Proceedings of the

Second ASERC Workshop on

Software Architecture

February 18- 19, 2003

Banff Centre

Banff, Alberta, Canada

Sponsored by the Alberta Software Engineering Research Consortium (ASERC) and the Software Engineering Research Laboratory, University of Alberta (SERL)

An Agent-based System for Class Elicitation and Modeling in Object Oriented Analysis and Design

Dong Liu, Kalaivani Subramaniam, Behrouz Homayoun Far and Armin Eberlein

Abstract—**Object elicitation and class modeling is critical in Object Oriented Analysis and Design (OOAD). It is known to be a hard task for software engineers since there are no well-defined guidelines. In this paper, an approach based on the Object Model Creation Process (OMCP) is reviewed, and a methodology that supports the elicitation of objects and classes from requirements and the building of the class model is introduced. Use cases are used as the means to specify the stakeholders' requirements. Robustness analysis helps to bridge the gap between requirements and detailed design. This paper presents the analysis and design of an agent-based system that implements the methodology, as well as some details of the agent system architecture.**

Index Terms—**Object Oriented Analysis and Design (OOAD), Use case, Class modeling, Agent, Java Message Service (JMS).**

I. INTRODUCTION

Object Oriented (OO) development is currently the most popular software development methodology. Objects and classes are core concepts for Object Oriented Analysis and Design (OOAD). Finding the objects and classes, and then building the class model for a problem are among the central decisions in OO software development. Usually, there is no right class model or wrong one, but it is for sure that a bad model will have a detrimental effect on the software product. When the time to market for the software project is limited, the task of class elicitation and modeling needs to be finished within a certain time period using a reasonable methodology.

However, it seems that there are no theoretical or even pragmatically well-developed guidelines to help the software engineers tackle this problem successfully. In [1], Booch mentions that, at a conference on software engineering, several developers were asked what were the rules they applied to identify objects and classes. Stroustrup, the designer of C++, responded "It's a holy Grail. There is no panacea." Gabriel, one of the designers of CLOS, stated, "That's a fundamental question for which there is no easy answer. I try things." Obviously, this task has puzzled the developers since OO was first introduced. Early approaches to solve this problem focused on the analysis of nouns in the requirement documents, because the nouns are likely candidates for objects or classes in the analysis model. The Object Model Creation Process (OMCP) is a classic method to tackle this problem and is introduced in Section II of this paper. The Rational Unified Process (RUP) provides guidelines, templates and examples for most aspects and stages of iterative software developments. Use cases are means to capture the contracts between the stakeholders of a system and its behavior [2]. Use cases are recommended by RUP as the method to acquire requirements. In our approach, all the artifacts are specified using UML (Unified Modeling Language). We can use use case and other RUP methodologies and artifacts to leverage class elicitation and modeling. The methodology introduced in this paper can be regarded as tailored RUP.

The structure of this paper is as follows: Section II introduces the object elicitation methodology. Section III briefly discusses the components and technologies of an agent-based system architecture. In Section IV the analysis and design of the agent-based system for class elicitation is presented and the deviation between the agent-based analysis and design and traditional OOAD is pointed out. A conclusion follows in Section V.

II. OBJECT ELICITATION METHODOLOGY

A. Object Model Creation Process (OMCP)

The Object Model Creation Process (OMCP) is a widely applied method to elicit objects and build a class model from requirements [1] and a few systems that implement the OMCP are already available [3]. An overview of OMCP is shown in Fig. 1. The requirements are acquired from the stakeholder through interviews or other methods, and are documented in some requirements document. Then the engineers responsible for analysis and design try to find the objects based on the problem defined by the requirements and their own understanding of the domain. Attributes, behaviors, relationships and other essentials of the objects are identified. The next task is to set up and refine the class model based on the object model for the problem using generalization. When

Manuscript received December 16, 2002.

Dong Liu, Kalaivani Subramaniam, Behrouz Homayoun Far and Armin Eberlein are all with the Department of Electrical and Computer Engineering, University of Calgary, Calgary, Alberta, Canada T2N 1N4 (e-mail: {liud,ksubrama,far,eberlein}@enel.ucalgary.ca).

the class model is done, implementation can begin. As shown in Fig. 1, the resources that support the engineers' decisions are domain knowledge, and public and private experiences.

