Proc. Estonian Acad. Sci. Eng., 2001, **7**, 1, 5–21

https://doi.org/10.3176/eng.2001.1.02

AGENT-BASED SOFTWARE DESIGN

Margus OJA[a], Boris TAMM[b], and Kuldar TAVETER[c]

[a] Department of Informatics, Tallinn Technical University, Ehitajate tee 5, 19086 Tallinn, Estonia; margus.oja@radiolinja.fi
[b] Cybernetica Ltd, Akadeemia tee 21, 12618 Tallinn, Estonia; btamm@is.cyber.ee
[c] VTT Information Technology, Tekniikantie 4B, Espoo, Finland; kuldar.taveter@vtt.fi

Received 23 November 2000

Abstract. This paper introduces an agent-based software development method. It defines a limited number of components of an agent-based software system and shows the possibility of designing and implementing actual software. Starting points in developing agent-based software are the business rules and the basic agent-based concepts as defined in the paper. A notation for visualizing the defined concepts is introduced. The notation of the unified modelling language UML is used. The possibility to define rules for mapping agent model elements onto source code is shown using the JADE agent platform.

Key words: software agents, software design, software engineering, UML, JADE.

1. INTRODUCTION

Several efforts have been made to develop agent-based software methodologies [1,2]. However, most of the studies concentrate on specific areas of the agent software – agent models, reasoning logic and agent actions, agent communication, agent programming languages and frameworks. To gain wide acceptance of agent-based software in practice, methods covering the full software development cycle from analysis to implementation are required. Such methods can be developed on the basis of agreed agent software concepts, modelling techniques supporting these concepts, and finally agent software implementation frameworks supporting the same concepts.

We are going to present a method covering all these stages. The aim of the paper is not to introduce a fully functional agent-based method but rather to show that it is possible to develop one.

2. SCOPE OF AGENTS, BACKGROUND, RELATED WORK

In the present work the word *agent* will be used in the meaning of an abstraction unit when designing and implementing software systems. Starting to design new software, the initial specification should describe what kinds of agents can be found in the system and how do they co-operate with each other in order to offer the needed functionality. When it comes to the design and implementation of the specified agents themselves, then traditional object-oriented programming languages must be used.

Agent-based approach is not applicable everywhere – it can be used only in the circumstances where software system will be built of autonomous units, each executed separately. The restriction is quite natural – like simple procedural approach is the best for designing simple software for calculating square root or like object-oriented approach is the best for designing complex software working as *one unit*.

Agents add value to the traditional software design by offering tools for handling the most general level of the problem domain. They represent the main building blocks of a distributed software system without describing the internal structure of the individual blocks.

According to Nwana [3] the concept of agent dates back to the early days of research into DAI in the 1970s, to Carl Hewitt's concurrent actor model (1977) where "...an actor is a computational agent which has a mail address and behaviour. Actors communicate by message passing and carry out their actions concurrently". Nwana states that the overuse of the word *agent* has masked the fact that, in reality, there is a truly heterogeneous body of research carried out in this manner. Nwana even states that the chance of agreeing on a consensus definition for the word *agent* is nil. He makes an effort to classify different types of existing software agents according to the abilities to learn, co-operate, act autonomously, and comes up with the following classification: collaborative agents, interface agents, collaborative learning agents, and smart agents (Fig. 1). Nwana admits also that there exist other ways of agent classification. This paper is using the word *agent* in the meaning of *collaborative agent* according to the above classification. Collaborative agents emphasize autonomy and co-operation.

In order to co-operate, agents need a communication language. Wagner refers to an informal definition of software agent by Genesereth and Ketchpel in his book [4] as follows: "An entity is a software agent if and only if it communicates correctly in an agent communication language (ACL)". Knowledge Query and Manipulation Language (KQML) [5] is the most widely used ACL. Java Agent Development (JADE) framework [6] uses the ACL [7], specified by the Foundation for Intelligent Physical Agents (FIPA), that is very similar to KQML. An overview of JADE will be presented later in this paper. FIPA is a non-profit association whose purpose is to promote the success of emerging agent-based applications, services, and equipment. FIPA has published a set of specifications for agent-based software development frame-works. Some of the normative specifications by FIPA are: Agent Management, ACL, and Agent Software Integration [7].

