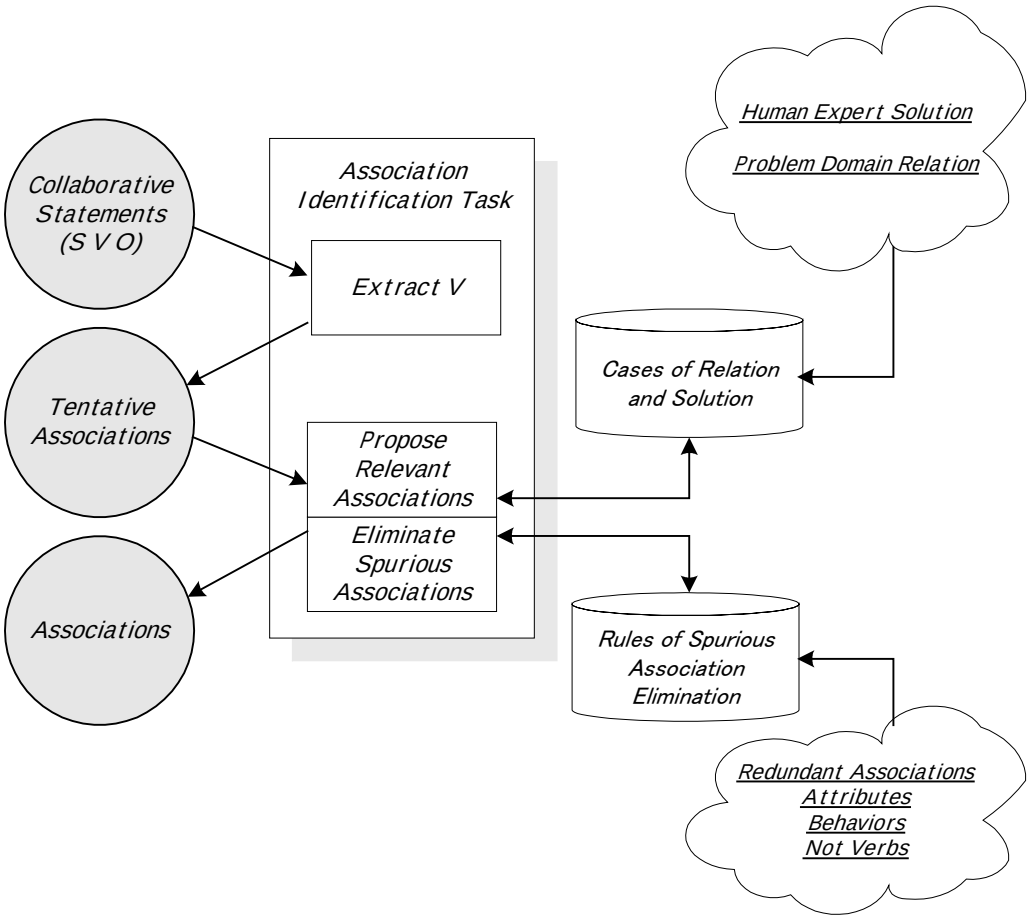

Figure 2.9: Proposed Approach for Association Identification Process

In other hand, CBR is based on psychological theories of human cognition. We collect design rules from human experts, and store/index them in the case-base. It rests on the intuition that human expertise does not depend on rules or other formalized structures, but on experiences. Human experts differ from novices in their ability to relate problems to previous ones, to reason based on analogies between current and old problems, to use solutions from old experiences, and to recognize and avoid old errors and failures. Two kinds of case-base indexed in our approach are: *Human Expert Solution* (*HES*) and *Problem Domain Relation* (*PDR*). The final result of the association identification process is a relevant association (*ASS*).

2.6 Behavior Identification and Its Computational Model

2.6.1 Behavior Identification Concepts

Behavior is how an object acts and reacts, in terms of its state changes and message passing **[Booch, 1991]**. So, behavior can most effectively be identified by explicitly stating what the object does. For example, which of these descriptions best exemplifies a word processor?

- A program that runs on a PC, drives a laser printer, employs columns, and users four megabytes of disk space.
- A program that lets users edit, format, and print text.

The first description identifies features of word processors but does not describe what word processors do. The first description could also identify a spreadsheet application. In contrast, the second description focuses on the behaviors unique to word processors. Due to its focus on behavioral, the second description eliminates many unrelated classes of applications. However, recently there is no methodology to extract behavior from requirements specification. But we can summarize from above discussion, as shown in Figure 2.10, behavior identification is a process to extract what the object does (object acts and reacts) from requirements specification.

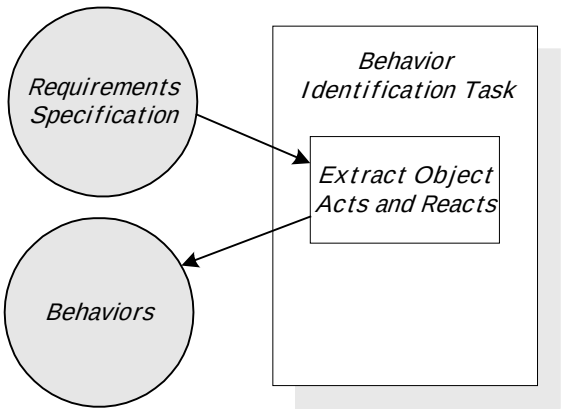

Figure 2.10: Behavior Identification Process

2.6.2 Models for Behavior Identification

Figure 2.11 shows our strategy for solving the behavior identification process. We use *behavioral statements* from OBFS to guide end users in describing their problem. The first step in behavior identification process is to extract *O* written in *the behavioral statements* to be *tentative behavior* (BEH_t).

$$\forall BS \in E\ [O_{bs} \Rightarrow BEH_t]$$

The next step is to eliminate spurious behaviors and propose relevant behaviors using Rule-Based Reasoning (RBR) and Case-Based Reasoning (CBR) paradigms. In the RBR, the system will discard unnecessary and incorrect behaviors according to the following criteria: *redundant behaviors* (BEH_{red}), *objects* (BEH_{obj}), *associations* (BEH_{ass}), *attributes* (BEH_{att}), *and not verb behaviors* (BEH_{nov}).

$$\forall BEH \in E\ [\neg BEH_{red} \wedge \neg BEH_{obj} \wedge \neg BEH_{ass} \wedge \neg BEH_{att} \wedge \neg BEH_{nov} \Rightarrow BEH]$$

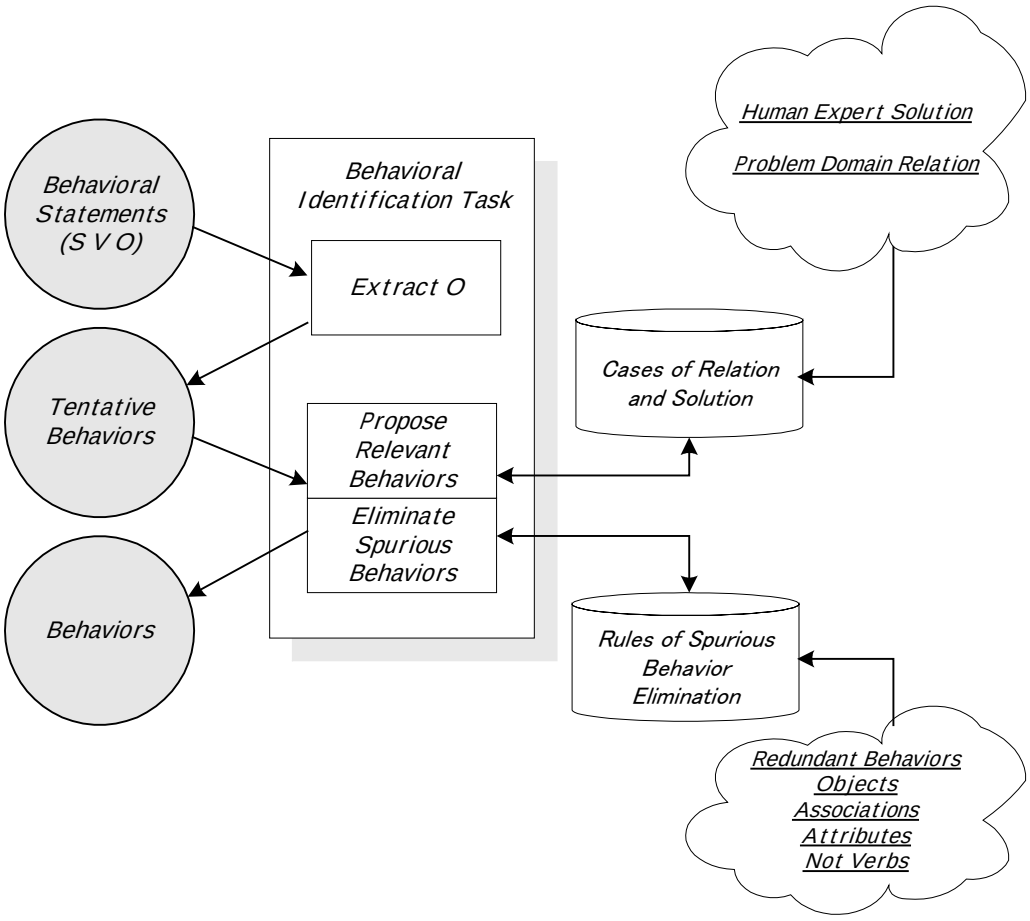

Figure 2.11: Proposed Approach for Behavior Identification Process

In other hand, CBR is based on psychological theories of human cognition. We collect design rules from human experts, and store/index them in the case-base. It rests on the intuition that human expertise does not depend on rules or other formalized structures, but on experiences. Human experts differ from novices in their ability to relate problems to previous ones, to reason based on analogies between current and old problems, to use solutions from old experiences, and to recognize and avoid old errors and failures. Two kinds of case-base indexed in our approach are: *Human Expert Solution* (*HES*) and *Problem Domain Relation* (*PDR*). The final result of the behavior identification process is a relevant behavior (*BEH*).

2.7 Object Refinement with Inheritance and Its Computational Model

2.7.1 Object Refinement with Inheritance Concepts

The final step of object model creation process is to organize classes by using inheritance to share common structure. Inheritance can be added in two directions, *bottom up* (generalization) and *top down* (specialization).

1. *Bottom Up* (Generalization): By generalizing common aspects of existing classes into a superclass. We can discover inheritance from the bottom up by searching for classes with similar attributes, associations, or behaviors. For each generalization, define a superclass to share common features.
2. *Top Down* (Specialization): By refining existing classes into specialized subclass. Top down specializations are often apparent from the application domain. Look for noun phrases composed of various adjectives on the class name. Avoid excessive refinement. If proposed specializations are incompatible with an existing class, the existing class may be improperly formulated. Enumerated subcases in the application domain are the most frequent source of specializations. Often, it is sufficient to note that a set of enumerated subcases exists, without actually listing them.

The object-oriented paradigm represents *Is-a-kind-of* association through inheritance. Inheritance implies that descendant object receive the attributes of their base classes. For example, a queen *is a kind of* chess piece, or a dog *is a kind of* animal. The *Is-a-kind-of* association is a generalization that encompasses several objects with similar methods.

> **Definition 2.8** *(Generalization and Specialization)*: Generalization and Specialization are relationships between concepts. Any type of A, each of whose objects is also an instance of a given type B, is called a specialization (or subtype) of B and is written as $A \subset B$. B is also called the generalization (or supertype) of A.

As with the classification relation the specialization relation can also be qualified in

extensional and intensional terms **[Odell et al., 1997]**. For example,

$$ext(Dog) \subset ext(Animal) \quad \text{or simply} \quad Dog \subset Animal$$

means that every number of the Dog is also a member of the Animal set. In contrast,

$$\text{int}(Dog) \subset \text{int}\,t(Animal)$$

means the definition of Dog must contain the definition of Animal. When viewed in extension, the left side of the $\subset$ involves fewer than the right, because the left side is a subset. When viewed in intension, the left side of the $\subset$ involves more than the right, because the definition of the left must also include the definition of the right. In short, when going down a generalization hierarchy, the extension gets smaller while the intension get bigger.

2.7.2 Models for Object Refinement with Inheritance

Models for Object Refinement with Inheritance Using Specialization

First, we denote the set of behaviors of a class *CLS* by CLS^{beh}. And we also denote the set of attributes of a class *CLS* by CLS^{att}. So in this case:

$$CLS^{beh} = \{BEH_1, BEH_2, ..., BEH_k\} \quad \text{and} \quad CLS^{att} = \{ATT_1, ATT_2, ..., ATT_k\}$$

As we know, a class is composed of behaviors and attributes. In this case, class *CLS* is composed of behaviors CLS^{beh} and attributes CLS^{att}.

$$CLS = CLS^{beh} \oplus CLS^{att}$$

We assume that the members of class *CLS* are CLS_1, CLS_2, …, CLS_k and the members of class *SCL* are SCL_1, SCL_2, …, SCL_k.

$$CLS = \{CLS_1, CLS_2, ..., CLS_k\} \quad \text{and} \quad SCL = \{SCL_1, SCL_2, ..., SCL_k\}$$

Class *SCL* is specializing CLS_1, CLS_2, …, CLS_k. This means that class *SCL* has CLS_1,

CLS_2, …, CLS_k as subclasses and define attributes and behaviors inherited from *SCL* as superclass. In this case, we can denote that class *SCL* will therefore be the union of all the class *CLS*. Furthermore, class *SCL* conforms to all its subclasses in the class hierarchy. This is expressed formally bellow.

$$SCL = \bigcup_{i=1}^{k} CLS_i$$

Models for Object Refinement with Inheritance Using Generalization

First, we denote the set of behaviors of a class *CLS* by CLS^{beh}. And we also denote the set of attributes of a class *CLS* by CLS^{att}. So in this case:

$$CLS^{beh} = \{BEH_1, BEH_2, ..., BEH_k\} \quad \text{and} \quad CLS^{att} = \{ATT_1, ATT_2, ..., ATT_k\}$$

As we know, a class is composed of behaviors and attributes. In this case, class *CLS* is composed of behaviors CLS^{beh} and attributes CLS^{att}.

$$CLS = CLS^{beh} \oplus CLS^{att}$$

We assume that the members of class *CLS* are CLS_1, CLS_2, …, CLS_k and the members of class *SCL* are SCL_1, SCL_2, …, SCL_k.

$$CLS = \{CLS_1, CLS_2, ..., CLS_k\} \quad \text{and} \quad SCL = \{SCL_1, SCL_2, ..., SCL_k\}$$

Class CLS_1, CLS_2, …, CLS_k are generalizing *SCL*. This means that classes *CLS* have SCL as superclass and define attributes and behaviors generalized from *CLS* as subclasses. In this case, we can denote that class *SCL* will therefore be the intersection of all the class *CLS*. Furthermore, all of classes *CLS* conform to *SCL*, and we can define that *SCL* is a superclass of *CLS* in the class hierarchy. This is expressed formally bellow.

$$SCL = \bigcap_{i=1}^{k} CLS_i$$

In *OOExpert* project, we use bottom-up (generalization) concepts as a basic approach to build a new model of object refinement process. As shown in Figure 2.12, object

refinement with inheritance process begins by listing objects found in the previous object model creation process, and searching similar object names, attributes, and behaviors. If the similar objects are found, the object will be a sub class and a tentative superclass will be generated automatically. The next process is to give the superclass a name. The superclass can be given from user, or automatically generates from similar object names. The result of this process is a class model with inheritance structure. Oppositely, if the similar object cannot be found, the object will be a class model (without inheritance structure) directly. The final result of the object refinement process is a *class model*, which is the combination of class model with inheritance and class model without inheritance.

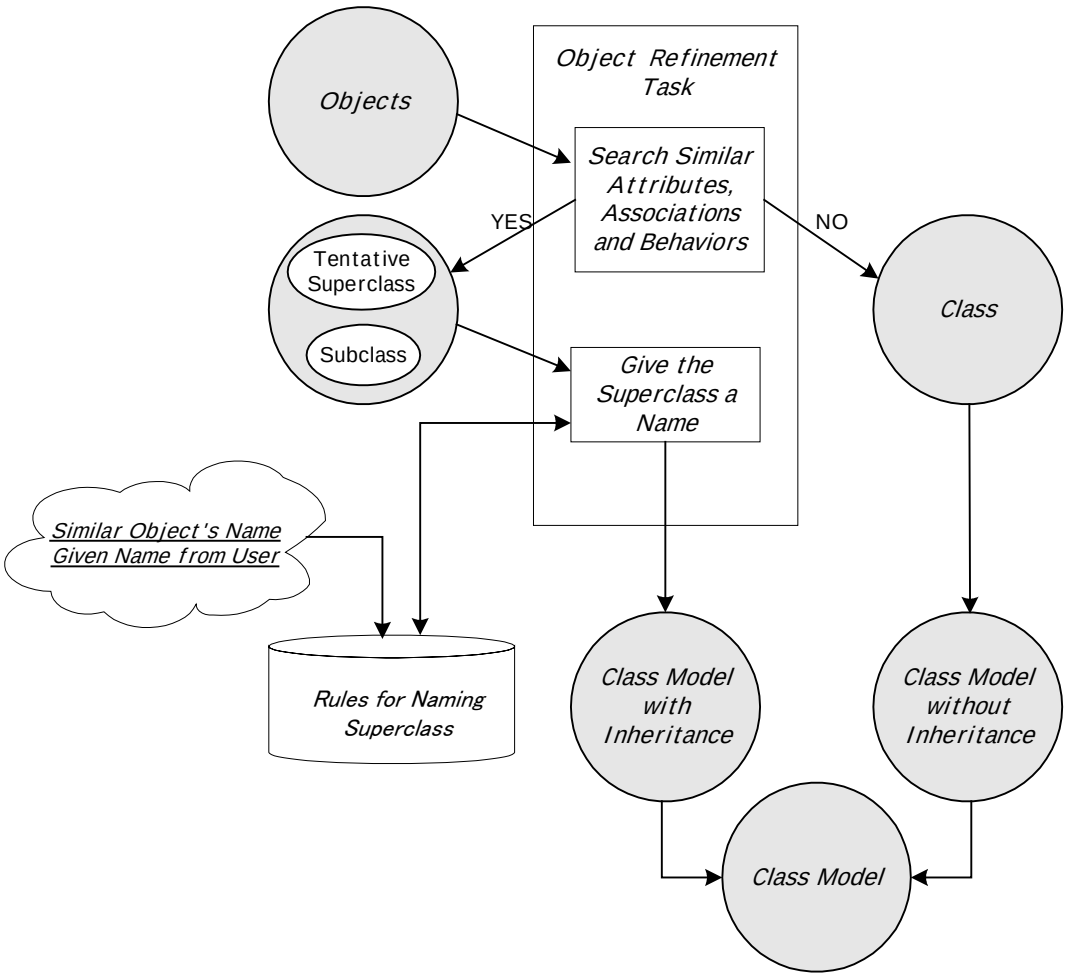

Figure 2.12: Object Refinement with Inheritance

Chapter 3
System Architecture And Design

In this chapter, we focus on how the problems on object model creation process introduced and formalized in the previous section can be designed to be a software system. In our research, object model creation process is viewed as a society of software agents that interact and negotiate with each other. We also construct the *OOExpert* agent framework so that inter-agent communication can be supported as well as the mobility of our agents across network. Finally, we explain system design and architecture of each *OOExpert* agent, including requirements acquisition agent, object identification agent, attribute identification agent, association identification agent, behavior identification, and object refinement agent.