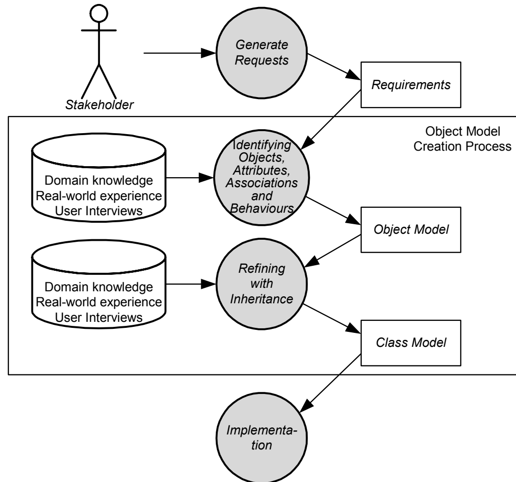

Fig. 1. Object Model Creation Process (OMCP).

B. Use Case

Use cases are widely used for requirements acquisition. They were introduced and applied to object oriented methodology very early [4]. Although there is some discredit for using use case in OO software process and finding classes from use cases [5], it is sure that use case are an efficient way to acquire requirement from the stakeholders, especially functional requirements. In UML, a use case is defined as "The specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or class can perform by interacting with outside actors." [6].

TABLE 1
TEMPLATE FOR USE CASE

1. Use Case Name
   1.1 Brief Description
2. Flow of Events
   2.1 Basic Flow
   2.2 Alternative Flows
      2.2.1 < First Alternative Flow >
      2.2.2 < Second Alternative Flow >
      …
3. Special Requirements
   3.1 < First special requirement >
   …
4. Preconditions
   4.1 < Precondition One >
   …
5. Postconditions
   5.1 < Postcondition One >
   …
6. Extension Points
   6.1 <name of extension point>
   …

UML supplies the notation for use case and use case diagram, but in Rational Rose or other computer aided software engineering (CASE) tools supporting UML visualization, the information supplied by the use case diagrams is not adequate. Therefore the use cases are accompanied by complementary documents, usually in textual format to be read by the engineers. Such text can be encoded in some way to be parsed by the CASE tool as well. There are some well-defined use case documentation templates in use [4]. Table 1 is the template used in our project. In the Flow of Events, what the actor does and what the system does in response are documented. The requirements other than functional requirements, which can be caught by system's behavior, are documented in the Special Requirements. Preconditions describe the states of the system that must be present prior to a use case being performed, and Postconditions describe possible states that the system can be in immediately after a use case has finished. Extension points document a location or set of locations within the behavioral sequence for a use case, at which additional behavior can be inserted.

C. Robustness Analysis

When UML is used to record all the artifacts during OOAD, some entities will appear repeatedly from requirements to detailed design: a use case diagram is for describing the system's behaviors; in use case realization, sequence diagrams or collaboration diagrams will be introduced; and finally classes and their attributes and behavior, the associations, aggregations, role names, multiplicities, navigations and inheritances are documented in the class diagram.

When a sequence diagram is drawn, the actor, the objects and the interactions between them are supposed to be known. But the objects need to be identified first from the use cases. That is why robustness diagrams are required to bridge the gap from use cases to the sequence diagrams [7]. Robustness diagrams are used to describe three kinds of object/class stereotypes and actors in the corresponding use case and their interactions. The notations for boundary, entity and control class are shown in Fig. 2.

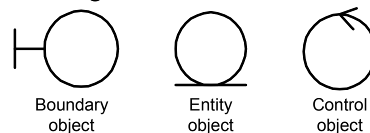

Fig. 2. Three kinds of object/class stereotype.