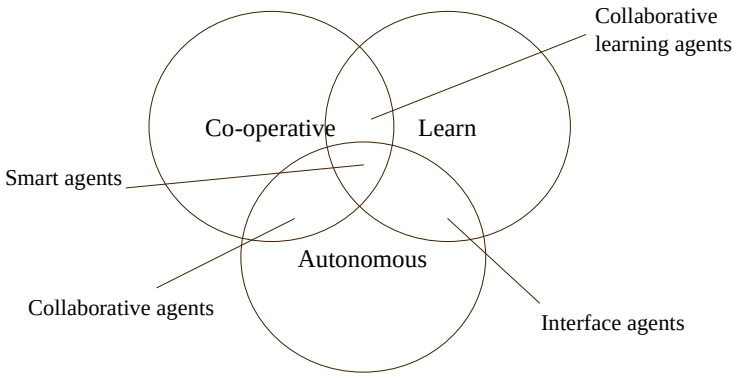

Fig. 1. Agent classification according to Nwana [3].

Taveter and Tamm [8] have introduced layered architecture of agent-based software where the software is considered as consisting of three layers: agent, object, and binary layer (Fig. 2). Similar approach is also followed in this paper. We regard agents as the top-level abstraction units in software design while the agents themselves are implemented using object-oriented programming languages. We accept also objects in the top agent layer, but these are the objects that agents manipulate with and not the implementation level objects. We do not accept any object-to-object communication in the agent layer. The computers will finally execute only the compiled binary code regardless of the high level languages used by humans. The first two levels are meant for humans and should make software developing easier, faster, and more reliable. Adding a new *agent layer* on top of the object layer will be justified only if it adds value compared to the object-oriented approach. The added value is achieved by scoping agents a) to distributed systems consisting of separate autonomous software units and b) to general level where common sense is sufficient to model the software system without going into any technical details.

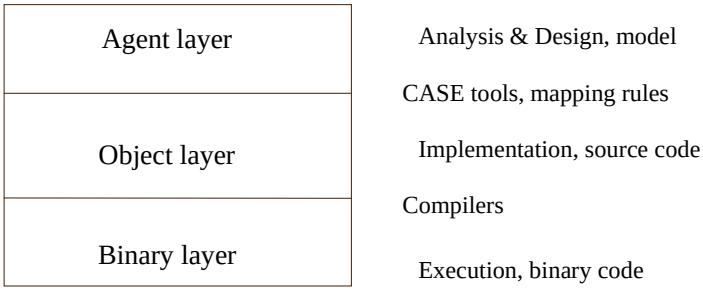

Fig. 2. Triple layer agent software architecture.

A related paper [1] states the following: “Gaining wide acceptance for the use of agents in industry requires both relating it to the nearest antecedent technology (object-oriented software development) and using artifacts to support the development environment throughout the full system lifecycle”. This statement matches well with our approach where we introduce a full development cycle from modelling agent software to the actual implementation. However, most of the work [1] deals with agent interaction protocols and shows the possibilities of using different UML diagrams for modelling agent interactions. The starting points are UML and object-oriented approach. Agents are presented as *an extension of active objects, exhibiting both dynamic autonomy (the ability to initiate action without external invocation) and deterministic autonomy (the ability to refuse or modify an external request)*. The scopes and aims of the work [1] and of the present paper are different. The first covers only visual modelling based on UML. It does not define the components of agent-based software and it does not propose any methods for implementing a visual model in software.

Another related work [2] concentrates similarly to [1] on adjusting UML visual diagrams to agents but lacks also a clear definition of concepts. The paper is UML-driven and maps concepts from notations in diagrams onto the metamodel (defined for object-oriented concepts in UML!). The mapping onto metamodel is questionable – how can we map agent-based concepts onto the metamodel elements based on object-oriented concepts?