3.1 Agent Model of *OOExpert*

In this research, object model creation process is viewed as a society of software agents that interact and negotiate with each other. We have devised six types of agents: requirement acquisition agent, object identification agent, attribute identification agent, association identification agent, behavior identification agent, and object refinement agent [Figure 3.1]. Each agent is an intelligent in its own field and may interact with its human counterpart or behave autonomously.

- The requirements acquisition agent manages the task concerning the requirements acquisition from *object-based formal specification* (OBFS).
- The *object identification agent* manages the task concerning the object identification.
- The *attribute identification agent* manages the task concerning the identification of object attributes.
- The *association identification agent* manages the task concerning the identification of associations between the identified objects
- The *behavior identification agent* manages the task concerning the identification of object behaviors.
- The *object refinement agent* manages the task concerning to refine objects and organize classes by using inheritance to share common structure

Each agent has a local knowledge base and a reasoning engine. All agents have a communication engine and a documentation engine. The communication and documentation engines facilitate communication and navigation of each agent on the network environment.

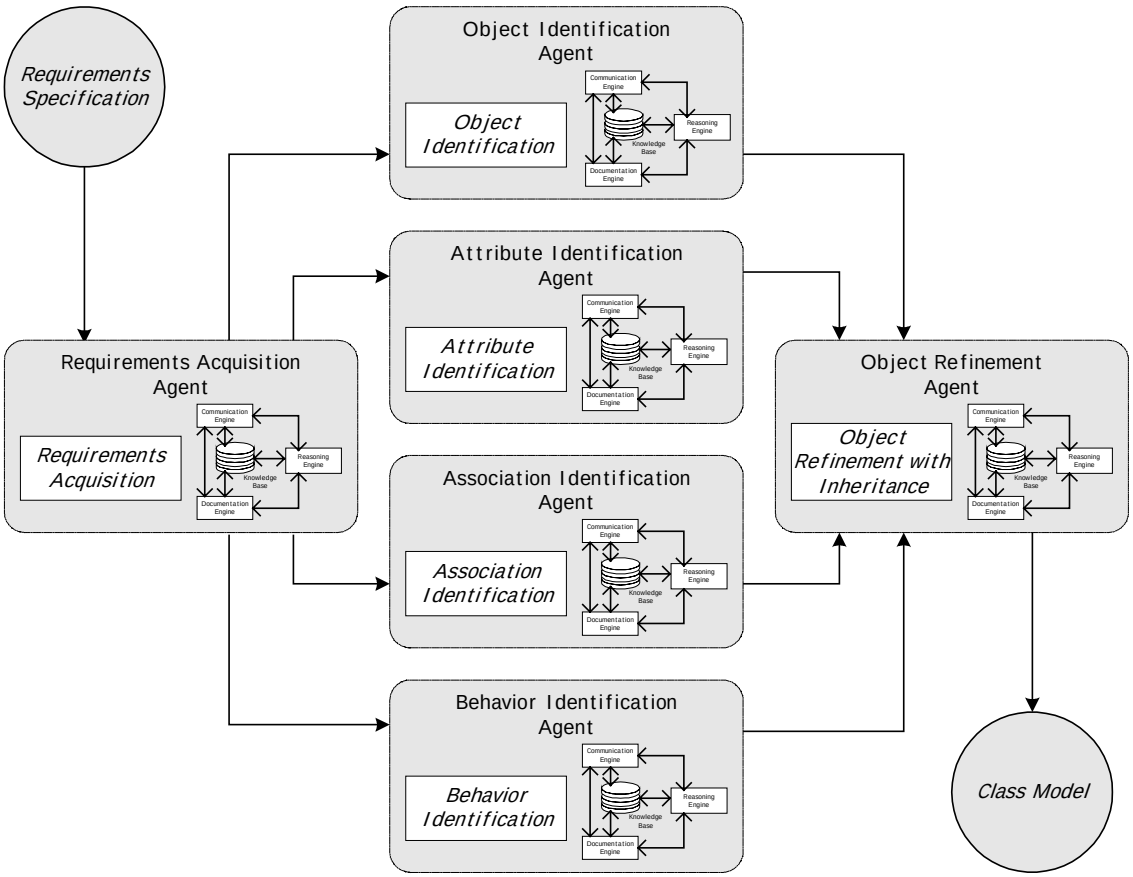

Figure 3.1: Architecture of *OOExpert* Agents

3.2 Agent Framework of *OOExpert*

3.2.1 Issues and Guidelines for *OOExpert* Agent Framework

There are some fundamental issues and guidelines for building a framework of *OOExpert* Agent.

- *OOExpert* agents must support a relatively sophisticated event-processing capability. OOExpert agents will need to handle events from the outside world, other agents, and signal events to outside applications. Java release features a powerful new event-processing model called the Delegation Event Model. This new framework was actually driven by the requirements of JavaBeans component model. This model is based on event sources and event listeners. There are many

different classes of events with different levels of granularity. We will use the JavaBeans event model in our work.

- Domain knowledge must be added to *OOExpert* agents using Rule-Based Reasoning (RBR) and Case-Based Reasoning (CBR) paradigms.
- Learning algorithms must be added to *OOExpert* agents to do classification, clustering and prediction.
- *OOExpert* agents must be supported using a KQML message protocol. In order to provide this functionality, we will have to go back to the drawing board, and come up with an agent that can handle tasks like a KQML facilitator or matchmaker.
- *OOExpert* agents should be persistent. That is, once an agent is constructed, there must be a way to save it in file and reload its state at a later time.

3.2.2 Communication Engine

An agent is an active object with the ability to perceive, reason and act. We assume that an agent has explicitly represented knowledge and mechanism for operating on or drawing inferences on its knowledge. We also assumed that an agent has the ability to communicate. This ability is part perception (the receiving of messages) and part action (the sending of messages). Furthermore, when our agents need to talk to each other, they can do this in a variety of ways. They can talk directly to each other, provided they speak the same language. Or they can talk through an interpreter or facilitator, providing they know how to talk to the interpreter, and the interpreter can talk to the other agent.

There is a level of basic language (the syntax and format of the messages), and there is a deeper level (the meaning or semantics). While the syntax is often easily understood, the semantics are not. For example, two English-speaking agents may get confused if one talks about the boot and bonnet, and the other about the hood and trunk of an automobile. They need to have a shared vocabulary of words and their meaning. This shared vocabulary is called ontology.

OOExpert agents use Knowledge Query and Manipulation Language (KQML) **[Finin et al., 1993] [Finin et al., 1994] [Labrou et al., 1997]** as an agent communication language. The KQML provides a framework for programs and agents to exchange information and knowledge. KQML and also Knowledge Interchange Format (KIF) **[Genesereth et al., 1992]** came out of the DARPA Knowledge Sharing Effort (KSE).

Whereas KIF deals with knowledge representations, KQML focuses on message format and message-handling protocols between running agents. KQML defines the operations that agents may attempt on each other's knowledge bases, and provide a basic architecture for agents to share knowledge and information through special agents called facilitators. Facilitators act as matchmakers or secretaries for the agents they service.

KQML messages are called performatives. Each message is intended to implicitly perform some specified action. There are a large number of performatives defined in KQML, and most agent-based systems support only a small subset. The performatives, or message types, are reserved words in KQML. Using performatives, agents can ask other agents for information, tell other agents facts, subscribe to the services of agents, and offer their own services. KQML uses ontologies, explicit specifications of the meaning, concepts, and relationships applicable to some specific domain, to insure that two agents communicating in the same language can correctly interpret statements in that language.

At the heart of KQML are more than three dozen performatives that define the allowed "speech acts" that agents may use, and which provide the substrate for constructing more complex co-ordination and negotiation strategies. These performatives are grouped into nine categories, as shown in Table 3.1.

Category	Reserved Performative Names
Basic Informational Performatives	tell, deny, untell, cancel
Basic Query Performatives	valuate, reply, ask-if, ask-about, ask-one, ask-all, sorry
Multi-Response Query Performatives	stream-about, stream-all
Basic Effector Performatives	achieve, unachieve
Generator Performatives	standby, ready, next, rest, discard, generator
Capability Definition Performatives	advertise
Notification Performatives	subscribe, monitor
Networking Performatives	register, unregister, forward, broadcast, pipe, break
Facilitation Performatives	broker-one, broker-all, recommend-one, recommend-all, recruit-one, recruit-all

Table 3.1: KQML Performatives

The KQML language can be viewed as consisting of three layers, or three different architectural levels: the content, message and communication layers, as shown in Figure 3.2.

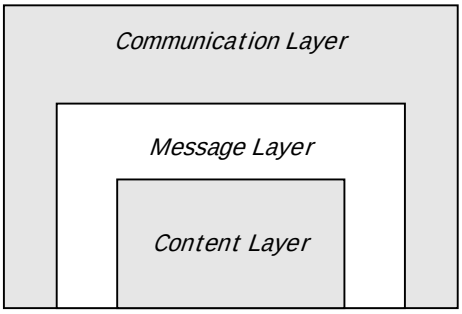

Figure 3.2: An Abstract View of the KQML Language

An example of a KQML message from agent *AGENT-A* asking about the price of a share of *MICROSOFT* stock might be encoded as:

(ask-one
:sender AGENT-A
:content (REAL PRICE = MICROSOFT.PRICE())
:receiver STOCK-SERVER
:reply-with MICROFT-STOCK
:language JAVA
:ontology MUS)

The KQML performative is *ask-one*; the receiver of the message is an agent named *STOCK-SERVER*. The *:content* parameter completely defines the content level. The *:reply-with*, *:sender*, and *:receiver* parameters specify information at the communication level. The performative name, the *:language* specification, and the *:ontology* name are part of the message level.

3.2.3 Reasoning Engine and Knowledge Base

Hierarchical Structure of Knowledge Levels

It is believed that human experts possess a conceptual model of how the objects in the external world interact based on standard operating procedures. Conceptual models

have a hierarchical structure defined best by the Skill-Rule-Knowledge (S-R-K) levels **[Rasmussen, 1985]** concerning with routine, innovative and creative problem solving tasks, respectively.

- **Level 1 (Skill-Based Level)**: This level is deal with routine task. It denotes the kind of task for which problem solving knowledge and strategies are well defined. At this level, reasoning is governed by stored patterns of predefined rules. Such context specific pattern are called rules-of-thumb, that map directly from an observation to a ready-made solution. At this level a query of agent is accepted and by searching the knowledge base, proper immediate action is selected. For instance, in case of search agent the query comes in the form of a list of keywords, submitted by the customer agent. Then search agent finds related keywords and conducts search using the new set of keywords.

- **Level 2 (Rule-Based Level)**: This level deals with the innovative tasks when dealing with familiar or similar problems. It denotes the kind of task for which problem solving knowledge is well-defined. Rule base behavior is conventionally described by case bases, decision tables, diagraphs, fuzzy sets and natural language models. At this level a query of an agent is accepted and a case database is consulted to determine the action. Then a set of similar cases are searched and cases matching the needs of the user are retrieved. Further research is conducted based on the instructions recorded on the matched cases.

- **Level 3 (Knowledge-Based Level)**: This level deals with the creative tasks for which common pattern in stored knowledge form do not exist and reasoning should start from the so called first principles, starting from problem identification. In other words, neither problem solving knowledge nor the strategy is well-defined. At this level a query is accepted and the agent uses its knowledge base to interact with the other agent and identify the actual needs. After this problem identification level, the proper action is determined by consulting other agents.

Reasoning With Rules

Rules may contain much information beyond their simple conditional if-then component. Whereas the antecedent and consequent of a rule specify data sufficient for inferring a conclusion or performing another action, other parts of a rule serve additional important roles. A rule whose antecedent clauses are all true is said to be

triggered or ready to fire. We fire a triggered rule by asserting the consequent clause and adding it is a fact to our working memory. At any time, a rule base may contain several rules that are ready to fire. It is up to the control strategy of the inference engine to decide which one gets fired.

Many rule-based systems benefit from hierarchical structuring, in which each rule may belong to one or more higher order collections. These collections, called rule- sets, aggregate and differentiate rules according to their function within the system. The rule is spoken of as a relatively independent piece or chunk of know-how. Psychologists, for some time, have emphasized the subjective reality of chunks. Chunks correspond to the elementary patterns people perceive and manipulate in thinking. They differ from person to person. They reflect the learned, appropriate, effective distinctions in each person's skill areas. A rule corresponds to a chunk of problem-solving know-how.

Rules are easily manipulated by reasoning systems. Forward chaining can be used to produce new facts and backward chaining can deduce whether statements are true or not.

Forward chaining is a data-driven reasoning process where a set of rules is used to derive new facts from an initial set of data. It does not use the resolution algorithm used in predicate logic. The forward chaining algorithm generates new data by simple and straightforward application or firing of the rules. Forward chaining is also used in real-time monitoring and diagnostic system where quick identification and response to problems are required. The following steps are part of the forward-chaining cycle:

1. Load the rule base into the inference engine, and any facts from the knowledge base into the working memory.
2. Add any additional initial data into the working memory.
3. Match the rules against the data in working memory and determine which rules are triggered, meaning that all of their antecedent clauses are true. This set of triggered rules id called the conflict set.
4. Use the conflict resolution procedure to select a single rule from the conflict set.
5. Fire the selected rule by evaluating the consequent clause(s); either update the working memory if it is a fact-generating rule, or call the effectors procedure, if it is an action rule. This is referred to act step.
6. Repeat step 3,4, and 5 until the conflict set is empty.

Backward chaining is often called goal-directed inferencing, because a particular

consequence or goal clause is evaluated first, and then we go backward through the rules. Unlike forward chaining, which uses rules to produce new information, backward chaining uses rules to answer questions about whether a goal clause is true or not. Backward chaining is more focused than forward chaining, because it only processes rules that are relevant to the question. It is similar to how resolution is used in predicate logic. However, it is does not use contradiction. It simply traverses the rule base trying to prove that clauses are true in a systematic manner. The following steps are part of the forward-chaining cycle:

1. Load the rule base into the inference engine, and any facts from the knowledge base into the working memory.
2. Add any additional initial data into the working memory.
3. Specify a goal variable for the inference engine to find.
4. Find the set of rules, which refer to the goal variable in a consequent clause. That is, find all rules which set the value of the goal variable when they fire. Put each rule on the goal stack.
5. If the goal stack is empty, halt.
6. Take the top rule off the goal stack.
7. Try to prove the rule is true by testing all antecedent clauses to see if they are true.

Integrating RBR and CBR paradigms for *OOExpert* Agents

The recent evolution of hybrid architectures for knowledge-based systems has resulted in several approaches that combine rule-base reasoning (RBR) with case-based reasoning (CBR) techniques to engender performance improvements over more traditional one-representation architectures.

CBR is used in learning and problem-solving systems to solve new problems by recalling and reusing specific knowledge obtained from past experience. RBR systems learn general domain-specific knowledge from a set of training data and represent the knowledge in comprehensible form as *if-then* rules. Due to their complementary properties, CBR and RBR techniques have been combined in some systems to solve problems to which single technique fails to provide a satisfactory solution.

We also use RBR and CBR integration approach for *OOExpert* agents' reasoning engine. RBR paradigm is used to eliminate spurious objects and propose relevant objects. In the RBR, the system will discard unnecessary and incorrect objects. Furthermore, CBR is used to propose relevant objects according to the human expert solution and problem

domain relation. For this reason, we set two kinds of case indexed in case-base, there are: *Human Expert Solution* and *Problem Domain Relation*.

3.3 Design of *OOExpert* Agents

According to each task of object model creation process, that is proposed in the previous chapter, we have devised six types of agents: requirement acquisition agent, object identification agent, attribute identification agent, association identification agent, behavior identification agent, and object refinement agent. This is expressed on a design picture bellow.