The boundary class is used to model the system interface, which interacts with actors directly. It handles the communication between the system surroundings and the inside of the system [8]. The entity class is used to model a real world entity, which is typically independent of the surroundings. The control class is responsible for the flow of events in the use case and is application-dependent. Determining the control classes for a problem is very subjective. Automatically identifying boundary, entity and control classes depends on the syntactic analysis of the sentence in the use case specification document. There are some rules, which can be used to validate the robustness diagram. Actors can talk only to boundary objects; boundary objects can talk only to controllers and actors; entity objects can talk only to controllers; controllers can talk to both boundary objects and controllers but not the actors.

D. Overview of the Methodology

With Robustness Analysis included, the transition from use cases to the class model in our methodology is illustrated in Fig. 3. The system's requirements are acquired and modeled

with use cases. For each use case, a specification document based on the template (see Table 1) is created to describe the details of the use case. Boundary objects/classes, entity objects/classes and control objects/classes are found from the specification with the support of a glossary, and the interactions between them are identified as well. During this phase, the classes' attributes are identified.

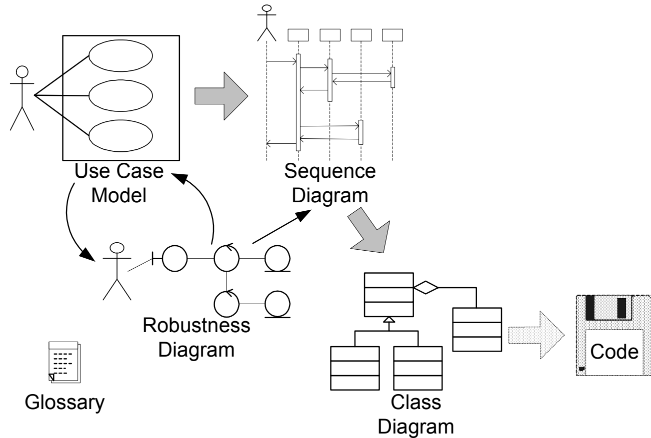

Fig. 3. Transition from use cases to class model.

The glossary includes all the terms for the problem domain, and all the requirements are documented using it. It helps to remove the vagueness introduced by using natural language in use cases. A sequence diagram is built on the basis of the robustness diagram. The messages between the actor and objects/classes are considered in this phase. The operations are allocated to the classes according to the services, which they can supply. Then association and aggregation relationships and other elements are added between the classes, whose attributes and operations have been identified in the former phases. The class model of the system is documented by class diagrams. Implementation can be carried out once the class model is completed.

To automate the transition from use case to class model, the methodology described in the former part is currently being implemented in a tool. As the use case specifications are normally written in nature language, there will be much vagueness in the context, which is the main difficulty for the machine when distilling useful information from the text. To remove the vagueness, the use case specifications are written or rewritten according to some rules [9]. The rules can also be seen as constraints on the written English language. Table 2 gives an initial list of such rules which may be expanded if needed. All the statements in use case specifications are encoded in XML to be parsed and processed. Three language patterns are introduced to represent most of the statements structures in the use case specification. The patterns and their simple descriptions are listed in Table 3. For the basic pattern, namely *Simple Statement*, there are 8 sub patterns, which determine the semantic details of the statement, and finally lead to different processing methods. Table 4 lists the 8 sub patterns for *Simple Statement*. With these rules and patterns, the transition from the use case model to the class model can be formalized to some extent, and finally be implemented by a CASE tool.

TABLE 2
RULES FOR WRITING/REWRITING USE CASE SPECIFICATION

Rule 1:	Write the use case specification in 3 statement patterns, namely simple statement, while-do statement and if-then statement.
Rule 2:	Speak in active voice rather than passive voice.
Rule 3:	Set up a glossary for the domain; use the same term from the beginning to the end; replace all the pronouns with specific names of the entities.
Rule 4:	Use the same verb for the same service or action in different statements.
Rule 5:	Keep the form of complex predicate unique.

TABLE 3
LANGUAGE PATTERNS IN USE CASE WRITING

Simple Statement	Most of the simple English sentences are of this pattern. The structure of this basic pattern is Subject + Predicate.
While-Do Statement	This pattern is used to represent the repeated event flows when a condition is fulfilled. Both While-part and Do-part are of Simple Statement.
If-Then Statement	This pattern is used to represent the specific event flow under a certain condition. If-part is of Simple Statement and Then-part is of Simple Statement or set of Simple Statement.