The present paper first defines the concepts of an agent-based software system as a starting point, then introduces visual notation for representing these concepts, and finally shows that it is possible to define rules for mapping the visual model onto the actual program code. UML is not used as a starting point; it is used rather as an aiding tool in visualizing the defined concepts.

3. CONCEPTS AND VISUAL NOTATION

In the following paragraphs the main concepts of an agent-based software model are outlined. We shall specify a limited list of components of agent-based software. There is evidently a need for more components (like actors, virtual knowledge base, state, transition, case studies, etc.) but this is out of the scope of this paper. We shall explain the meanings (brief semantics) of selected components and introduce the notation of the components on diagrams. After that we shall introduce mapping rules from the visual model onto actual software code. JADE framework will be used in the code examples. Note that there is a difference between the code, generated according to the mapping rules, and the final fully functional code. The generated code serves as a frame and that must be manually modified in order to become fully functional. Not every detail should be present in the model. Unfortunately, there are no strict rules about what to include in the visual model and what should be added only to the code. Similar statement can be found also in the UML Summary [9]: “The UML, a visual modelling language, is not intended to be a visual programming language”.

Business rules [10] are a set of constraints and directions. The rules define how can we get what is needed. Business rules can be applied to the behaviour of a single agent and also to the co-operation of a group of agents. The business rules are visualized by diagrams; there is no visual notation for a business rule because it is not an independent component in our approach. Rather the rules will be used in a model as the *glue* between the components – they define the presentation of the components in diagrams.

An **agent** is an autonomous software unit that can exist independently of other similar units in the software system. An agent performs some functions for other agents or external actors. Agents communicate with each other via messages in an agent communication language. It is interesting to compare this definition with the FIPA agent definition [7]: "An agent is the fundamental actor in a domain. It combines one or more service capabilities into a unified and integrated execution model which can include access to external software, human users and communication facilities". The main difference is that in our approach we emphasize the software nature of an agent and the communication between agents. An agent is expressed on diagrams as a dashed rectangle (Fig. 3). The rectangle can contain only the name of an agent because in our approach an agent does not have any internal structure. There are two reasons for using dashed rectangles for agent notation: a) to distinguish an agent from an object and b) an agent can be in the role of a system boundary in used case diagrams and in UML this is denoted with dashed lined rectangle.

A **message** is a speech act that one agent performs in order to request or send information to other agent(s) in the format of an ACL. Note that UML defines message from a different viewpoint [11], putting emphasis on the general-to-specific relationship: "A message instance is a communication between objects

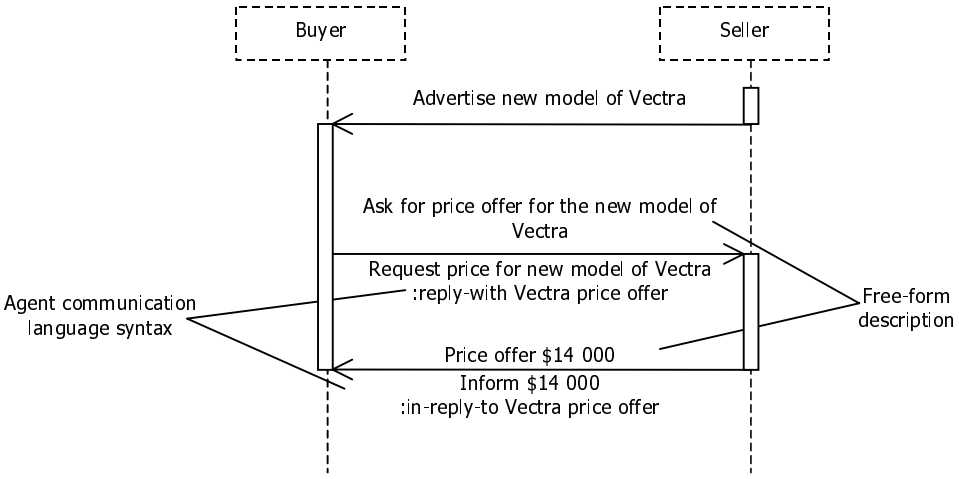

Fig. 3. Agent, message, and behaviour notations.