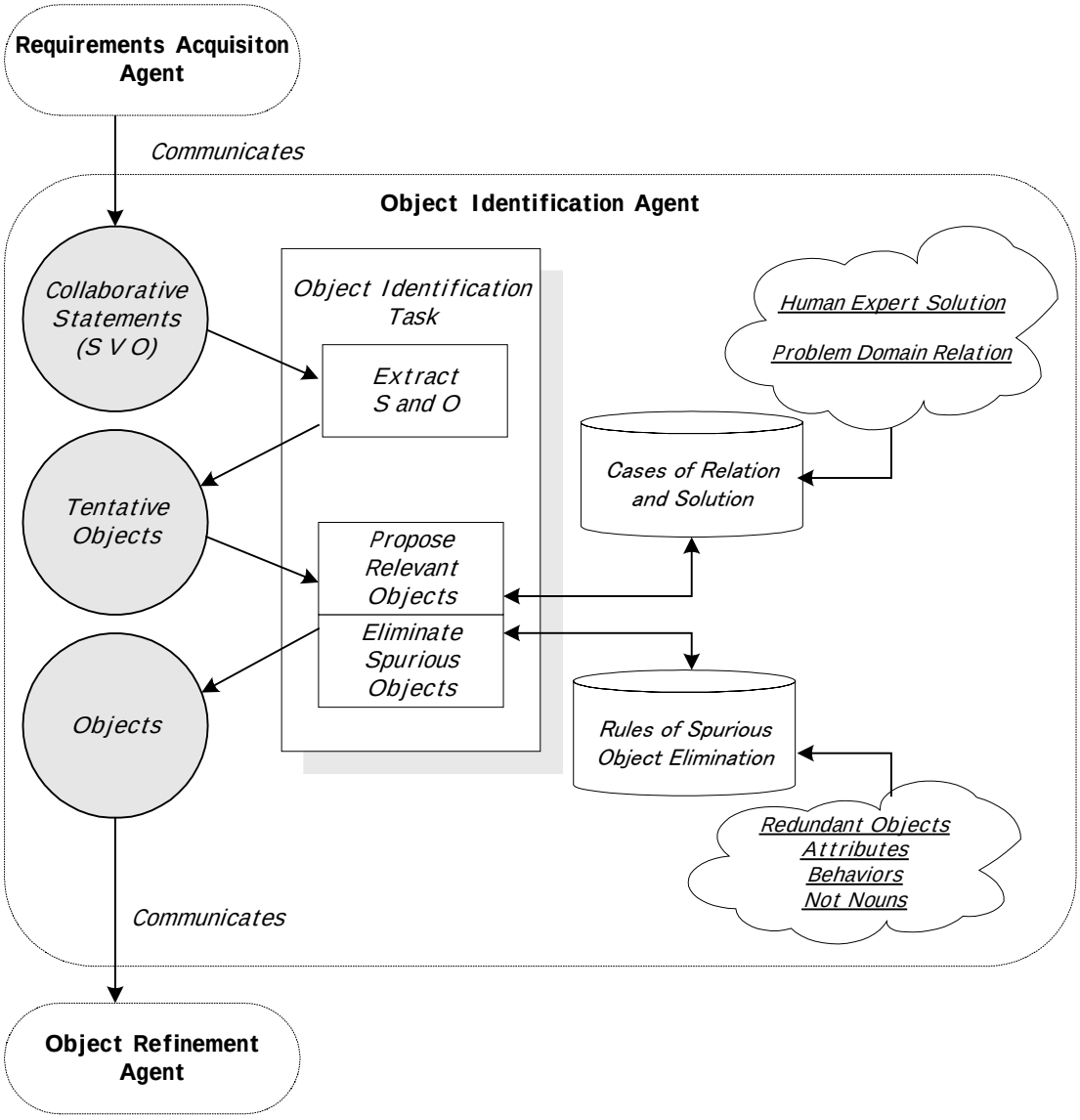

Figure 3.3: Object Identification Agent

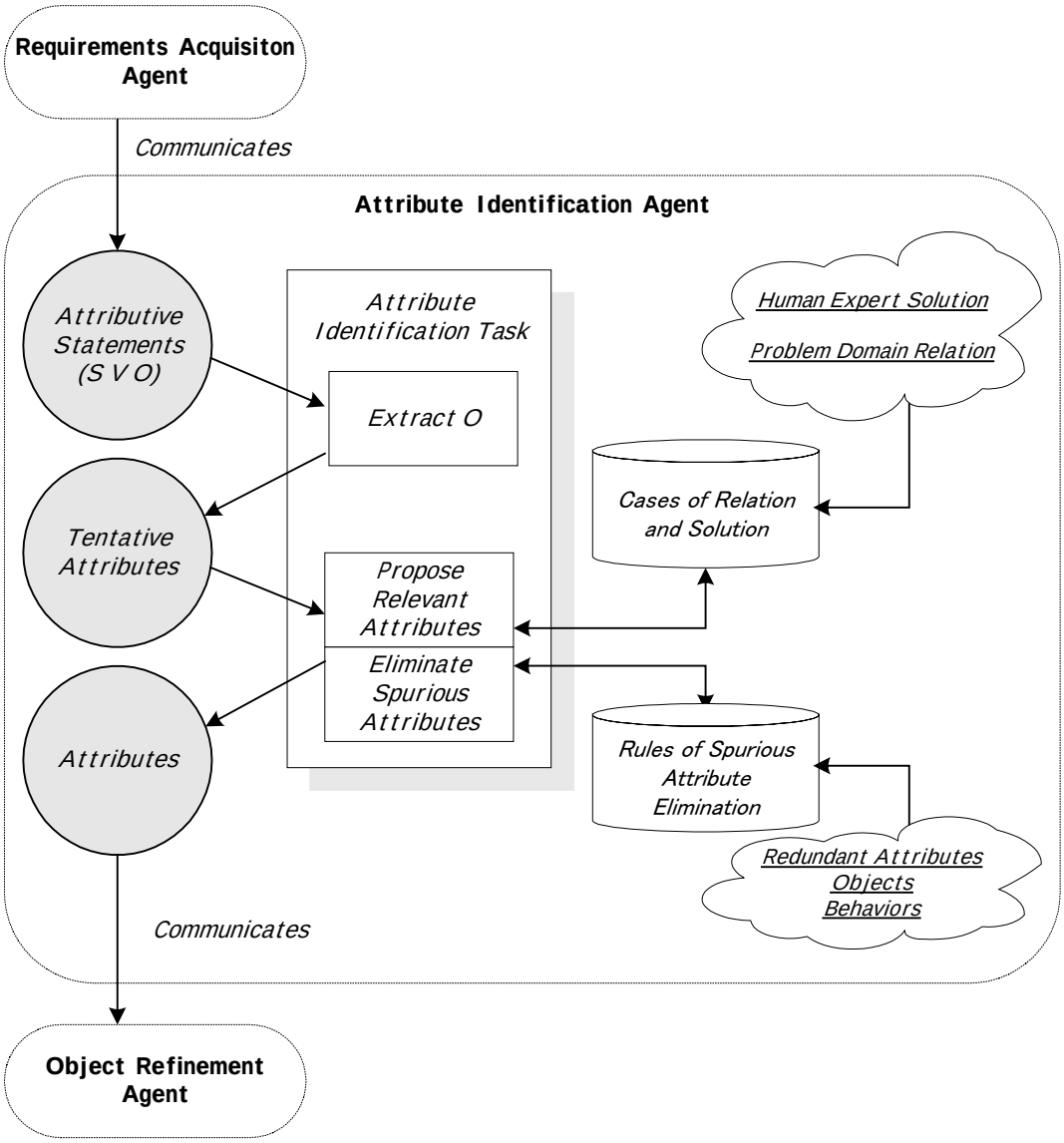

Figure 3.4: Attribute Identification Agent

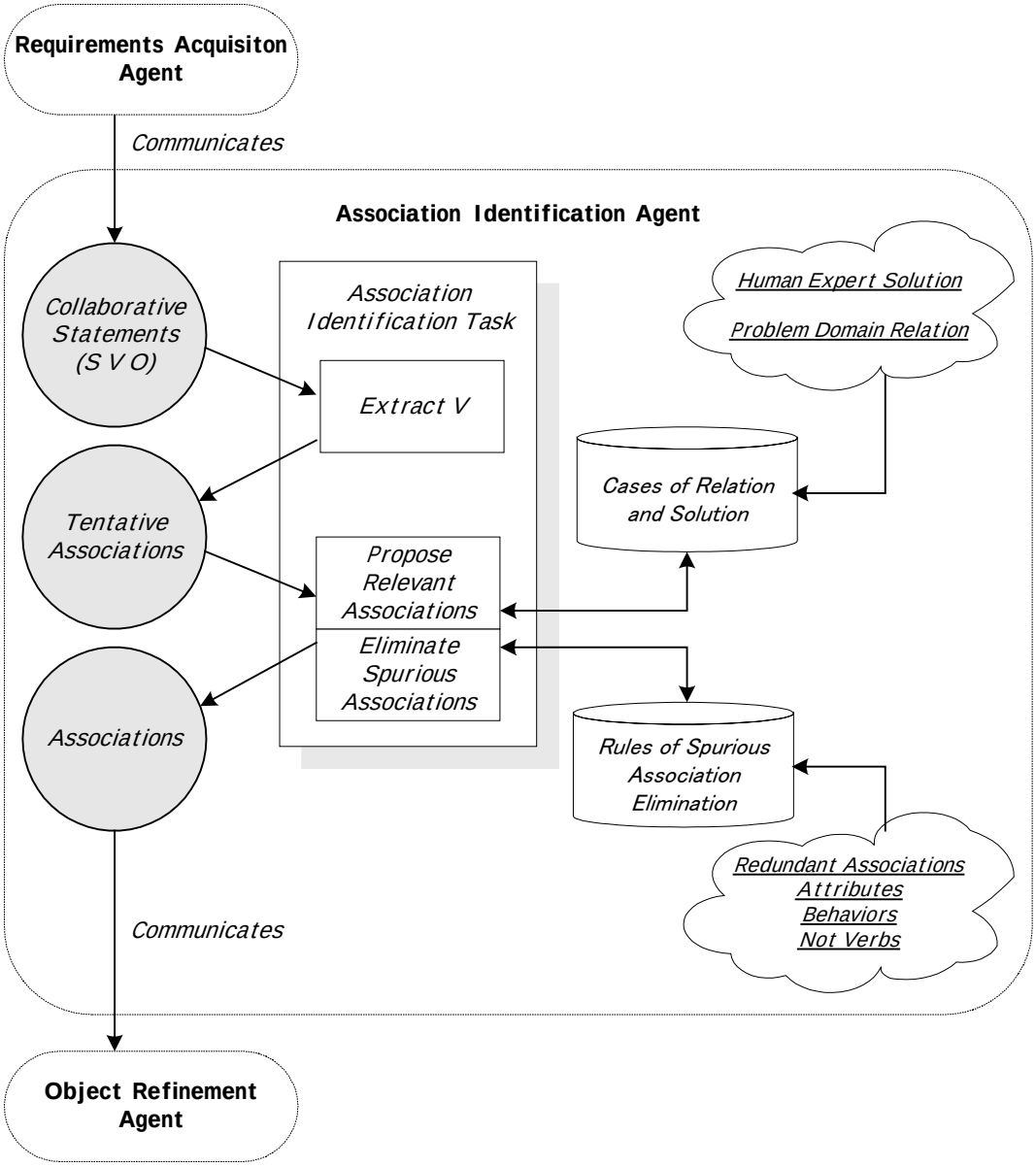

Figure 3.5: Association Identification Agent

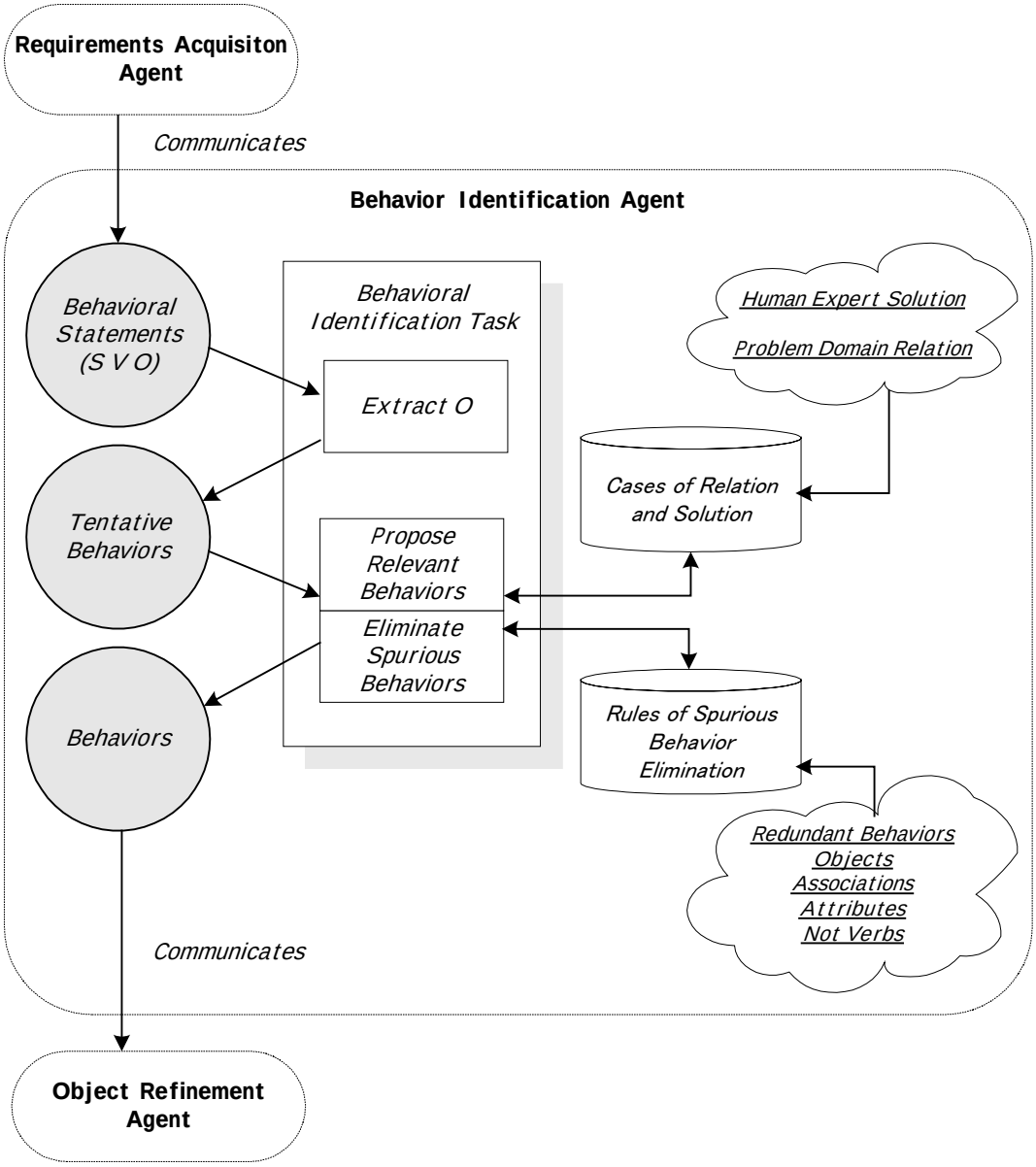

Figure 3.6: Behavior Identification Agent

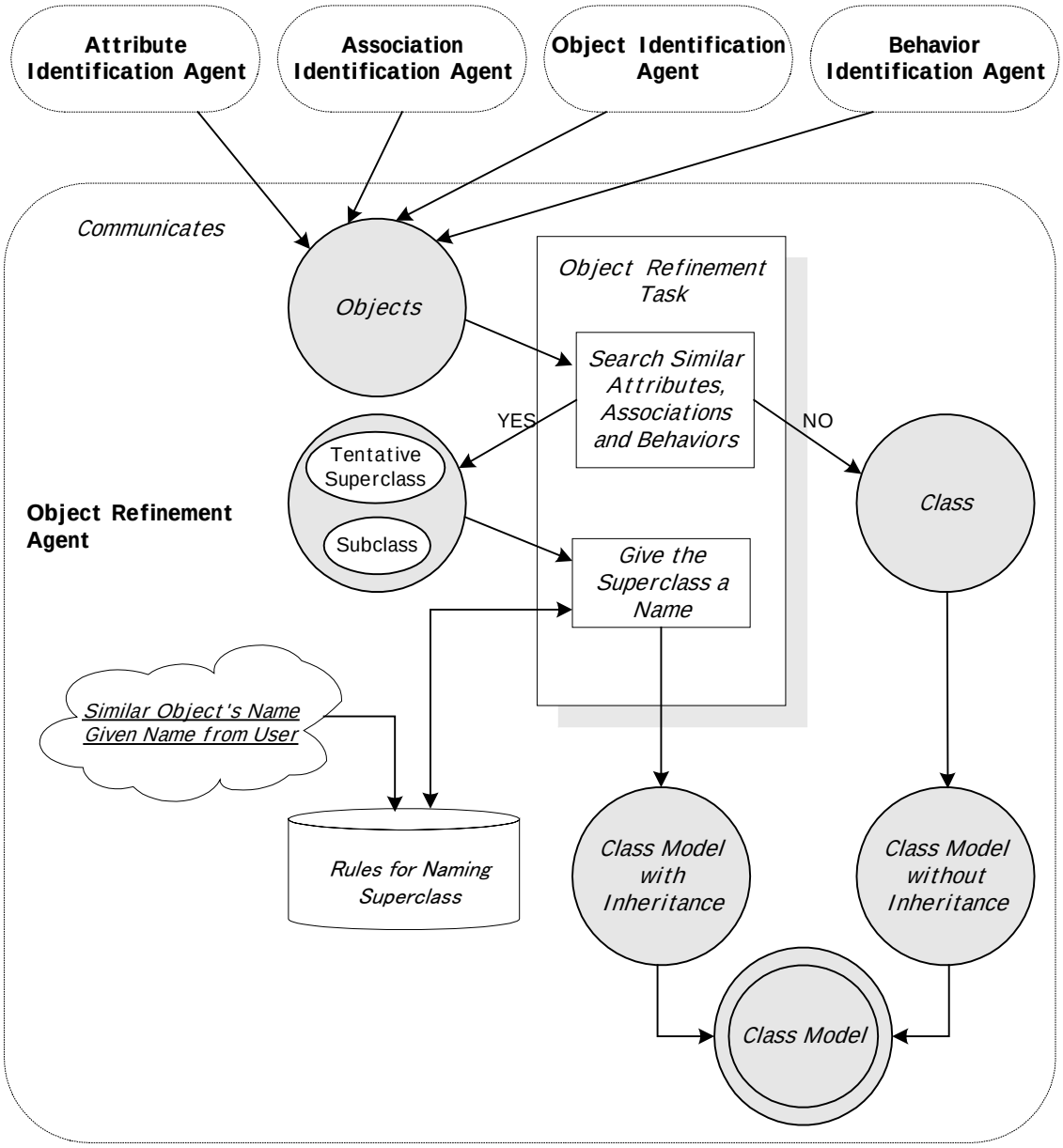

Figure 3.7: Object Refinement Agent

Chapter 4
Implementation

In this chapter, we focus on how the problems on object model creation process introduced, formalized, and designed in the previous section can be implemented to be a software system. It starts with an explanation about why Java is used as programming language to implement *OOExpert* agents. However, There are specific features of Java, which support intelligent agent paradigm: autonomy, intelligence and mobility. How the *OOExpert* agents work is also presented at the end of this chapter.