TABLE 4
SUB PATTERNS OF SIMPLE STATEMENT

I	Subject + Verb
II	Subject + (Verb + Object)
III	Subject + (Be + Predicative)
IV	Subject + (Verb + Direct Object + to/for + Indirect Object) *Subject + (Verb + Indirect Object + Direct Object)*
V	Subject + (Verb1 + Object + to + Verb2)
VI	Subject + (Verb1 + Object1 + to + Verb2 + Object2)
VII	Subject + (Verb + Object + Present Participle) Subject + (Verb + Object + Adjective)
VIII	Subject + (Verb + Object + Past Participle)

Several paradigms can be used to develop a CASE tool to support class elicitation and modeling. Since agents were initially developed to support such features as built-in intelligent functionality, autonomy, reactivity and easy communication, it is a good choice to build an agent-based system to implement this tool. The agent architecture is introduced in the next section, and the details of the technologies to implement the system are discussed. The analysis and design for this system are presented in Section IV.

III. AGENT SYSTEM ARCHITECTURE

The Foundation for Intelligent Physical Agents (FIPA) has set up a series of specifications about agents, some of which focus on agent architecture [10], [11]. There are also other approaches for agent architectures, e.g., the open agent architecture from SRI International [12]. The basic aspects related to the agent architecture are similar: agent platform, agent management system, directory facilitator, agent messaging and so on. When the agent system is implemented, the detail technologies are decided. The agent architecture used in this project is introduced below.

A. Agent Platform

The Agent Platform is the key element to the enterprise

architecture and provides the physical infrastructure on which agents can be deployed. The Agent Platform consists of the machine(s), operating system, agent support software and other elements. The minimum set of agent platform capability comprises an agent management system, a directory facilitator and agent messaging. For the system described in this paper, J2EE is chosen as the agent platform since it provides adequate facilities to deploy agent-based systems. Some of the requirements for the platform are: high-speed communications, a single system manager and a single security enclave, which are good for the system's performance and users' management of the system. J2EE fulfills these requirements well. J2EE specification defined a number of standard services for use of J2EE components: Java Naming and Directory Interface (JNDI), JDBC, CORBA Compliance, Java Transaction, XML Deployment Descriptors, and Java Message Service (JMS). Most of them are useful for agent implementation. It is quite natural for the developer to design agents as Enterprise Java Beans (EJB), and deploy them on a J2EE platform.

B. Agent Management System

The agent management system is responsible for supervising access to and use of the agent platform. It is a mandatory component of the agent platform, which means that an agent platform can only have one agent management system. The Agent Management System maintains a directory of agent Identifiers, which is related to the registration addresses of agents. The agents deployed on the agent platform must register with the agent management system to get a valid agent identifier. The agent management system should support such functions as: register, deregister, modify, search, get-description and agent lifecycle control. J2EE Software Developer's Kit (SDK), which is a complete implementation of J2EE, provides an application deployment tool and an array of scripts for assembling, verifying, and deploying J2EE applications and managing development and production environments.

C. Directory Facilitator

The agent directory facilitator provides yellow page services to the agents. It is a reification of the agent directory service. An agent directory service is a shared information repository in which agents may publish their agent directory entries and in which they may search for agent directory entries of interest. The Java Naming and Directory Interface (JNDI) of J2EE perfectly fits this task.

D. Agent Messaging

Agent communication is a key issue of an agent-based system. Here, we specify the communication in a message-based way. In this sense, all the information transferred between agents is carried by messages and all the actions triggered are driven by messages. Communication language, communication ontology and message transportation mechanism are required.

1) Communication language: When the agents are communicating with each other, a standard agent communication language is needed, so different agents can interpret the messages received and thus interoperate with the other agents according to the messages. The messages will be encoded into the standard form before being sent, and decoded after being received. The Knowledge Query and Manipulation Language (KQML), which is also part of the FIPA specifications, is a language designed to support interactions among intelligent software agents [13]. We use eXtensible Markup Language (XML) as agent communication language in the architecture. Although XML is not specifically designed for use as agent communication language, its characteristics make it suitable for this task.