that conveys information with the expectation that action will ensure. A message is the description of a set of message instances of the same form". Messages are sent asynchronously; an agent can have multiple *conversations* at the same time and can receive different messages from different agents with no particular order. A message is depicted as a labelled arrow (Fig. 3). The label contains the message description and the arrowhead direction defines the sender and the receiver. The message label string can have two different forms (both at the same time) – a free form description and a form that accords to the syntax of the ACL used in the system. The free form description should be used for discussing the model with people not familiar with the ACL. The language specific syntax should be used for easier migration from the model to the actual software code.

A **behaviour** is a sequence of agent's actions performed as a result of a specific event. Actions in our approach can be sending of messages, waiting for incoming messages, internal actions, and object manipulations. An event can be receiving a message or a specific return value of some periodical test procedure (timeout, end of day, etc.). Note that UML and FIPA do not define the term *behaviour*. The closest concept in UML is *activation*. Behaviour is depicted on sequence diagrams by grouping an agent's actions with a rectangle area on the agent's lifeline (Fig. 3). A single diagram can express multiple behaviours of an agent.

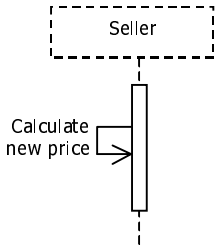

Fig. 4. Notation of the internal action.

An **internal action** is an activity or a group of sequential activities of an agent. Internal action is not related to any objects manipulated by the agent or to any messages sent or received. An internal action is depicted as a labelled arrow directing back to the agent's lifeline (Fig. 4). The label contains a free form description of the action. An internal action should not be mixed with a message even if they look visually the same – an agent does not send messages to itself. It is possible for an agent to send a message to itself but it just does not make sense.

An **object** is a passive component in the system that is manipulated directly by an agent. An agent's virtual knowledge base consists of the information tied to objects. Examples of objects in our approach are *bill*, *schedule*, *switch*, etc. An agent can manipulate an object in order to get or change some information or to perform some action with the object like, for example, deletion. An object is depicted as a rectangle with a name (Fig. 5).

We define the communication between an agent and an object as a **manipulation**. An agent communicates with objects not in ACL but in the form corresponding to the object. For example, if an object is a row in a relational database table then the agent should use the SQL commands specific to the database for manipulation. In our approach we do not allow any communication between objects. A manipulation is expressed as a labelled arrow (Fig. 5). The arrow is directed from the agent to the object being manipulated with and the

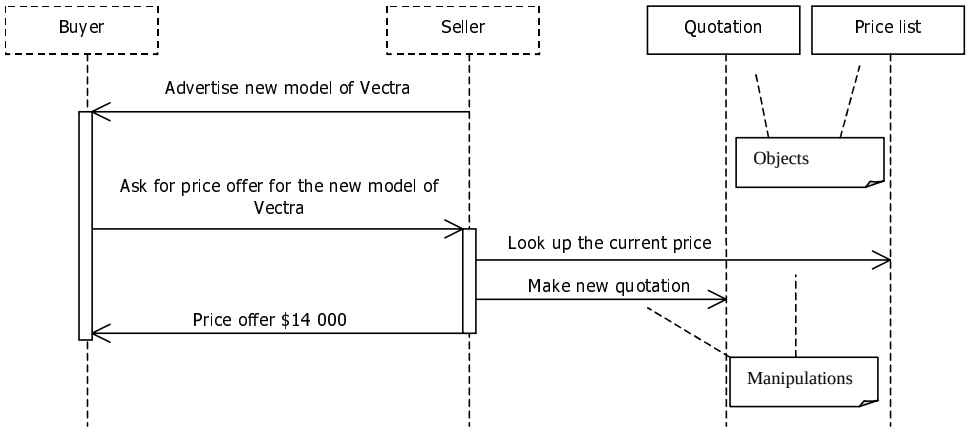

Fig. 5. Object and manipulation notation.