4.1 Implementing *OOExpert* Agents Using Java

4.1.1 The Main Reasons to Deal with Java

Java is an object-oriented programming language developed by Sun Microsystems. It was originally designed for programming real-time embedded software for customer electronics, particularly set-top boxes to interface between cable, provider, broadcasters, and televisions or television like appliances. Six main reasons to deal with Java programming language can be identified.

(a) Java Supports Intelligent Agent Application

There are specific features of Java, which support intelligent agent paradigm: **[Bigus et al., 1997]**

- **Autonomy**: For software program to be autonomous, it must be a separate process of thread. Java applications are separate processes and as such can be long running and autonomous. Java applications can communicate with other programs using sockets. In an application, an agent can be a separate thread of control. Java supports threaded applications and provide support for autonomy using both using both techniques. In the Introduction chapter, we described intelligent agents as autonomous programs or process. As such they are always waiting, ready to respond to a user request or a change in the environment. One question that comes to mind is "How does the agent know when something changes?" As with many others, the agent is informed by sending it an event. From an object-oriented design perspective, an event is nothing more than a method call or message, with information passed along on the method call, which defines what happened or what action we want the agent to perform, as well as data required to process the event.

- **Intelligence**: The intelligent in intelligent agents can range from hard coded procedural or object-oriented logic to sophisticated reasoning and learning capabilities. While Prolog and Lisp are the two languages usually associated with artificial intelligence programming, in recent years, much of the commercial AI work has been coded in C and C++. As a general purpose, object-oriented

programming language, Java provides all of the base function needed to support these behaviors. There are two major aspects to AI applications, knowledge representation and algorithms, which manipulate those representations. All knowledge representations are based on the use of slots or attributes, which hold information regarding some entity, and links or references or other entities. Java objects can be used to encode this data and behavior as well as the relationships between objects. Standard AI knowledge representation such as frames, semantic nets, and if-the rules can be easily and naturally implemented using Java.

- **Mobility**: There are several different aspects to mobility in the context of intelligent agents and intelligent applications. Java's portable bytecodes and JAR files allow groups of compiled Java classes to be sent over a network and then executed on the target machine. One of the prime requirements for mobile programs is the ability to save the state of the running process, ship it off, and then resume where the process left off, only now it is running on different system. Computer science researchers have explored this topic in great detail in relation to load balancing on distributed computer systems such as networks of workstations. Having homogeneous machines was a crucial part of making this work. Once again, the Java Virtual Machine (JVM) comes to rescue. By providing a standard computing environment for a Java process to run in, the JVM provides a homogenous virtual machine that allows java agents to move between heterogeneous hardware systems without losing a beat.

(b) Simple, Object Oriented, and Familiar

Primary characteristics of Java include a simple language that can be programmed without extensive programmer training while being attuned to current software practices. The fundamental concepts of Java are grasped quickly; programmers can be productive from the very beginning. Java is designed to be object oriented from the ground up. Object technology has finally found its way into the programming mainstream after a gestation period of thirty years. The needs of distributed, client-server based systems coincide with the encapsulated, message-passing paradigms of object-based software. To function within increasingly complex, network-based environments, programming systems must adopt object-oriented concepts. Java provides a clean and efficient object-based development platform. Programmers using Java can access existing libraries of tested objects that provide functionality ranging from basic data types through I/O and network interfaces to graphical user interface toolkits. These libraries

can be extended to provide new behavior. Even though C++ was rejected as an implementation language, keeping Java looking like C++ as far as possible results in Java being a familiar language, while removing the unnecessary complexities of C++.

(c) Robust and Secure

Java is designed for creating highly reliable software. It provides extensive compile-time checking, followed by a second level of run-time checking. Language features guide programmers towards reliable programming habits. The memory management model is extremely simple: objects are created with a new operator. There are no explicit programmer-defined pointer data types, no pointer arithmetic, and automatic garbage collection. This simple memory management model eliminates entire classes of programming errors that bedevil C and C++ programmers. Java is designed to operate in distributed environments, which means that security is of paramount importance. With security features designed into the language and run-time system, Java lets us construct applications that can't be invaded from outside. In the network environment, applications written in Java are secure from intrusion by unauthorized code attempting to get behind the scenes and create viruses or invade file systems.

(d) Architecture Neutral and Portable

Java is designed to support applications that will be deployed into heterogeneous network environments. In such environments, applications must be capable of executing on a variety of hardware architectures. Within this variety of hardware platforms, applications must execute atop a variety of operating systems and interoperate with multiple programming language interfaces. To accommodate the diversity of operating environments, the Java compiler generates bytecodes, an architecture neutral intermediate format designed to transport code efficiently to multiple hardware and software platforms. The interpreted nature of Java solves both the binary distribution problem and the version problem; the same Java language byte codes will run on any platform. Architecture neutrality is just one part of a truly portable system. Java takes portability a stage further by being strict in its definition of the basic language.

(e) High Performance

Performance is always a consideration. Java achieves superior performance by adopting a scheme by which the interpreter can run at full speed without needing to check the run-time environment. The automatic garbage collector runs as a low-priority background thread, ensuring a high probability that memory is available when required,

leading to better performance. Applications requiring large amounts of compute power can be designed such that compute-intensive sections can be rewritten in native machine code as required and interfaced with the Java platform. In general, users perceive that interactive applications respond quickly even though they're interpreted.

(f) Interpreted, Threaded, and Dynamic

The Java interpreter can execute Java bytecodes directly on any machine to which the interpreter and run-time system have been ported. In an interpreted platform such as Java system, the link phase of a program is simple, incremental, and lightweight. You benefit from much faster development cycles--prototyping, experimentation, and rapid development are the normal case, versus the traditional heavyweight compile, link, and test cycles. Modern network-based applications typically need to do several things at the same time. Java's multithreading capability provides the means to build applications with many concurrent threads of activity. Multithreading thus results in a high degree of interactivity for the end user. Java supports multithreading at the language level with the addition of sophisticated synchronization primitives: the language library provides the Thread class, and the run-time system provides monitor and condition lock primitives.

4.1.2 Implementing Reasoning Engine Using Java

Figure 4.1 shows the strategy to implement rule-based reasoning using object-oriented programming. This implementation includes a *Rule* class, a *Variable* class, a *RuleVariable* class, a *RuleBase* class, a *Clause* class, a *Fact* class, and also a *Sensor* class and an *Effector* class to inference the rule functionalities.

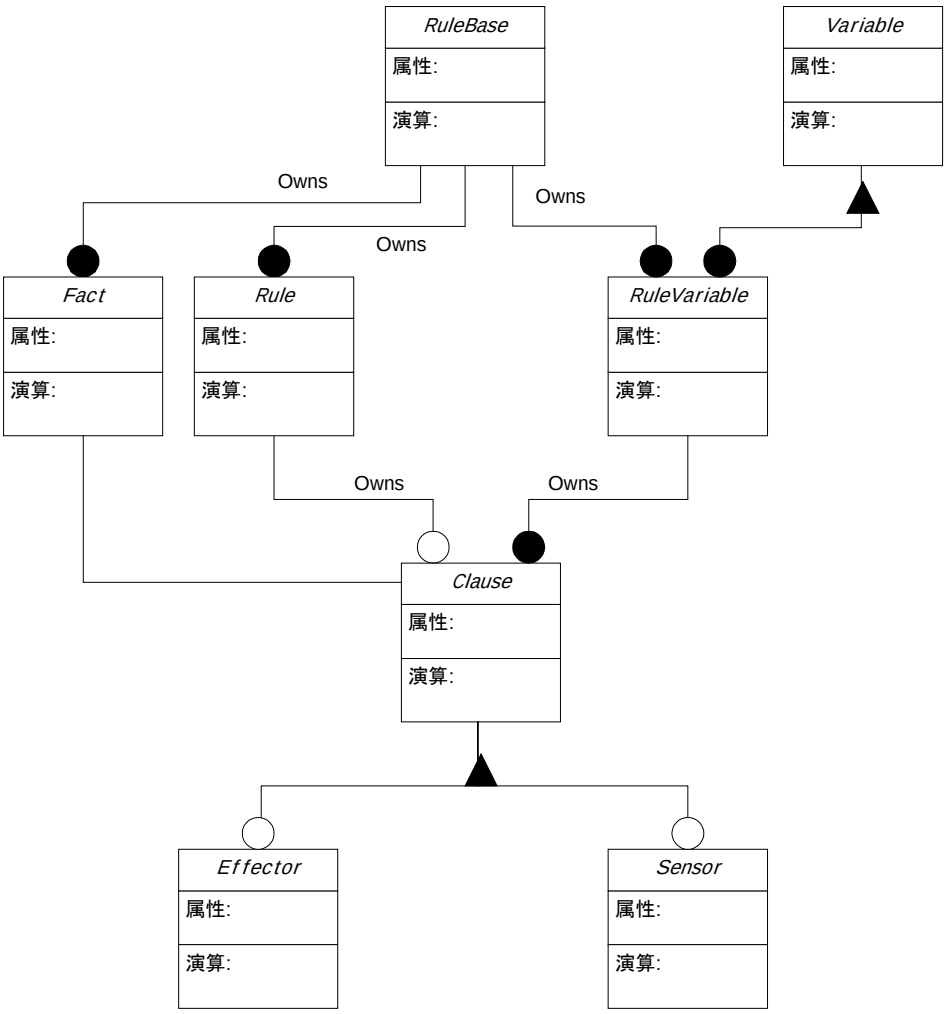

Figure 4.1: The Object Model of Rule-Based Reasoning

Rule Class: The *Rule* class is used to define a single rule and also contains methods that support the inferencing process. Each *Rule* class has a name data member, a reference to the owning *RuleBase* object, an array of antecedent clauses and a single consequent clause.

Variable Class: We define a base class for variables, which support the function we need for rule processing and for learning in the next chapter. The *Variable* class has a name member to identify the variable, and a string value member.

RuleVariable Class: For rule processing, we subclass our *Variable* class and add some rule-specific behavior. It provides the support necessary for variables used in inferencing. The constructor takes the name of the variable as the only parameter.

RuleVariable class inherit the discrete symbolic behavior of the base *Variable* class

RuleBase Class: The *RuleBase* class defines a set of *RuleVariable* and *Rule* class, along with the high level methods for forward and backward chaining. The *RuleBase* has a name, a variable list, which contains all of the *RuleVariable* class referenced by the *Rule* class, and the rule list, which contains all of the *Rule* class.

Clause Class: *Clause* class is used both in the antecedent and consequent part of a *Rule* class. A *Clause* class is usually made up of a *RuleVariable* class on the left-hand side, a condition, which tests equality, greater than, or less than, and the right-hand side, which in our implementation is a string value.

Fact Class: To support facts, we add a new class called *Fact* whose constructor takes a single clause as a parameter. A fact can be an assignment of a value to a *RuleVariable* class, a sensor call, or an effector call. The *Facts* are defined as part of the *Rulebase* class with the other rule.

Sensor and Effector Class: *Sensor* and *Effecor* class is an instance of *Clause* class. The *Sensor* class makes a call to a sensor method and registers it with the *RuleBase* class. At runtime the *RuleBase* looks up the sensor name and calls the method on the registered sensor object. A similar technique is used for the *Effector* class.

4.1.3 Implementing Communication Engine Using Java

OOExpert agents must communicate with other agents in order to work flexibly and autonomously. We have considered building Java-based, KQML messaging and socket pipe communicating agents that communicate over the network environment. A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent. The communication engine is mainly responsible for maintaining connection to the network, communicating with other agents and managing messages.

The following *KQMLMessage* class is used as the argument object in the related communication engine classes. It is basically a collection of data that corresponds to the

major slots of a KQML message.

```
import java.io.*;
import java.util.*;
import java.lang.*;

public class KQMLMessage{

  String performative, content, inReplyTo, language ;
  String ontology, receiver, replyWith, sender ;

  KQMLMessage(String Performative, String Content, String InReplyTo, String Language,
                    String Ontology, String Receiver, String ReplyWith, String Sender){

    performative        = Performative ;
    content             = Content ;
    inReplyTo           = InReplyTo ;
    language            = Language ;
    ontology            = Ontology ;
    receiver            = Receiver ;
    replyWith           = ReplyWith ;
    sender              = Sender ;
  }

  KQMLMessage(String Performative, String Content, String InReplyTo,
                    String Receiver, String ReplyWith, String Sender){

    performative        = Performative ;
    content             = Content ;
    inReplyTo           = InReplyTo ;
    receiver            = Receiver ;
    replyWith           = ReplyWith ;
    sender              = Sender ;
  }

  KQMLMessage(String Performative, String Content,
                    String Receiver, String Sender){

    performative        = Performative ;
    content             = Content ;
    receiver            = Receiver ;
    sender              = Sender ;
  }

  public void display() {
    System.out.println(    "Performative: "     + performative        + "\n" +
                           "Content: "          + content             + "\n" +
                           "InReplyTo: "        + inReplyTo           + "\n" +
                           "Language: "         + language  + "\n" +
                           "Ontology: "         + ontology            + "\n" +
                           "Receiver: "         + receiver            + "\n" +
                           "ReplyWith: "        + replyWith           + "\n" +
                           "Sender: "           + sender              + "\n");
  }

  public void displaySimple() {
    System.out.println(     "Performative: "    + performative        + "\n" +
                            "Content: "         + content             + "\n" +
                            "Receiver: "        + receiver            + "\n" +
                            "Sender: "          + sender              + "\n");
  }
}
```

4.2 How the *OOExpert* Works

4.2.1 Getting Started with *OOExpert*

When we start to run *OOExpert* agents, for example Requirement Acquisition Agent, it will display a user interface window as shown in Figure 4.2. The user interface window contains a *standard toolbar*, *the viewer for OBFS tree and directories tree*, and a *control window*. Requirements Acquisition Agent displays Object Based Formal Specification (OBFS) menu in the control window, including Description Statements, Collaborative Statements, Attributive Statements, Behavioral Statements and Inheritance Statements. The user writes requirements in this place based on OBFS standard. Especially for other *OOExpert* agents, the reasoning processes of agents are displayed in this control window.

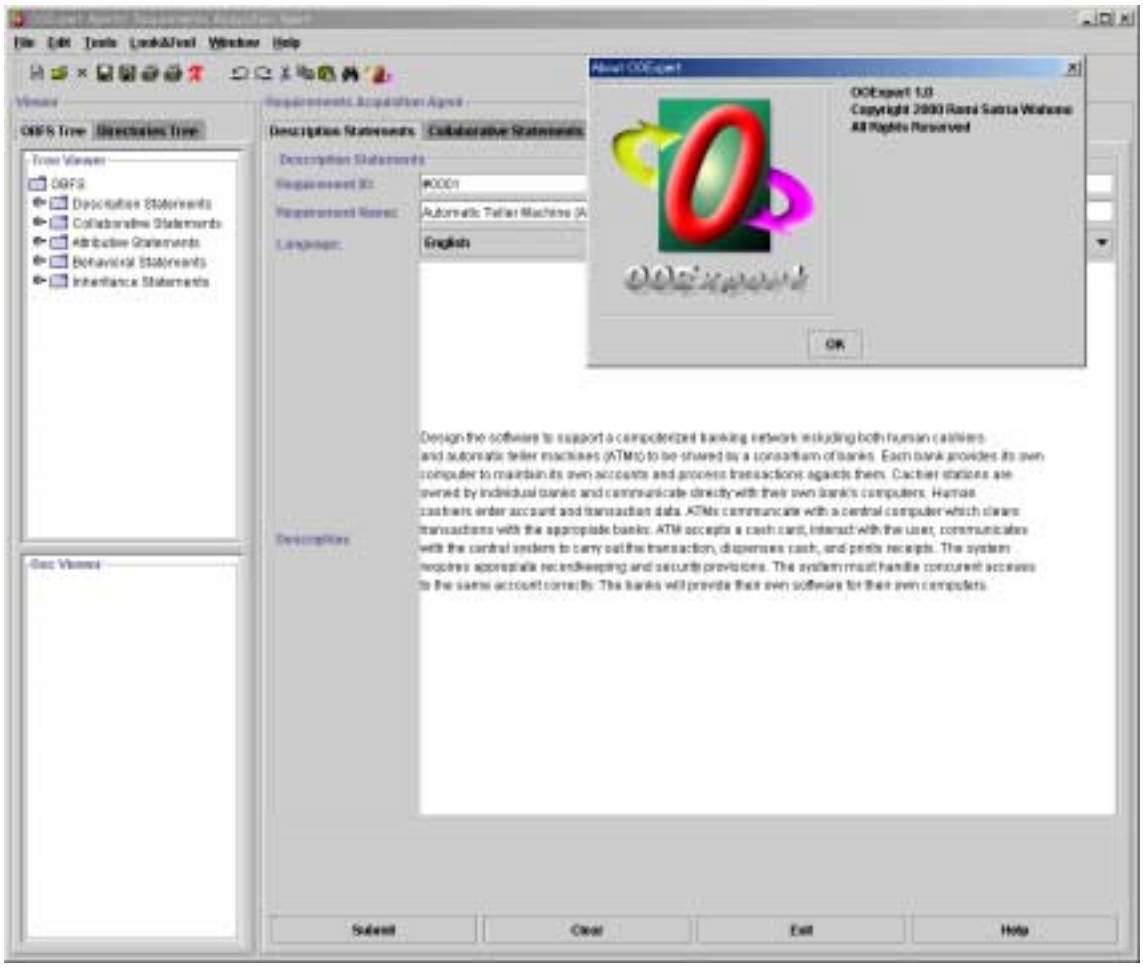

Figure 4.2: Requirements Acquisition Agent in Action

4.2.2 Working with *OOExpert*

As already described completely in the previous chapter, *OOExpert* agents is viewed as a society of software agents that interact and negotiate with each other. We have devised six types of agents: requirement acquisition agent, object identification agent, attribute identification agent, association identification agent, behavior identification agent, and object refinement agent. Running all of the *OOExpert* agents are however, the first step toward working with *OOExpert*.

```
[administrator@KISO4] java RequirementsAcquisitionAgent
Status: Connected to Object Identification Agent!
Status: Connected to Association Identification Agent!
Status: Connected to Attribute Identification Agent!
Status: Connected to Behavior Identification Agent!
Status: Connected to Object Refinement Agent!
```