2) Communication Ontology: An ontology is a formal explicit description of concepts in a domain of discourse (classes, sometimes called concepts), properties of each concept describing various features and attributes of the concept (slots, sometimes called roles or properties), and restrictions on slots (facets, sometimes called role restrictions) [14]., An ontology for agent communication is used to define the concepts and class systems that enable information to be understood by different agents. An agent communication ontology is tightly coupled with the agent communication language. With XML being the communication language in our approach, the XML validation methods are responsible for ontology checking. There are basically two types of XML validation methods: XML DTD (Document Type Definition) and XML schema. Because XML schema precedes XML DTD on some aspects of data expression, XML schema is preferred to implement the communication ontology. Nearly all the XML parser API packages support both XML DTD and XML schema.

3) Message Transportation Mechanism: In agent environments, messages should be schedulable, as well as event driven. They can be sent in synchronous or asynchronous modes. Furthermore, the transportation mechanism should support unique addressing. CORBA, RMI, DCOM and other technologies are able to support agent messaging. Java Messaging Service (JMS) is used in our approach. JMS is an integral part of J2EE platform and there are other vendors who support JMS in their products.

IV. AGENT SYSTEM ANALYSIS AND DESIGN

An agent-based system was designed based on OMCP [3]. Fig. 4 shows the structure of the system. The Requirements Acquisition Agent is responsible for collecting the requirements from the stakeholders and transforming them into the artifacts that later agents in the structure can read and process. The Object Identification Agent finds the objects from the information provided by the Requirement Acquisition Agent. The Attribute Identification Agent finds the states for the object and makes them the attributes for the corresponding object; the Association Identification Agent finds the semantic relationships between the objects identified in the domain; the Behavior Identification Agent works on add operations to the objects. Finally, the Object Refinement Agent refines the object model with inheritance and outputs the hierarchical class model.

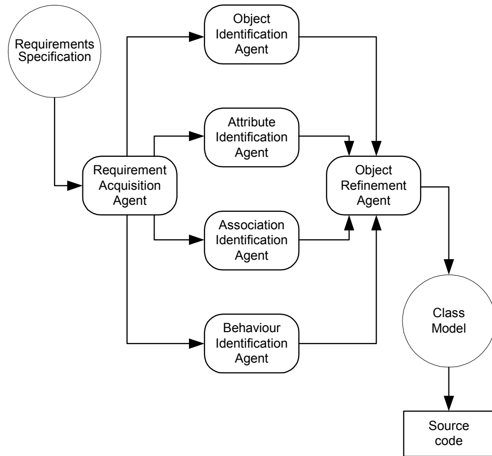

Fig. 4. Structure of the agent-based system for OMCP.

In this paper we devise a new agent architecture for the system. A methodology named Gaia [15], [16] is applied to analyze the agents for the system. The design of the system is discussed as well.

A. Role model

A *role* is viewed as an abstract description of an entity's expected function. Roles are characterized by two types of attributes: permissions (or rights) and responsibilities. Permissions are the type and the amount of resources that can be exploited when carrying out the role. Responsibilities are the role's functionalities. The participating agents in the system are identified in terms of role, and the role schemas are defined.

From the user's point of view, the system's boundary is established. The user will use the system to acquire requirements and the system will generate the class model from the requirements. Two roles are responsible for interaction with the user, namely RequirementDistiller and ClassGenerator. RobustnessAnalyst creates the robustness diagram, and SequenceAnalyst creates the sequence diagram. The details of the role models are documented by role schemas shown in Fig. 5 to Fig. 8. This system can work for a certain problem domain. When the problem domain is changed, the knowledge the agents used needs to be modified. Therefore, another agent in charge of knowledge for variant domains should be introduced to the system. This agent is responsible for supplying proper knowledge for the other agents. Introducing this agent, all the other agents should respond to another protocol: LoadKnowledge. The role schema for so called KnowledgeSupplier is shown in Fig. 9.