label contains the manipulation description. Note that a manipulation is denoted with an arrow exactly ike a message, the difference is defined by the receiver. The manipulation labels can have two different forms (both simultaneously like message labels) – a free form description as shown in Fig. 5 and with the formal syntax of the programming language used during implementation. The UML notation can also be used for expressing the formal syntax [11].

Now, having defined all the needed concepts let's look how we can tie them together in a visual model using business rules. A business rule can be visualized using one or more behavioural diagrams matching the rule best. UML defines a set of behavioural diagrams [9]: statechart, activity, sequence, and collaboration diagrams. The same rule can be expressed on different diagrams and one diagram can visualize several rules. Different diagrams emphasize visually different aspects of a rule. We are going to use only the sequence diagrams to keep the scope in reasonable limits. Figure 6 contains the visual representation of sample business rules about the lifecycle of a sales quotation:

– A salesperson prepares a new quotation draft.

– Only quotations satisfying the customer's requirements are presented to the customer.

– A customer can accept or reject a presented quotation.

– Quotations not presented to a customer are archived as drafts.

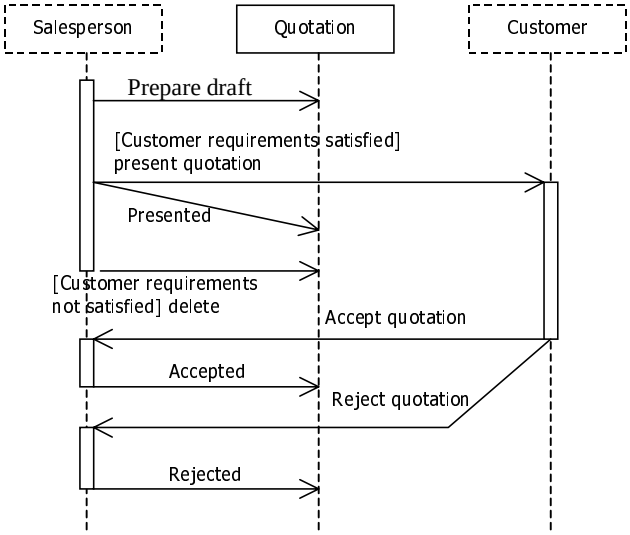

Fig. 6. Visual representation of sample business rules in the case of a sales quotation handling.

4. MAPPING MODEL CONCEPTS ONTO SOURCE CODE

In this section we show that it is possible to define rules for generating the actual program code on the basis of a visual model. Generation of the program code is based on mapping rules from the modelling language concepts onto parts of the program code. We use JADE as the target source code platform. The introduced rules apply only in the JADE environment. Some of the rules are of a more general nature, some are Java language specific and some JADE specific. We are not going to classify the rules in this paper.

The generated code is compilable but it still needs manual additions to make it fully functional. Places where programmer should (or could) add an additional code are marked with comments *to do*. The generated code can be executed and the messages sent between agents can be visualized using the Sniffer agent provided by JADE. In the source code examples below we are using ***emphasized*** text for names that are generated from different names or texts used in the visual model. We are using **CAPITAL** letters to highlight the beginning of mapping definitions of a previously defined concept.

JADE is a software development and runtime framework fully implemented in Java language. It simplifies the implementation of multi-agent systems through a middleware that claims to comply with the FIPA specifications and through a set of tools that support the debugging and deployment phase. The agent platform can be distributed between machines (which do not even need to share the same OS) and the configuration can be controlled via a remote graphic user interface. JADE is completely implemented in Java language and the only system requirement is the version 1.2 of JAVA (the run time environment or the JDK).

JADE is implementing agent behaviours using behaviour classes. A non pre-emptive multitasking of executing different behaviours of an agent has been implemented for JADE.