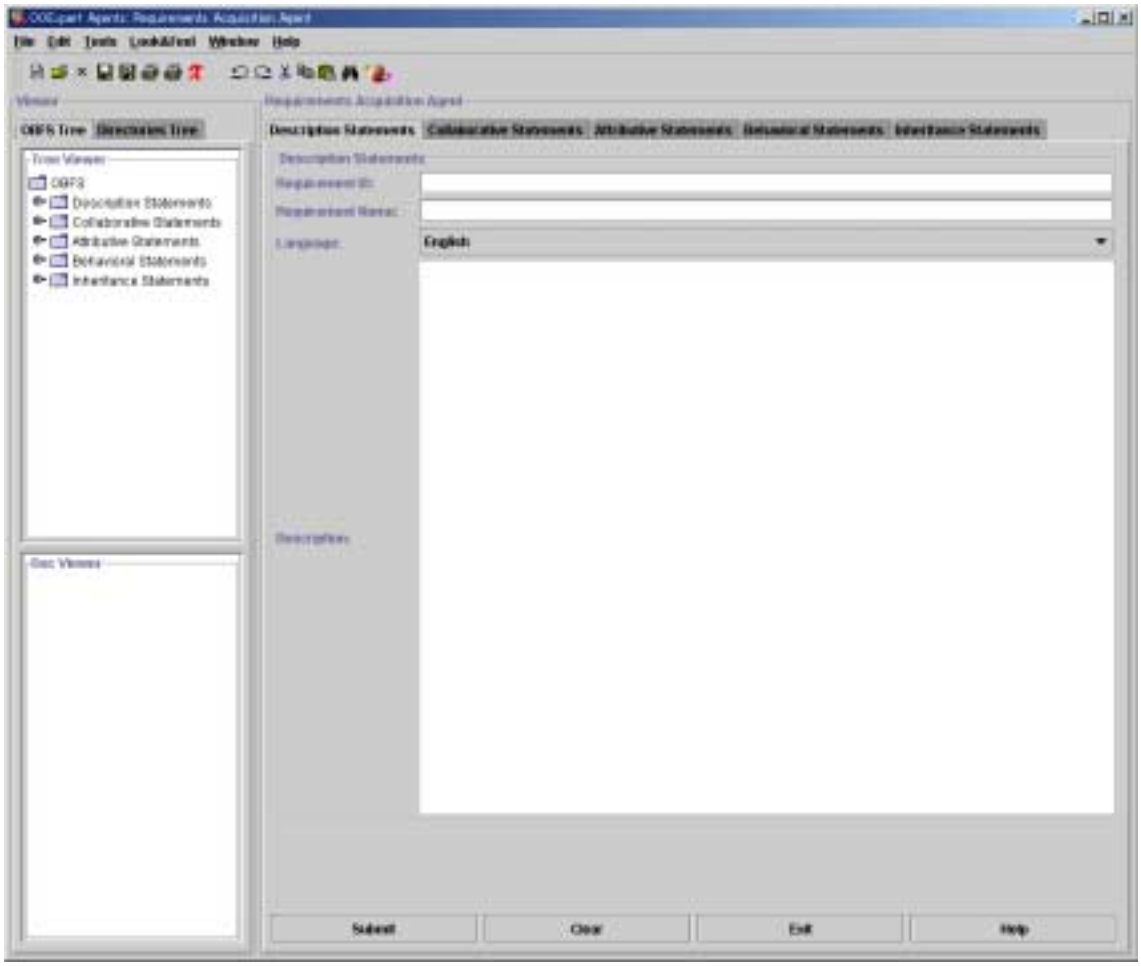

Figure 4.3: Running the Requirements Acquisition Agent

We present an example of the using of the *OOExpert* by create object model for Automatic Teller Machine (ATM) Network System. The *OOExpert*'s running procedures can be divided into six distinguished steps as following:

STEP 1: Requirement Acquisition Process (Requirement Acquisition Agent)

The requirements acquisition agent manages the task concerning the requirements acquisition from object-based formal specification (OBFS).

- The user has to specify Description Statements (DS) about system, which he wants to build (Figure 4.4).

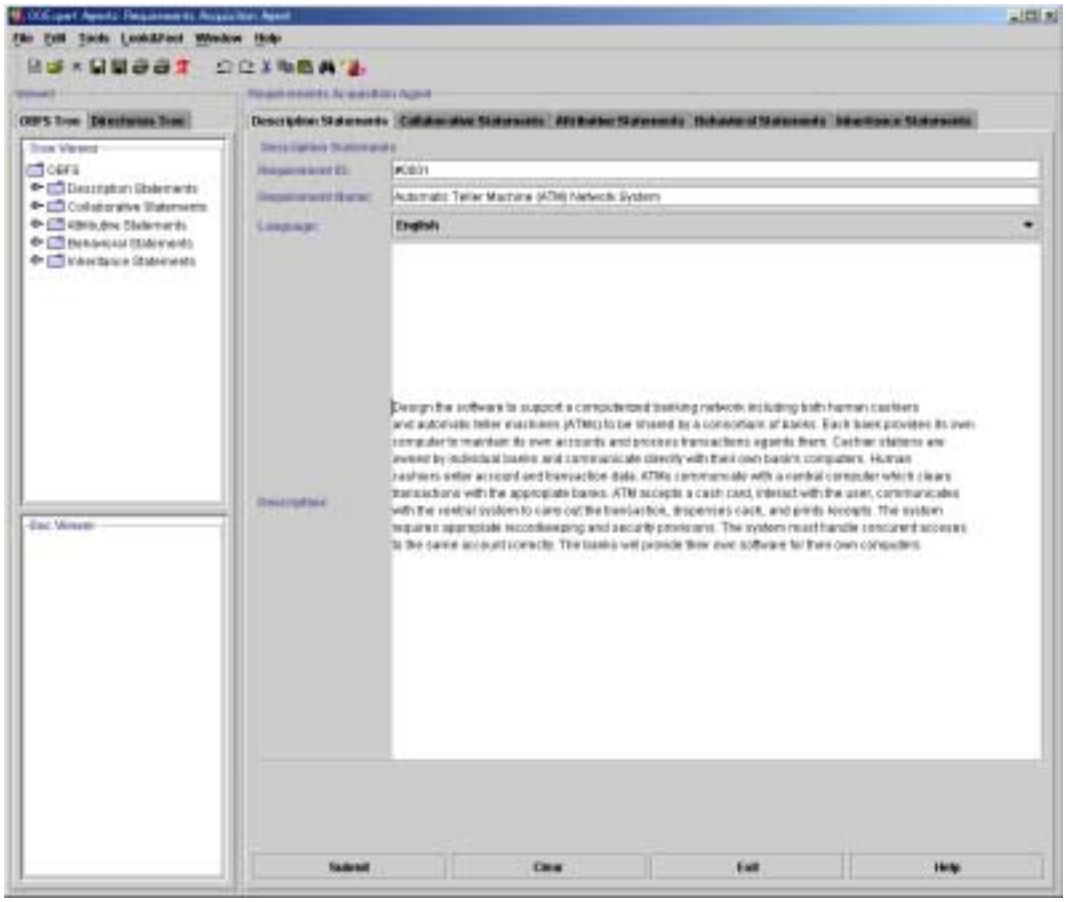

Figure 4.4: Writing the Description Statements

- The user has to specify Collaborative Statements (CS) about system, which he wants to build (Figure 4.5).

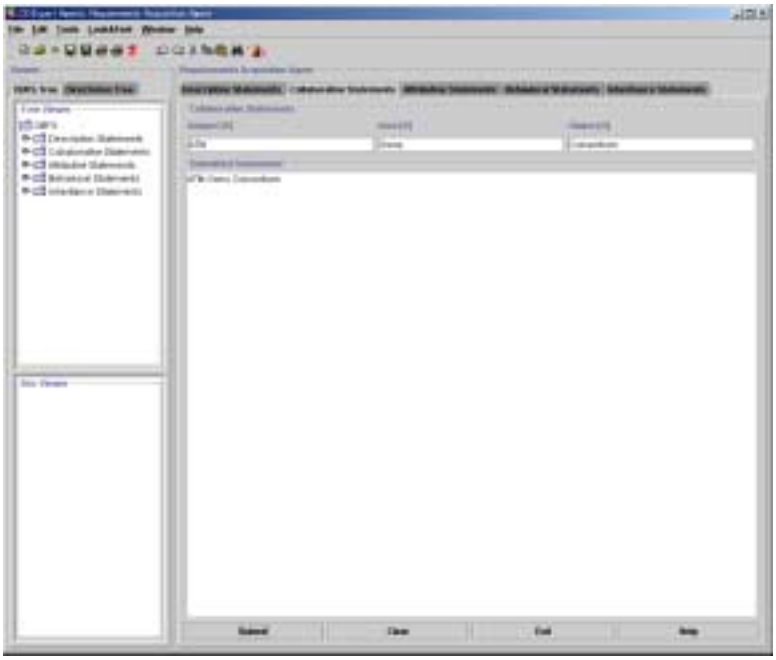

Figure 4.5: Writing the Collaborative Statements

- The user has to specify Attributive Statements (AS) about system, which he wants to build (Figure 4.6).

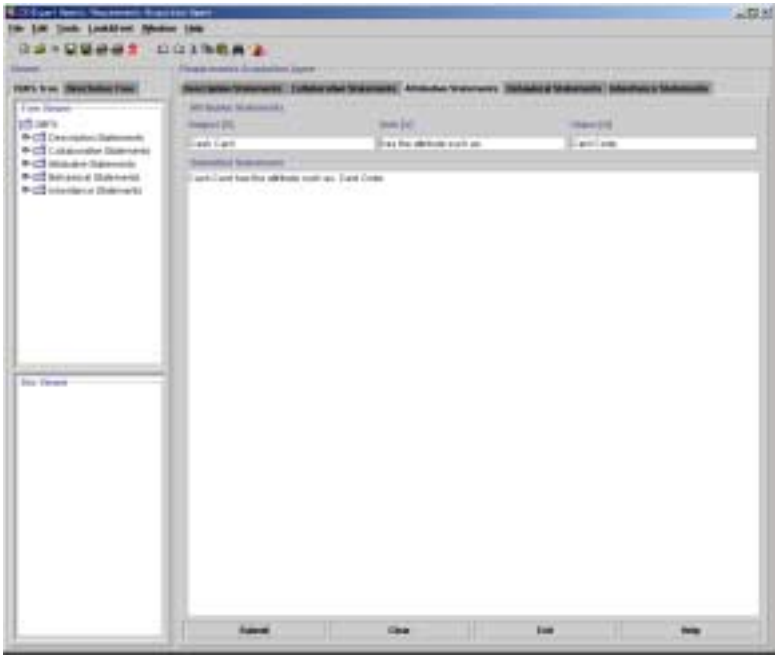

Figure 4.6: Writing the Attributive Statements

- The user has to specify Behavioral Statements (BS) about system, which he wants to build (Figure 4.7).

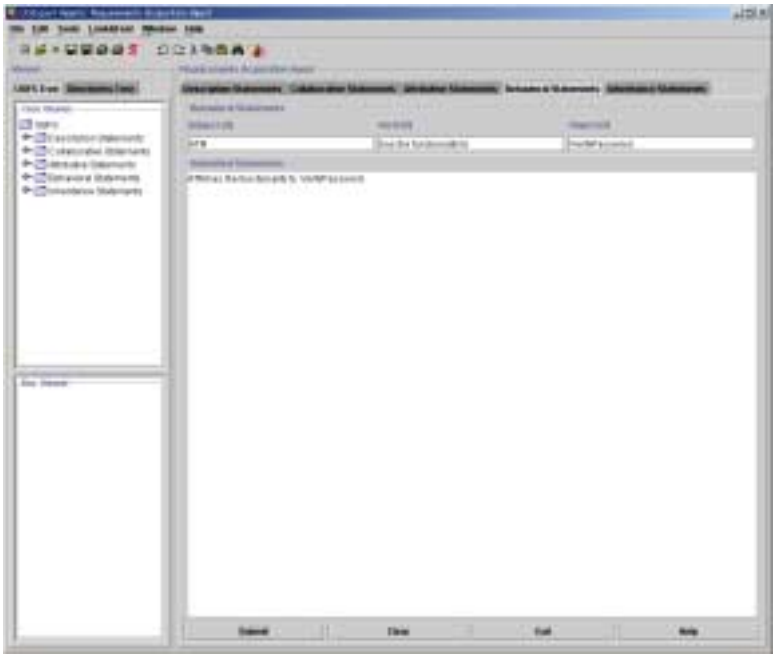

Figure 4.7: Writing the Behavioral Statements

- The user has to specify Inheritance Statements (IS) about system, which he wants to build (Figure 4.8).

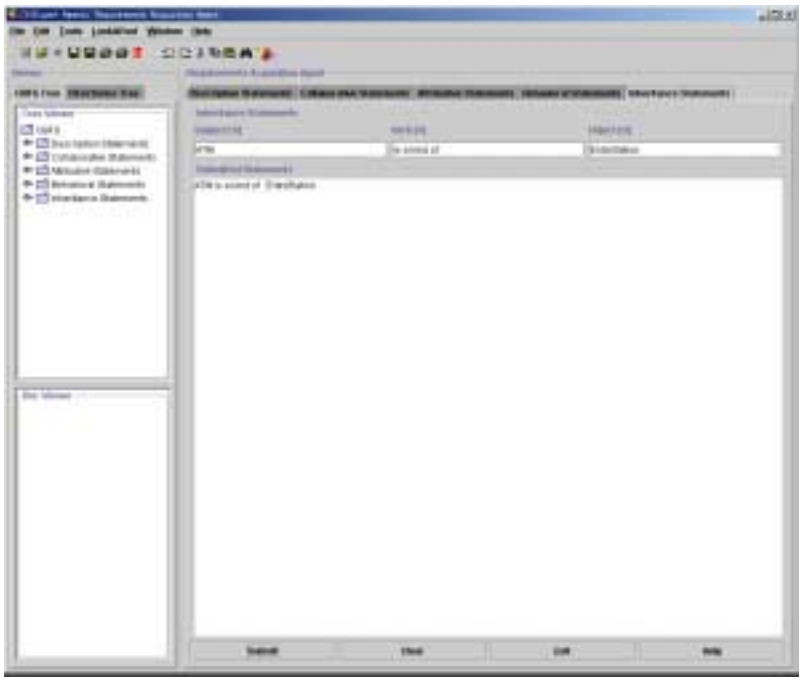

Figure 4.8: Writing the Inheritance Statements

STEP 2: Object Identification Process (Object Identification Agent)

The object identification agent manages the task concerning the object identification.

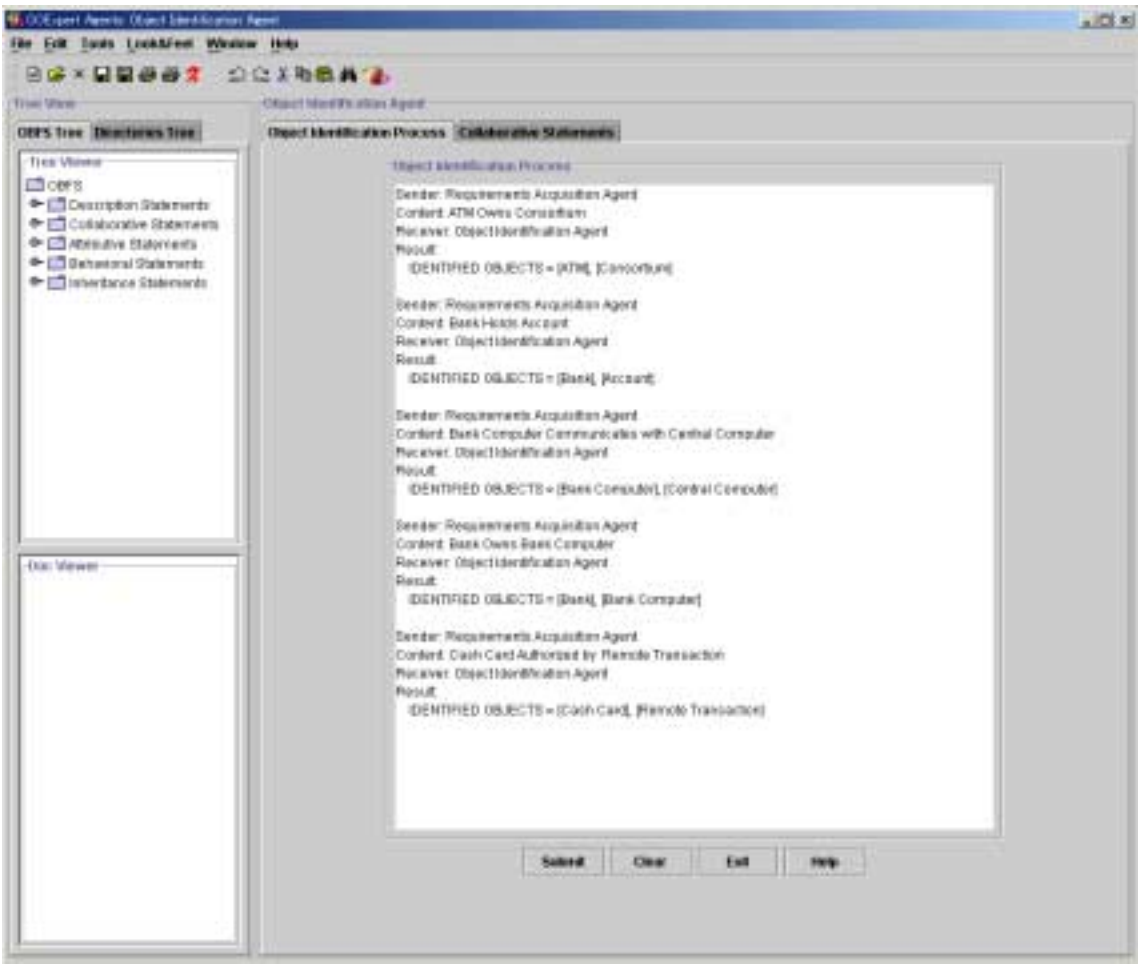

Figure 4.9: Object Identification Process

STEP 3: Association Identification Process (Association Identification Agent)

The association identification agent manages the task concerning the identification of associations between the identified objects.