Role Schema: REQUIREMENTDISTILLER
Description: Get requirements from the stakeholder; process the requirement into use case specification; generate the message; send the message with use case specification to other agents needing it.
Protocols and Activities: AcquireRequirements, Process, GenerateMessage, SendMessage
Permissions: ***read*** Requirements ***generate*** Message
Responsibilities Liveness: REQUIREMENTDISTILLER = (AcquireRequirements. Process. GenerateMessage. SendMessage)$^{\omega}$ Safety: Message is received

Fig. 5. RequirementDistiller schema.

Role Schema: ROBUSTNESSANALYST
Description: Receive the message containing use case specification; parse the message; process the information; generate robustness diagram; generate the message; send the message.
Protocols and Activities: ReceiveMessage, Parse, Process, GenerateDiagram, GenerateMessage, SendMessage
Permissions: ***read*** Message ***generate*** Message
Responsibilities Liveness: ROBUSTNESSANALYST = (ReceiveMessage. Parse. Process. GenerateDiagram. GenerateMessage. SendMessage)$^{\omega}$ Safety: Sent message is received

Fig. 6. RobustnessAnalyst schema.

Role Schema: SEQUENCEANALYST
Description: Receive the message containing robustness analysis information; parse the message; process the information; generate sequence diagram; generate the message; send the message.
Protocols and Activities: ReceiveMessage, Parse, Process, GenerateDiagram, GenerateMessage, SendMessage
Permissions: ***read*** Message ***generate*** Message
Responsibilities Liveness: SEQUENCEANALYST = (ReceiveMessage. Parse. Process. GenerateDiagram. GenerateMessage. SendMessage)$^{\omega}$ Safety: Sent message is received

Fig. 7. SequenceAnalyst schema.

Role Schema: CLASSGENERATOR
Description: Receive the messages containing robustness analysis information and sequence analysis information; parse the messages; process the information; generate class diagram; generate output; send output.
Protocols and Activities: ReceiveMessage, Parse, Process, GenerateDiagram, GenerateOutput, SendOutput
Permissions: ***read*** Message ***generate*** Output
Responsibilities Liveness: CLASSGENERATOR = (ReceiveMessage. Parse. Process. GenerateDiagram. GenerateOutput. SendOutput)$^{\omega}$ Safety: Output is saved

Fig. 8. ClassGenerator schema.

Role Schema: KNOWLEDGESUPPLIER
Description: Determine the proper knowledge for other agents; send the knowledge to the agents.
Protocols and Activities: ReceiveMessage, Parse, PrepareKnowlsdge, SendKnowledge
Permissions: ***read*** Message ***generate*** Knowladge
Responsibilities Liveness: KNOWLEDGESUPPLIER = (ReceiveMessage. Parse. PrepareKnowlsdge. SendKnowledge)$^{\omega}$ Safety: Knowledge is loaded

Fig. 9. KnowledgeSupplier schema.

B. From Agent Analysis to Design

In Gaia, three models are suggested in the phase of agent analysis. The purpose of the agent model is to document the various agent types that will be used in the system under development, and the agent instances that will realize these agent types at run-time. This idea just comes from the classic object oriented thinking, i.e., agent type and agent instance are mapped directly from class and object. However, agent-based analysis and design is different from the classic OOAD. Normally, there are not so many agents to be considered as the objects in OO analysis, so agent type makes no sense as far as abstraction is considered. Agents' lifecycle is much

longer than objects, which means the instantiations and destructs of agents occur less frequently.

The design process in Gaia involves generating three models: *agent model* that identifies the agent types that will make up the system, and the agent instances that will be instantiated from these types; *service model* that identifies the main services, which will be associated with each agent type; and *acquaintance model* that documents the acquaintances for each agent type. Actually what constitute the contents of the service and acquaintance models have already been identified in the agent analysis phase. This is another source of deviation from the OOAD techniques. In OOAD, the design phase is a segment of the development process connecting analysis and implementation phases. But for an agent-based system, the implementation is usually based on another paradigm, such as OO design. Therefore, in agent-based development analysis is the only crucial task. As a result, the detailed internal design of constituent agents and the interactions between them may be better achieved using OO design or similar paradigms.