An **AGENT** in a visual model is implemented as a subclass of Agent with one predefined behaviour named MessageHandler. Both classes are included in a new package named after the agent. Below are the exact rules for generating JADE code for an agent:

– The name of the agent class is the name of the agent in the visual model.

– The name of the new package is also named after the agent. The package contains the agent class, it's behaviour classes, and classes for objects manipulated by the agent.

– Two JADE standard member functions "setup()" and "takeDown()" are included for adding manually application specific start-up and clean-up codes.

– The predefined behaviour MessageHandler is started by "setup()".

A code, generated using these rules, is shown below.

```
package agentName;

import jade.core.*;

/**
 * to do: description of the agent for the javadoc tool.
 */
public class AgentName extends Agent {

  protected void setup() {
    // to do: add necessary start-up code

    addBehaviour(new MessageHandler(this));
  }

  protected void takeDown() {
    // to do: add necessary clean-up code

  }
}
```

The reason for introducing a predefined MessageHandler class is the need for implementing agent feature to respond to different messages from different agents. The task of the MessageHandler is to listen to incoming messages and forward all the received messages to appropriate behaviours. Exact rules for generating JADE code for MessageHandler class are the following:

– The class MessageHandler is derived from JADE class CyclicBehaviour and will therefore run continuously.

– If no messages have arrived, the behaviour will block and restart after a new message has arrived.

– If a message has arrived, the MessageHandler has to check the message information, start corresponding behaviour of the agent and resume waiting for incoming messages. Checking the message and starting another behaviour are described afterwards as rules for mapping a message onto the program code.

A code, generated using these rules, is shown below.

```
package agentName;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;

/**
 * Predefined incoming message handler
 */
class MessageHandler extends CyclicBehaviour {

  /** constructor of the behaviour */
  public MessageHandler(Agent a) {
    super(a);
  }

  /** actual implementation of the behaviour */
  public void action() {

    // wait for a message
    ACLMessage received = myAgent.receive();
    if (received == null) {
      block();
    }
    else {

      // check message information to
      // start corresponding behaviour of the agent
    }
  }
}
```

A **BEHAVIOUR** is implemented as a subclass of JADE OneShotBehaviour class. A **MESSAGE** is mapped in JADE onto the code of behaviours of both sending and receiving agents. Exact mapping of an incoming message depends on whether the message is the first message of the behaviour in a diagram or there are other messages included in the behaviour before. The rules of generating JADE code are as follows:

– If the first message of a behaviour is an incoming message then this message is considered as the trigger to start the behaviour. Start-up of the behaviour is implemented in the code of the MessageHandler. The message is passed as a parameter to the constructor of the behaviour class.

– If an incoming message is not the first message of a behaviour, then the receiving of the message is implemented in the behaviour class code and not in the MessageHandler**.**

– Sending of messages is always implemented in the behaviour class of the sending agent.

– The behaviour class will be in the same package with the agent.

– The behaviour class is named after the first message.

– Sending and receiving of messages and performing internal actions are implemented in the same sequence as shown in sequence diagrams.

The code below makes no assumption if the first message is incoming or outgoing. It shows the code generated according to common rules in both cases. Code, generated on the basis of the message specific rules, is shown afterwards.

```
package agentName;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;

/**
 * to do: description of the behaviour for the javadoc tool.
 */
class MessageFreeFormDescription extends OneShotBehaviour {

  /** constructor of the behaviour */
  public MessageFreeFormDescription(Agent a) {
    super(a);
  }

  /** actual implementation of the behaviour */
  public void action() {
    // to do: add message handling code

  }
}
```

An outgoing message is implemented as a part of the code in the “action()” method of the behaviour. If the ACL syntax is provided in the model then the new message of the defined type and with defined parameters is created in the code. If ACL syntax is not provided, the new message is of type UNKNOWN. Message parameters are set as follows:

– Receiver agent’s name is specified in lower case letters. This is due to the JADE feature that during registration the agent name is converted to lower case.