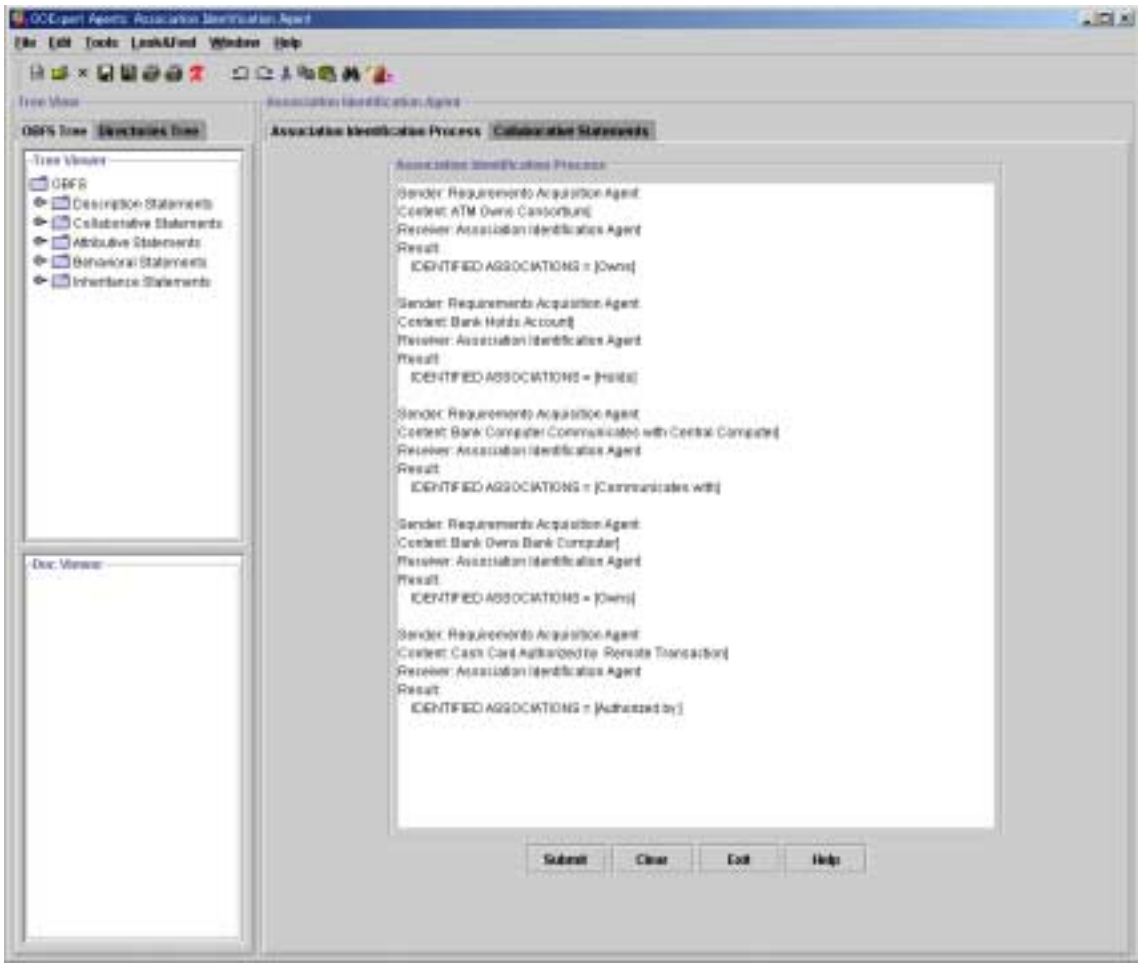

Figure 4.10: Association Identification Process

STEP 4: Attribute Identification Process (Attribute Identification Agent)

The attribute identification agent manages the task concerning the identification of object attributes.

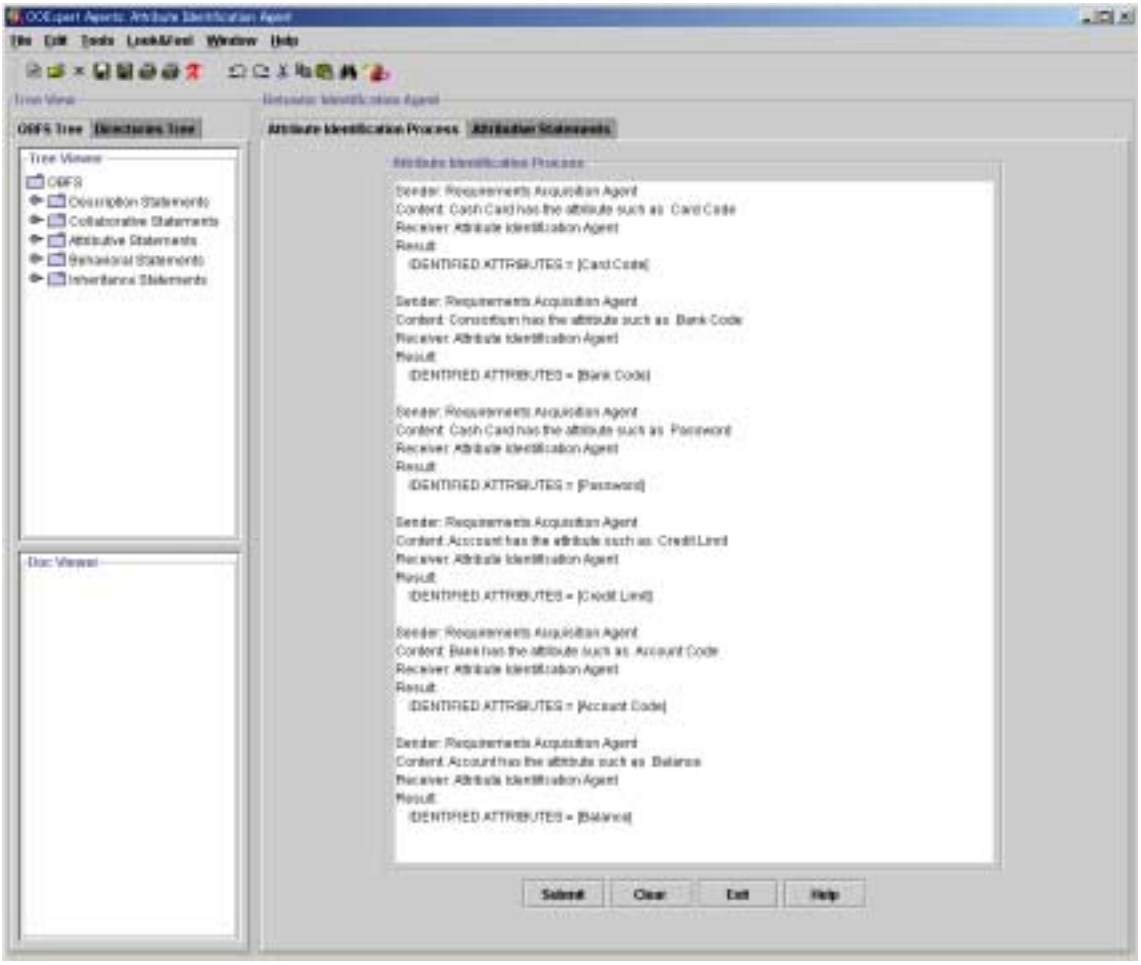

Figure 4.11: Attribute Identification Process

STEP 5: Behavior Identification Process (Behavior Identification Agent)

The behavior identification agent manages the task concerning the identification of object behaviors.

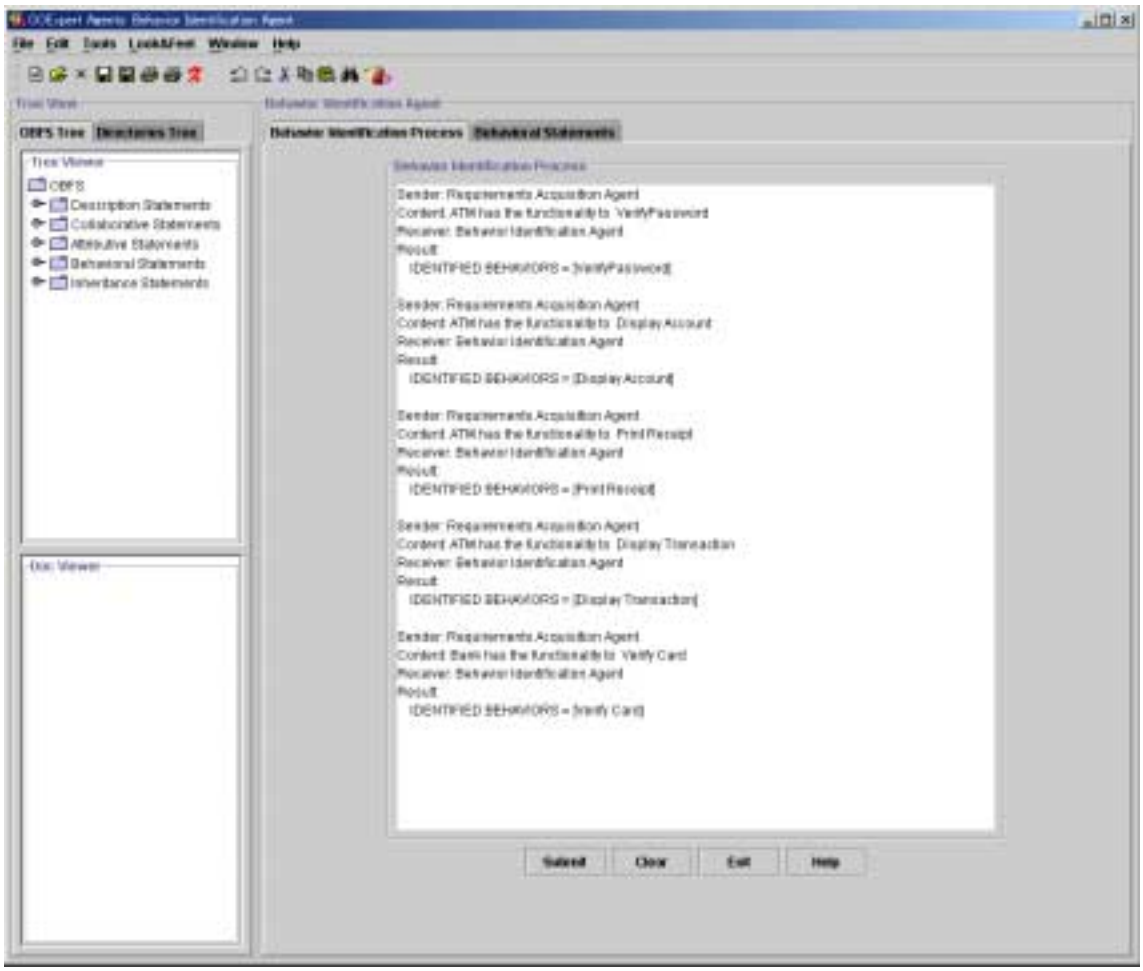

Figure 4.12: Behavior Identification Process

STEP 6: Object Refinement Process (Object Refinement Agent)

The object refinement agent manages the task concerning to refine objects and organize classes by using inheritance to share common structure.