The agent model for our system, which comprises of agents in the name of the roles, is shown in Fig. 10. The structure is different from the one in Fig. 4. This is mainly determined by the methodology applied in the two approaches. The original one is based on classic OMCP, while the one discussed later is based on RUP and robustness analysis. Let's name the original one model 1 and the latter one model 2. Model 1 is a typical hierarchy structure, and the system can be divided into input layer, output layer and the median layers, which are responsible for the system's functions. Every agent in the median layers functions for one aspect of the object/class, which comprises of object/class name, attributes, behaviors and association relationships. Model 2 is not as tidy as model 1, but looks more compact. The tasks of finding the object/class, identifying the attributes, and allocating the behaviors are conducted one by one, and incrementally. In model 1, all the tasks are parallel in the structure.

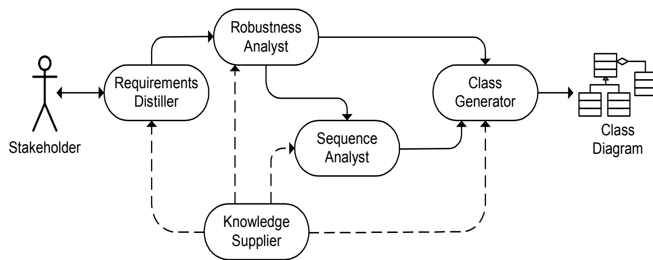

Fig. 10. System structure.

V. CONCLUSION

We proposed a methodology to acquire stakeholders' requirements, normalize the requirements, elicit objects and classes from the requirement, and finally build class model. An agent-based system is analyzed and designed to aid the methodology. The architecture of such an agent-based system is discussed and the detail technologies are given to implement the agent architecture.

REFERENCES

[1] G. Booch, *Object oriented design with applications*. The Benjamin Cummings Publishing Company, 1991.

[2] A. Cockburn, *Writing Effective Use Cases*. Addison-Wesley, 2000.

[3] R.S. Wahono, B.H. Far, "A Framework of Object Identification and Refinement Process in Object-Oriented Analysis and Design". Proceedings of The 1st IEEE International Conference on Cognitive Informatics, ICCI2002, Calgary, Canada, 2002.

[4] I. Jacobson, M. Christerson, P. Jonsson and G. Övergaard: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Wokingham (England), 1992.

[5] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall, 2002.

[6] J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[7] D. Rosenberg, K. Scott, *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*. Addison-Wesley Professional, 2001.

[8] T. Quatrani, G. Booch, *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley, 1999.

[9] Liwu Li, "A semi-automatic approach to translating use cases to sequence diagrams". Proceedings of Technology of Object-Oriented Languages and Systems, 1999.

[10] Foundation of Intelligent Physical Agents, "Abstract Agent Architecture Specification". FIPA, 2002. Available: http://www.fipa.org/specs/fipa00001/

[11] Foundation of Intelligent Physical Agents, "Agent Management Specification". FIPA, 2002. Available: http://www.fipa.org/specs/fipa00023/

[12] D. Martin, A. Cheyer, and D. Moran, "The Open Agent Architecture: A framework for building distributed software systems". Applied Artificial Intelligence, 13(1):91-128, 1999.

[13] T. Finin, R. Fritzson, D. McKay, and R. McEntire, "KQML as an agent communication language". The Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94). ACM Press, 1994.

[14] N.F. Noy and D.L. McGuinness, "Ontology Development 101: A Guide to Creating Your First Ontology". Stanford Knowledge Systems Laboratory Technical Report KSL-01-05, 2001. Available: http://www.ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness-abstract.html

[15] M. Wooldridge, N. R. Jennings, and D. Kinny. "The Gaia Methodology for Agent-Oriented Analysis and Design" Journal of Autonomous Agents and Multi-Agent Systems 3 (3), 2000.

[16] M. Wooldridge, N. R. Jennings, and D. Kinny. "A Methodology for Agent-Oriented Analysis and Design" Proc. 3rd Int Conference on Autonomous Agents (Agents-99) Seattle, WA, 1999.