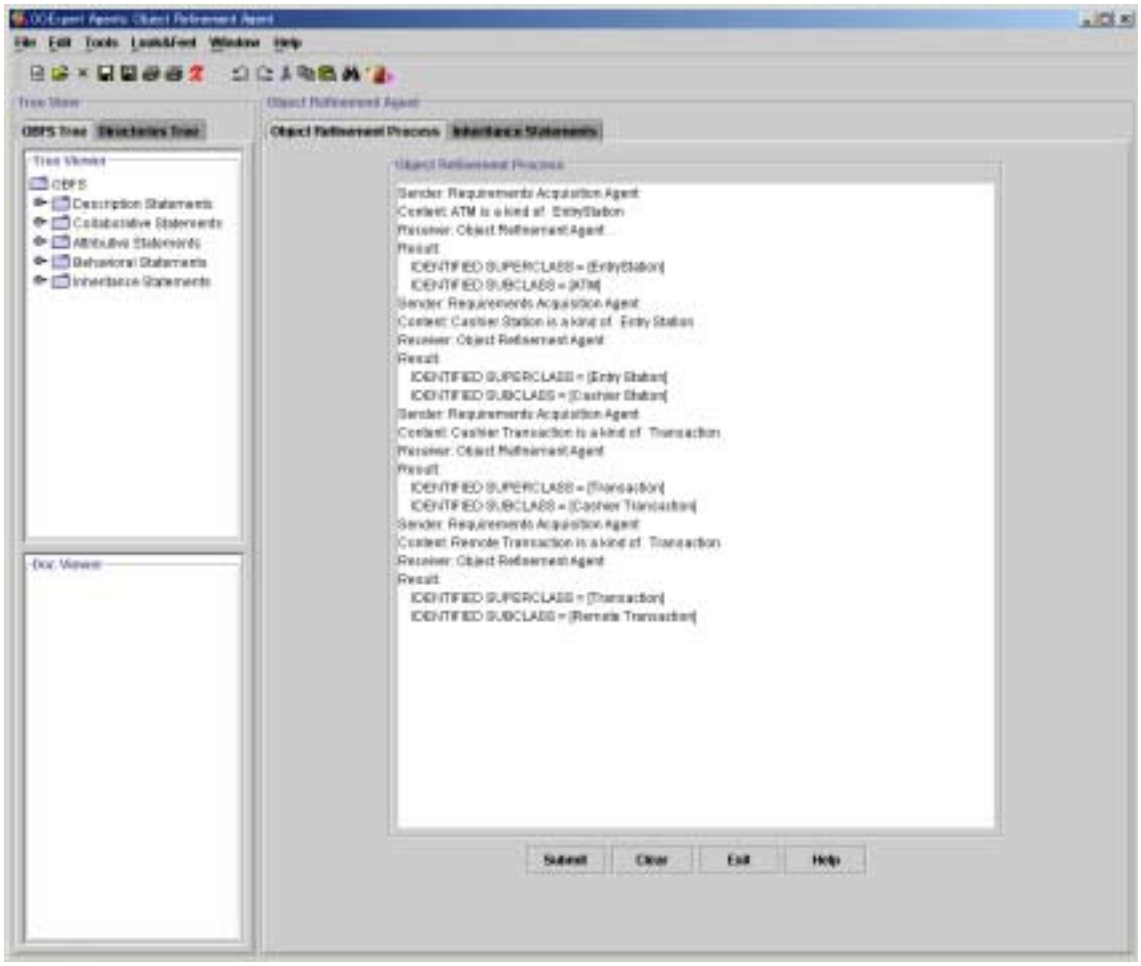

Figure 4.13: Object Identification Process

4.2.3 Summary of How the *OOExpert* Works

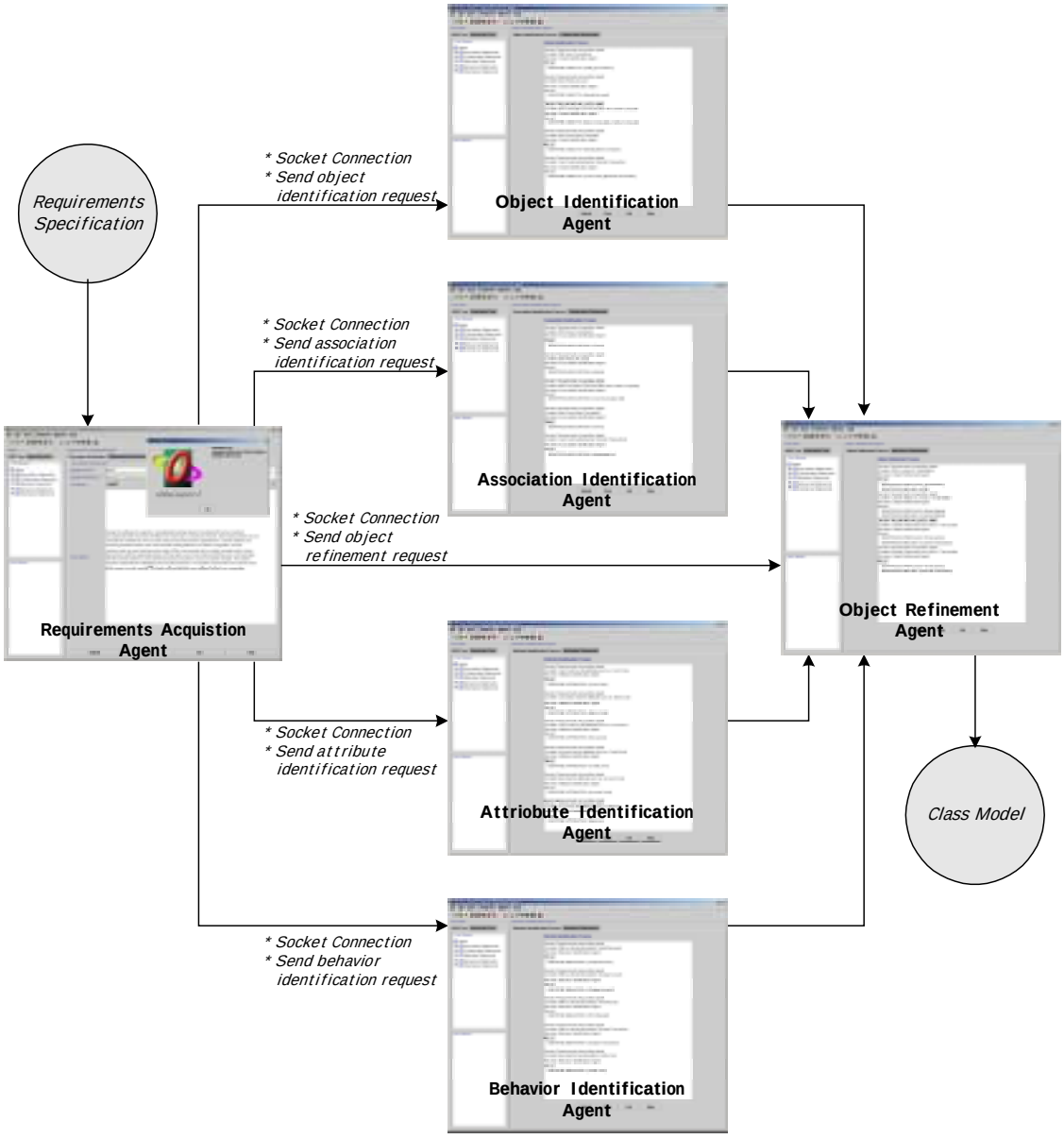

Figure 4.14: Summary of How the *OOExpert* Works

Chapter 5
Conclusion

At this point we have described and addressed the problem of object model creation process in object-oriented analysis and design. Furthermore, We have defined and formalized our approach to overcome above problems. We also have designed and implemented our idea to be a software system, that we called it *OOExpert*. The final step will be to summarize the argument presented in this thesis and reflect on it.

5.1 System Evaluation and Future Directions

In compare with the other object-oriented CASE (Computer Aided Software Engineering) systems, our system has the potential of handling and solving problems on object model creation process in object-oriented analysis and design. However, object-oriented CASE systems which exist now, like *Rational Rose* (www.rational.com), *Together* (www.togethersoft.com), *Object Domain* (www.objectdomain.com), etc. have concentrated on the problem solving of the object modeling notation and forward/engineering engineering too much, but the problem on the previous phase, that is an object model creation process phase, has not been solved yet. Our works concentrate on how we can handle and solve the problems on object model creation process in object-oriented analysis and design.

The contribution of this thesis has been a proposal for solving problems described in the chapter 2. Much more work is required to develop satisfied *OOExpert* agents according to the computational model proposed in this thesis. In particular, the following limitations need to be addressed:

- Autonomous rule acquisition and its administration
- Learning capabilities for capturing the solution of human designer
- Friendly user interface for negotiating and determining the best object-oriented design between designer and *OOExpert*
- Forward and reverse engineering for coding

The proposal for future works are derived from the limitations of the work presented above and is based on addressing some of the additional issues identified in this thesis. In particular, the future works are categorized into extensions to the:

- Knowledge management in the *OOExpert* agents
- Implementation strategy for indexing two kinds of case-base: *Human Expert Solution* (*HES*) and *Problem Domain Relation* (*PDR*)
- Negotiation and coordination strategy among *OOExpert* agents
- The functionalities for doing forward and reverse engineering

5.2 Summary and Conclusion

The challenges of object-oriented analysis and design are, to identify the objects and their attributes needed to implement the software, describe the associations between the identified objects, define the behavior of the objects by describing the function implementations of each object, and refine objects and organize classes by using inheritance to share common structure **[Beringer, 1997] [Booch, 1991] [Holland et al., 1996] [Liang et al., 1998]**. The object identification and refinement process are very important process in object-oriented analysis and design, and we called this process by *object model creation process*. Researchers and software designers come to a conclusion that object model creation process is an ill-defined task, regarding of the difficulties of heuristic **[Holland et al., 1996] [Kato, 1998]** and there is no unified methodology for object-oriented software analysis and design. This is mainly due to lack of formalism for object-oriented software analysis and design.

In our project, we are developing an intelligent agents system that aims to help designers while designing object-oriented software by automating the difficulties and ill-defined tasks in the object model creation process, including identification of objects, associations, attributes, behaviors, and organization of objects with inheritance. First of all, we propose formal models of the object model creation process. And then we formulate design patterns and rules for solving above problems, and store them in the agent's knowledge bases.

A summary of this thesis follows:
In chapter 1, we give a brief introduction and overview to the two major topics covered in this thesis: object-oriented analysis and design, and intelligent agents. It starts with a short introduction of the object-oriented paradigm, as frameworks rely heavily on its mechanisms such as object, class, inheritance, polymorphism and so on. And then, we will take a look at object-oriented analysis and design, and its problem that motivate us to do research on this topic. We present the key attributes of intelligent agents such as autonomy, mobility, and intelligence, and also provide the benefits and taxonomy of various intelligent agents technology. The research motivations and objectives are also presented at the end of this chapter.

In chapter 2, we focus on object model creation process and why it has the capacity to play a key role in object-oriented analysis and design. However, building software

engineering tools, and defining repository requires quantitative approach, because everything must be clear and unambiguous. One way to ensure clarity of ideas is through mathematical formalism. This chapter is an initial attempt to produce such formalism for object model creation process used to represent the result of our works. It presents a basic ontology for expressing our concepts and their relationships using set of theory and functions. In this chapter, we explain our concepts, idea and approach toward well-defined object model creation process and its computational model.

In chapter 3, we focus on how the problems on object model creation process introduced and formalized in the previous section can be designed to be a software system. In our research, object model creation process is viewed as a society of software agents that interact and negotiate with each other. We also construct the *OOExpert* agent framework so that inter-agent communication can be supported as well as the mobility of our agents across network. Finally, we explain system design and architecture of each *OOExpert* agent, including requirements acquisition agent, object identification agent, attribute identification agent, association identification agent, behavior identification, and object refinement agent.

In chapter 4, we focus on how the problems on object model creation process introduced, formalized, and designed in the previous section can be implemented to be a software system. It starts with an explanation about why Java is used as programming language to implement *OOExpert* agents. However, There are specific features of Java, which support intelligent agent paradigm: autonomy, intelligence and mobility. How the *OOExpert* agents work is also presented at the end of this chapter.

Acknowledgements

I want to thank many people who made this thesis possible. This thesis could not have been written without help of a great many people who assisted me with their comments, their criticisms and above all their support.

First, I am particularly grateful to Professor Behrouz H. Far for being an ideal supervisor, and his tact and encouragement assisted me in maintaining my confidence in this research project. I am also very grateful for Professor Zenya Koono for his invaluable support and guidance. I am happy to express my gratitude here. My thanks also go to Hajji Hassan, Shadan Shanipour, Ewin Mardhana, Hiroyuki Onjo and Sumihiko Gouda for providing me with constructive comments and criticisms when the thesis existed only in embryonic form. I found their views very helpful. They also gave me friendship, their help and their encouragement, for which I am profoundly grateful.

I am also grateful to the members of the Koono Laboratory and many others too numerous to mention here. Their kindnesses, and the particularly fruitful scientific discussions we had together, were a great help to me.

Most of all, I want to thank the scores or even hundreds of persons who contributed to this thesis, who contributed to the community of ideas in intelligent agent, object-orientated analysis and design methodology, software engineering, and numerous other areas of computer science. It is impossible to list them all, or indeed to track even the major chains of influence, without a major scholarly effort, and this is an

engineering thesis, not a historical review. Many are well known, but many good ideas came from those who did not have the good fortune to be widely recognized.

Finally, without the patience of my wife, Wulan, and my sons, Irsyad and Hasan, there would have been no thesis about my research. And I also would like to thank all my companions, my family and friends, for their understanding, their unconditional support and their affection over the past few years.

Bibliography

[Anumba et al., 1997] C.J. Anumba and N.F.O. Evbuomwan, "Concurrent Engineering in Design-Build Projects", *Construction Management and Economics*, Vol. 15, No. 3, May, pp 271-281, 1997.

[Aoyama, 1992] 青山幹雄, “分散環境：新しい開発環境像を求めて”, *情報処理*, Vol.33 No.1, 1992.

[Bailin, 2000] Sidney C. Bailin, “Object-Oriented Requirements Analysis”, *Software Requirements Engineering*, pp. 334-355, IEEE Computer Society Press, 2000.

[Bates, 1994] J. Bates, "The Role of Emotion in Believable Agents", *Communications of the ACM*, Vol. 37, No. 7, pp. 122-125, July 1994.

[Belo et al., 1996] Orlando Belo and Antonio Ribeiro, “A Web-Based Framework for Distributed Expert Systems”, *Proceedings of WWW National Conference*, 1996.

[Beringer, 1997] Dorothea Beringer, “Modelling Global Behaviour with Scenarios in Object-Oriented Analysis”, *PhD*

Dissertation at the Software Engineering Laboratory, Swiss Federal Institute of Technology in Lausanne (EPFL), May 1997.

[Bigus et al., 1997] Joseph P. Bigus and Jennifer Bigus, "Constructing Intelligent Agents with Java: A Programmer's Guide to Smarter Applications", *John Wiley & Sons, Inc*, USA, 1997.

[Booch, 1991] Grady Booch, "Object-Oriented Analysis and Design with Application", *Benjamin/Cummings*, 1991.

[Booch et al., 1999] Grady Booch, James Rumbaugh, and Ivar Jacobson, "The Unified Modeling Language User Guide", *Addison-Wesley*, 1999.

[Caglayan et al., 1997] A. Caglayan, Colin Harrison, Alper Caglayan, and Colin G. Harrison, "Agent Sourcebook: A Complete Guide to Desktop, Internet, and Intranet Agents", *John Wiley & Sons Inc.*, January 1997.

[Cercone et al., 1999] Nick Cercone, Aijun An, and Christine Chan, "Rule-Induction and Case-Based Reasoning: Hybrid Architectures Appear Advantageous", *IEEE Transactions on Knowledge Engineering and Data Engineering*, Vol. 11, No.1, January/February 1999.

[Chaves et al., 1996] A. Chaves and P. Maes, "Kasbah: An Agent Marketplace for buying and selling goods", *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM-96)*, pp. 75-90, 1996.

[Clark et al., 1999] Robert G. Clark and Ana M.D. Moreira, "Formal

Specifications of User Requirements", *Automated Software Engineering*, Vol. 6, No. 3, 1999, Kluwer Academic Publishers.

[Coad et al., 1991] Peter Coad and Edward Yourdon, "Object-Oriented Analysis", *Yourdon Press*, 1991.

[Decker et al., 1994] Rick Decker and Stuart Hirshfield, "The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught in CS 1", *Selected Papers of the Twenty-Fifth Annual SIGCSE Symposium on Computer Science Education*, pp. 51-55, 1994.

[Dorfman, 2000] Merlin Dorfman, "Requirements Engineering", *Software Requirements Engineering*, pp. 7-22, IEEE Computer Society Press, 2000.

[Drucker, 1998] Drucker P., "The Coming of the New Organization", *Harvard Business Review*, Jan.-Feb. 1998, pp. 45-53, 1988.

[Far et al., 1994] B.H. Far, Takeshi Takizawa, and Zenya Koono, "An Expert System for Reproducing Huan Cognitive Processes in Automatic Software Design", *World Congress of Expert System*, Portugal, 1994.

[Far et al., 1996] B.H. Far and Zenya Koono, "Ex-W-Pert System: A Web-Based Distributed Expert System for Groupware Design", *World Congress on Expert Systems' 96*, pp. 545-552, Seoul, Korea, February 4-9, 1996.

[Far et al., 1997] B.H. Far, H. Mukai, Zenya Koono, "Intelligent Agents for Electronic Commerce", *Proceedings of The 11st Annual Conference of Japanese Society for Artificial Intelligence*, 1997.

[Far et al., 1997] B.H. Far, Hidenari Mukai, and Zenya Koono, “Intelligent Agents for Electronic Commerce”, *Proceedings of the 11st Annual Conference of Japanese Society for Artificial Intelligence*, pp. 482-485, Tokyo, Japan, June 1997.

[Far et al., 1998] B.H. Far, S.O. Soueina, H. Hajji, S. Saniepour, A.H. Hashimoto, “An Integrated Reasoning and Learning Environment for WWW Based Software Agents for Electronic Commerce”, *IEICE Transaction on Information and System*, Vol. E81-D, No.12, pp.1374-1386, December 1998.

[Faulk, 2000] Stuart R. Faulk, “Software Requirements: A Tutorial”, Software Requirements Engineering, pp. 158-179, *IEEE Computer Society Press*, 2000.

[Ferber, 1999] Jacques Ferber, “Multi-Agent Systems An Introduction to Distributed Artificial Intelligence”, *Addison-Wesley*, 1999.

[Finin et al., 1993] Tim Finin, Jay Weber, Gio Wiederhold, Michael Geneseret, Richard Frtitzson, James McGuire, Stuart Shapiro and Chris Beck, “DRAFT Specification of the KQML Agent-Communication Language -- plus example agent policies and architectures”, *The DARPA Knowledge Sharing Initiative*, 1993.

[Finin et al., 1994] Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire, “KQML: An Information and Knowledge Exchange Protocol”, *Knowledge Building and Knowledge Sharing*, Ohmsha and IOS Press, 1994.

[Finin et al., 1994] Tim Finin, Richard Fritzson Don McKay and Robin McEntire, "KQML as an Agent Communication Language", *The Proceedings of the Third*

International Conference on Information and Knowledge Management (CIKM'94), ACM Press, November 1994.

[Finin et al., 1997] Tim Finin, Yannis Labrou, and James Mayfield, “KQML as an Agent Communication Language”, *Software Agents*, MIT Press, Cambridge, 1997.

[Genesereth et al., 1992] Michael R. Genesereth and Richard E. Fikes, "Knowledge Interchange Format Version 3.0 Reference Manual", *Technical Report Logic-92-1*, Stanford University, 1992.

[Goguen et al., 1993] Joseph A Goguen and Charlote Linde, “Techniques for Requirements Elicitation”, *Proceedings of International Symposium on Requirements Engineering*, pp. 152-164, 1993.

[Grand et al., 1998] S. Grand and D. Cliff, "Creatures: Entertainment Software Agents With Artificial Life", *Autonomous Agents and Multi-Agent Systems*, Vol.1 No.1, 1998.

[Griffeth et al., 1994] N.D. Griffeth and H. Velthuijsen, "The Negotiating Agents Approach to Run-Time Feature Interaction RResolution", *Feature Interactions in Telecommunications System*, pp.217-235, IOS Press, 1994.

[Gruber, 1993] T. R. Gruber, “A Translation Approach to Portable Ontologies”, *Knowledge Acquisition*, 5(2), pp.199-220, 1993.

[Halladay et al., 1993] Steve Halladay, Michael Wiebel, “Object-Oriented Software Engineering”, *R&D Publications*, 1993

[Harabayashi et al., 1993] Toshiyuki Harabayashi, Atsuo Kawai, and Tsutomu

Shiino, “Software Requirment Specification in Controled Natural Language and its Analysis”, *Software Engineering*, 95-3, 1993.

[Harwell et al., 1993] Richard Harwell, Erik Aslaksen, Ivy Hooks, Roy Mengot, and Ken Ptack, “What is Requirement?”, *Proceedings of the Third Annual International Symposium*, pp. 17-24, National Council on System Engineering, 1993.

[Hayes et al., 1989] B. Hayes, M. Hewett, R. Washington R. Hewett, and A. Seiver, "Distributing Intelligence Within an Individual", *Distributed Artificial Intelligence Volume II*, pp. 385-412, Pitman Publishing: London and Morgan Kaufman: Sun Mateo, 1989.

[Heisel et al., 1998] Maritta Heisel and Jeanine Souquieres, “Methodological Support for Requirements Elicitation and Formal Specification”, *Proceedings of the 9th International Workshop on Software Specification and Design*, Ise-Shima (Isobe), Japan, April 16-18, 1998.

[Henderson et al., 1992] Brian Henderson and Sellers, “A Book of Object-Oriented Knowledge – Object-Oriented Analysis Design and Implementation: A New Approach to Software Engineering”, *Prentice Hall*, 1992.

[Holland et al., 1996] Ian M. Holland and Karl J. Lieberherr, “Object-Oriented Design”, *ACM Computing Surveys*, Vol. 28, No. 1, March 1996.

[Honiden et al., 1994] 本位田真一，山城明宏，“オブジェクト指向分析・設計”，*情報処理*, Vol.35 No.5, May 1994.

[Huang et al., 1995] J. Huang, N.R. Jennings, and J. Fox, "An Agent-Based Approach to Health Care Management", *Applied Artificial Intelligence*, Vol. 9, No. 4, pp.401-420, 1995.

[IEEE Std 1233, 1998] Software Engineering Standards Committee of the IEEE Computer Society, "IEEE Guide for Developing System Requirements Specifications", *IEEE Std 1233-1998*, IEEE, New York, 1998.

[IEEE Std 830, 1998] Software Engineering Standards Committee of the IEEE Computer Society, "IEEE Recommended Practice for Software Requirements Specifications", *IEEE Std 830-1998*, IEEE, New York, 1998.

[Iglewski et al., 1997] Michal Iglewski and Tomasz Muldner, "Comparison of Formal Specification Methods and Object-Oriented Paradigms", *Journal of Network and Computer Applications*, Vol. 20, No. 4, 1997, Academic Press.

[Ishida et al., 1992] 石田亨, 桑原和広, "分散人工知能（1）：協調問題解決", *人工知能学会誌*, Vol.7 No.6, 1992.

[Jackson et al., 1995] Michael Jackson and Pamela Zave, "Deriving Specifications from Requirements: an Example", *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA USA, April 24 - 28, 1995.

[Jacobson et al., 1992] Ivar Jacobson, Magnus Christerson, Patrik Jonson, and Gunnar Overgaard, "Object-Oriented Software Engineering: A Use Case Driven Approach", *Addison-Wesley*, 1992.

[Jacobson et al., 1999] Ivar Jacobson, Grady Booch, and James Rumbaugh,

“The Unified Software Development Process”, *Addison-Wesley*, 1999.

[Jalote, 1997] P. Jalote, “An Integrated Approach to Software Engineering”, *Springler-Verlag*, New York, 1997.

[Jennings et al., 1993] N.R. Jennings, L.Z. Varga, R.P.Aarnts, J.Fuchs、P. Skarek, “Transforming Standalone Expert Systems into a Community of Cooperating Agents”, *Eng.Applic.Artificial Intelligent*, Vol.6 No.4, 1993.

[Jennings et al., Dec 1996] N.R. Jennings, J. Corera, I. Laresgoiti, E.H. Mamdani, F. Perriolat, P. Skarek, and L.Z. Varga, "Using ARCHON to Develop Real-World DAI Applications for Electricity Transportation Management and Particle Acceleration Control", *IEEE Expert*, Vol. 11, No. 6, pp. 60-88, December 1996.

[Jennings et al., 1996] N.R. Jennings, P. Faratin, M.J. Johnson, T.J. Norman. P. O'Brien, and M.E. Wiegand, "Agent-Based Business Process Management", *International Journal of Cooperative Information Systems*, Vol. 5, No. 2-3, pp. 105-130, 1996.

[Jennings et al., 1998] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge, "A Roadmap of Agent Research and Development", *Autonomous Agents and Multi-Agent Systems*, pp. 7-38, Kluwer Academic Publishers, Boston, 1998.

[Kato, 1998] 加藤貞行, “オブジェクト識別についての一考察とその効果”, *情報処理学会研究報告*, Vol.90, No.100,1998.

[Kawamura et al., 1993] 河村一樹、 “ソフトウェア工学入門”、 *啓学出*

版、1993.

[Kolodner, 1993] J.Kolodner, “Case-Based Reasoning”, *Morgan Kaufman*, San Francisco, 1993.

[Komiya, 1998] Seiji Komiya, “A Model for Recording Software Design Decisions and Design Rationale”, *IEICE Transaction on Information and Systems*, Vol.E81-D No.12, pp.1350-1363, December 1998.

[Koono et al., 1994] Z. Koono, B.H. Far, T. Sugimoto, M. Ohmori and Hui Chen, “A Systematic Approach for Design Knowledge Acquisition from Documents”, *Proceedings of The Third Japanese Knowledge Acquisition for Knowledge-Based Systems Workshop*, 1994.

[Koono et al., 1997] 河野 善彌, 陳 慧, B.H. ファー, “人の設計知識の構造とソフトウェア工学”, *情報処理学会ソフトウェア工学研究会 97-SE-114*, pp. 33-40, May 1997.

[Labrou et al., 1994] Yannis Labrou and Tim Finin, “A semantics approach for KQML - A General Purpose Communication Language For Software Agents”, *Third International Conference on Information and Knowledge Management (CIKM'94)*, November 1994.

[Labrou et al., 1997] Yannis Labrou and Tim Finin, "A Proposal for a new KQML Specification", *TR CS-97-03*, February 1997.

[Lester, 1997] J.C Lester and B.A Stone, "Increasing Believability in Animated Pedagogical Agents", *Proceedings of the First International Conference on Autonomous Agents (Agents 97)*, pp. 16-21, 1997.

[Liang et al., 1993] Y. Liang, M.A. Newton, and H.M. Robinson, "Analysis of Information Systems using Object-Oriented Methodologies", *Proceeding of BCS ISM SG and BSS Joint Conference on the Theory, Use and Integrative Aspects of IS Methodologies*, pp. 55-70, 1993.

[Liang et al., 1998] Ying Liang, Daune West, and Frank A. Stowell, “An Approach to Object Identification, Selection and Specification in Object-Oriented Analysis”, *Information Systems Journal*, Vol. 8, No. 2, 1998, pp. 163-180, Blackwell Science Ltd., 1998.

[Ljunberg et al., 1992] M. Ljunberg and A. Lucas, "The OASIS Air Traffic Management System", *Proceedings of the Second Pacific Rim International Conference on AI (PRICA-92)*, Seoul, Korea, 1992.

[Maes, 1994] P. Maes, "Agents that reduce work and information overload", *Communication of the ACM*, Vol. 37, No.7, pp. 31-40, July 1994.

[Martin et al., 1995] James Martin and James J. Odell, “Object-Oriented Methods: A Foundation”, *Prentice Hall*, 1995.

[Martin et al., 1996] James Martin and James J. Odell, “Object-Oriented Methods: Pragmatic Considerations”, *Prentice Hall*, 1996.

[Mayfield et al., 1996] James Mayfield, Yannis Labrou, and Tim Finin, "Evaluation of KQML as an Agent Communication Language", *Intelligent Agents Volume II -- Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag, 1996.

[Miyazaki et al., 1998] 宮崎善史、廣田豊彦、橋本正明, “自分自身を編集出来るオブジェクトモデルエディタ”, *情報処理学会研究報告*, Vol.90 No.100, 1998.

[Nishida, 1994] 西田 豊明, “大規模知識ベースシステム”, *情報処理*, Vol.35 No.2,1994.

[Nishida, 1995] 西田 豊明, “ソフトウエアエージェント”, *人工知能学会誌*, Vol.10 No.5,1995.

[Odell et al., 1997] James Odell and Guus Ramackers, “Toward a Formalization of OO Analysis”, Journal of OO Programming, pp. 64-68, July 1997.

[Orfali et al., 1997] Robert Orfali and Dan Harkey, “Client/Server Programming with Java and CORBA”, *John Wiley & Sons Inc.*,1997.

[Parsons et al., 1999] Michael G. Parsons, David J. Singer, and John A. Sauter, "A Hybrid Agent Approach For Set-Based Conceptual Ship Design", *Proceedings of the International Conference on Computer Applications in Shipbuilding*, Cambrige, June 1999.

[Parunak, 1987] H. Van Dyke Parunak, "Manufacturing Experience with the Contract Net", Distributed Artificial Intelligence, *Pitman Publishing: London and Morgan Kaufmann: San Mateo*, pp.285-310, 1987.

[Petrie et al., 1999] Charles Petrie, Sigrid Goldmann, and Andreas Raquet, "Agent-Based Project Management", *Lecture Notes in AI - 1600*, Springer-Verlag, 1999.

[Ralston et al., 1993] Anthony Ralston, Edwin D. Reilly, “Encyclopedia of Computer Science”, *Van Nonstrand Reinhold*, IEEE Press, 1993.

[Rasmussen, 1985] J. Rasmussen, “The Role of Hierarchical Knowledge Representation in Decision Making and System Management”, *IEEE Transaction on System, Man, and Cybernetics*, Vol. SMC-15, No. 2, pp.234-243, March/April 1985.

[Rentsch, 1982] T. Rentsch, “Object Oriented Programming”; *SIGPLAN Notices*; Vol.17 No.12; pp.51, September 1982.

[Romi, 1999] Romi Satria Wahono, “Distributed Knowledge Based System for Automatic Object-Oriented Software Design: System Design”, (オブジェクト指向ソフトウェア自動設計用分散型知識ベースシステムの開発：システム設計), *B.Eng. Dissertation at the Department of Information and Computer Sciences, Faculty of Engineering*, Saitama University, Saitama, Japan, February 1999.

[Romi et al., March 1999] Romi Satria Wahono and B.H. Far, “Distributed Expert System Architecture for Automatic Object-Oriented Software Design”, *Proceedings of the Third Workshop on Electro-Communication and Information (WECI-III)*, pp. 131-134, Japan, March 1999.

[Romi et al, June 1999] Romi Satria Wahono and B.H. Far, “OOExpert: Distributed Expert System for Automatic Object-Oriented Software Design”, *Proceedings of the 13th Annual Conference of Japanese Society for Artificial Intelligence*, pp.456-457, Tokyo, Japan, June 1999.

[Romi et al., March 2000] Romi Satria Wahono and B.H. Far, “Reasoning with Cases in the CBR System: A Case Study for Applying OOExpert System”, *Proceedings of the*

IECI Japan Workshop 2000 (IJW-2000), pp. 89-93, Japan, March 2000.

[Romi et al., July 2000] Romi Satria Wahono and B.H. Far, "Hybrid Reasoning Architecture for Solving Object Class Identification Problem in the OOExpert System", *Proceedings of the 14th Annual Conference of Japanese Society for Artificial Intelligence*, Tokyo, Japan, July, 2000.

[Rosenberger., 1998] Jeremy Rosenberger, "Teach Yourself CORBA in 14 Days", *Sams Publishing*, USA, 1998.

[Rumbaugh et al., 1991] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson, "Object-Oriented Modeling and Design", *Prentice Hall*, 1991.

[Rumbaugh et al., 1999] James Rumbaugh, Ivar Jacobson, and Grady Booch, "The Unified Modeling Language Reference Manual", *Addison-Wesley*, 1999.

[Saeki et al., 1989] Motoshi Saeki, Hisayuki Horai and Hajime Enomoto, "Software Development Process from Natural Language Specification", *Proceedings of the 11th International Conference on Software Engineering*, pp.64-73, 1989.

[Saiedian, 2000] Hossein Saiedian, "Formal Methods in Information System Engineering", *Software Requirements Engineering*, pp. 384-397, IEEE Computer Society Press, 2000.

[Sakashita, 1992] 坂下善彦, "分散開発環境の事例と今後の展望", *情報処理*, Vol.33 No.1, January 1992.

[Shapiro et al., 1992] Stuart C. Shapiro, “Encyclopedia of Artificial Intelligence Second Edition Volume 2”, *John Wiley & Sons, Inc.*, 1992.

[Shlaer et al., 1988] Sally Shlaer and Stephen J. Mellor, “Object-Oriented System Analysis: Modeling the World in Data”, *Yourdon Press*, 1988.

[Siddiqi et al, 1996] Jawed Siddiqi and M. Chandra Shekaran, “Requirements Engineering: The Emerging Wisdom”, *IEEE Software*, Vol. 13, No. 2, March 1996, pp. 15-19.

[Sugimoto et al., 2000] Hideaki Sugimoto, Atsushi Ohnishi, “A Supporting Method of Making a Consistent Software Requirements Specification Based on the Dempster and Shafer's Theory”, *IEICE Transaction on Information and Systems*, Vol.E83-D No.4 p.659-668, April 2000.

[Sycara et al., 1996] K.P. Sycara, K. Decker, A. Pannu, M. Williamson and D. Zeng, "Distributed Intelligent Agents", *IEEE Expert*, Vol.11, No. 6, 1996.

[Tao, 1995] Yonglei Tao, “Using Expert Systems To Understand Object-Oriented Behavior”, *The 26th SISCSE Technical Symposium on Computer Science Education*, 1995.

[Tarumi, 1992] 垂水幸, “グループウエアのソフトウエア開発への応用”, *情報処理*, Vol.33 No.1, January 1992.

[Thayer, 2000] Richard H. Thayer, “Software System Engineering: An Engineering Process”, Software Requirements Engineering, pp. 84-106, *IEEE Computer Society Press*, 2000.

[Trapl et al., 1997] R. Trapl and P. Petta, "Creating Personalities for Syntentic Actors", *Springler-Verlag*, 1997.

[Tsatsoulis et al., 1997] Costas Tsatsoulis, Qing Cheng, and Hsin-Yen Wei, “Integrating Case-Based Reasoning and Decision Theory”, *IEEE Expert*, Vol. 12, No. 4, pp. 46-55, July/August 1997.

[Uetake et al., 1998] Tomofumi Uetake and Morio Nagata, “A Support Tool for Specifying Requirements Using Structures of Documents”, *IEICE Transaction on Information and Systems*, Vol.E81-D No.12, pp.1429 - 1438, December 1998.

[Vienneau, 1993] Robert Vienneau, “A Review of Formal Methods”, *Kaman Science Corrporation*, pp. 3-15 and 27-33, 1993.

[Wavish et al., 1996] P. Wavish and M. Graham, "A Situated Action Approach to Implementing Characters in Computer Games", *Applied Artificial Intelligence*, Vo. 10, No. 1, pp. 53-74, 1996.

[Weiss, 1999] Gerhard Weiss, "Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence", *The MIT Press*, 1999.

[Wirfs-Brock et al., 1990] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener, “Designing Object-Oriented Software”, *Prentice Hall*, 1990.

[Zave et al., 1997] Pamela Zave and Michael Jackson, “Four Dark Corners of Requirements Engineering”, *ACM Transaction on Software Engineering and Methodology*, Vol. 6, No. 1, pp. 1-30, Januari 1997.

List of Publications

[Romi et al., March 1999] **Romi Satria Wahono** and B.H. Far, "Distributed Expert System Architecture for Automatic Object-Oriented Software Design", *Proceedings of the Third Workshop on Electro-Communication and Information (WECI-III)*, pp. 131-134, Japan, March 1999.

[Romi et al, June 1999] **Romi Satria Wahono** and B.H. Far, "OOExpert: Distributed Expert System for Automatic Object-Oriented Software Design", *Proceedings of the 13th Annual Conference of Japanese Society for Artificial Intelligence*, pp.456-457, Tokyo, Japan, June 1999.

[Romi et al., March 2000] **Romi Satria Wahono** and B.H. Far, "Reasoning with Cases in the CBR System: A Case Study for Applying OOExpert System", *Proceedings of the IECI Japan Workshop 2000 (IJW-2000)*, pp. 89-93, Japan, March 2000.

[Romi et al., July 2000] **Romi Satria Wahono** and B.H. Far, "Hybrid Reasoning Architecture for Solving Object Class

Identification Problem in the OOExpert System", *Proceedings of the 14th Annual Conference of Japanese Society for Artificial Intelligence*, Tokyo, Japan, July, 2000.

[Gouda et al., 2001] Gouda Sumihiko, **Romi Satria Wahono**, and B.H. Far, "Design Pattern Usage Support System for Software Design", *Technical Report of IEICE*, KBSE2000 54-65, pp. 39-44, January 2001.

[Romi et al., 2001] **Romi Satria Wahono** and B.H. Far, "Towards the Use of Intelligence Agents in Collaborative Object-Oriented Analysis and Design", *Proceedings of the International Session of 15th Annual Conference of Japanese Society for Artificial Intelligence*, Matsue Japan, May, 2001. (To appear)

Glossary

Attributive Statements

Attributive statement (AS) is a requirement statement used to identify object attributes, which consists of subject (S), verb (V), and object (O) as well as the English (E) natural language.

Behavioral Statements

Behavioral Statement (BS) is a requirement statement used to identify object behaviors, which consists of subject (S), verb (V), and object (O) as well as the English (E) natural language.

Class

A class is a blueprint or prototype that defines the variables and methods common to all objects of a certain kind.

Collaborative Statements

Collaborative Statement (CS) is a requirement statement used to identify objects, and association between objects, which consists of subject (S), verb (V), and object (O) as well as the English (E) natural language.

Encapsulation

Encapsulation is the concept of the localization of knowledge within a module. Because objects encapsulate data and implementation, the user of an object can view the object as a black box that provides services. Instance variables and methods can be added, deleted, or changed, but as long as the services provided by the object remain the same, code that uses the object can continue to use it without being rewritten.

Description Statements

Description Statement (DS) is a requirement statement used to write an overview of the system that we want to build, which consists of Requirement ID, Requirement Name, Language, and Description.

Generalization and Specialization

Generalization and Specialization are relationships between concepts. Any type of A, each of whose objects is also an instance of a given type B, is called a specialization (or subtype) of B and is written as $A \subset B$. B is also called the generalization (or supertype) of A.

Inheritance

Inheritance is a mechanism for sharing attributes and behaviors among classes based on a hierarchical relationship.

Inheritance Statements

Inheritance Statement (IS) is a requirement statement used to organize classes by using inheritance, and to share common object attributes and behaviors, which consists of subject (S), verb (V), and object (O) as well as the English (E) natural language.

Object

Object is the principal building blocks of object-oriented programs. Each object is a programming unit consisting of attribute (instance variables) and behavior (instance methods). An object is a software bundle of variables and related methods.

Object-Based Formal Specification

Object-Based Formal Specification (OBFS) is a semi-formal requirements template used to reveal ambiguity, incompleteness, and inconsistency in an object-oriented software system, and to guide end users take an active role while describing their problem statements. OBFS is composed of Description Statements (DS), Collaborative Statements (CS), Attributive Statements (AS), Behavioral Statements (BS), and Inheritance Statements (IS).

Object Model Creation Process

Object model creation process is a main process of object-oriented analysis and design process, which starts with identification of objects, behaviors, attributes, and associations from requirements, and ends with object refinement with inheritance process.

Object-Oriented Analysis and Design

Object-Oriented Analysis and Design (OOAD) is a way of thinking about problems using models organized around real-world concepts.

Polymorphism

Polymorphism means that the same behavior may behave differently on different classes.