– Sender agent’s name is set automatically by JADE and no additional code is needed.

– The conversation ID of the message is set to the diagram title.

– If the message description does not have ACL syntax then the whole free form description is regarded as the content. Otherwise the actual content value from the description in the diagram is set as content of the message in program code.

– The content must be enclosed with parenthesis due to JADE framework feature.

– Other optional parameters are assigned according to the ACL syntax description.

Below is an example of the source code of an agent's behaviour by sending a message. The part of the code, generated according to the rules presented above, is highlighted.

```
package agentName;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;

/**
 * to do: description of the behaviour for the javadoc tool.
 */
class MessageFreeFormDescription extends OneShotBehaviour {

  /** constructor of the behaviour */
  public MessageFreeFormDescription(Agent a) {
    super(a);
  }

  /** actual implementation of the behaviour */
  public void action() {
    // to do: add message handling code

    // message free form description
    CLMessage send = new ACLMessage(ACLMessage.MESSAGETYPE);
    send.addDest("receiveragentname");
    send.setConversationId("Dialog_Title");
    send.setContent("(content from the label)");
    myAgent.send(send);
  }
}
```

If start of the agent's behaviour is triggered by an incoming message then the message is implemented by parts of the code in the predefined behaviour MessageHandler and by parts of the code in the behaviour class. The rules are the following:

– Class MessageHandler checks the incoming message using the message type and all available parameter values from the model. If the incoming message passes the check then the behaviour is started using method "addBehaviour()" of the agent class.

– In the behaviour class (subclass of OneShotBehaviour) a private member variable "received" of type ACLMessage is defined. The incoming message is assigned to the variable in the constructor.

Below is an example of the source code of the agent's behaviour started by receiving a message. The parts of the code, generated according to the incoming message rules, are highlighted.

```
package agentName;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;

/**
 * to do: description of the behaviour for the javadoc tool.
 */
class MessageFreeFormDescription extends OneShotBehaviour {

  // placeholder for the received message
  private ACLMessage received;

  /** constructor of the behaviour */
  public MessageFreeFormDescription(Agent a, ACLMessage msg) {
    super(a);
    received = msg;
  }

  /** actual implementation of the behaviour */
  public void action() {
    // to do: add message handling code

  }
}

/**
 * Predefined incoming message handler
 */
class MessageHandler extends CyclicBehaviour {

  /** constructor of the behaviour */
  public MessageHandler(Agent a) {
    super(a);
  }

  /** actual implementation of the behaviour */
  public void action() {

    // wait for a message
    ACLMessage received = myAgent.receive();
    if (received == null) {
      block();
    }
    else {

      // check message information to
      // start corresponding behaviour of the agent
```

```
      if ( received.getPerformative() == ACLMessage.MESSAGETYPE &&
      received.getSource().equals("senderagentname") &&
      received.getConversationId().equals("Dialog_Title") &&
      received.getContent().equals("(content from the label)") ) {
        myAgent.addBehaviour (new MessageFreeFormDescription
(myAgent, received));
      }
    }
  }
}
```

Incoming message – inside a behaviour. If an incoming message is not the first message of a behaviour then receiving of the message is implemented as part of the code of the behaviour method "action()". The rules are the following:

– A variable "received" of type ACLMessage is defined in the method "action()" of the behaviour class.

– Method "blockingReceive()" of the class Agent is used to wait for incoming message.

– The free form description of the message is used as source code comment for better readability of the generated code. The comment is placed in front of the method call "blockingReceive()**.**

This approach assumes that there are no other messages sent to the agent than the awaited one. If this is not the case then the model must be redesigned so that the behaviour MessageHandler is waiting for incoming messages.

An **INTERNAL ACTION** is implemented as comment in the code of the "action()" method of the behaviour. Text of the comment is set to the description of the action label.

An **OBJECT** is implemented in JADE as a class definition. The class is used as "wrapper" around the actual object and is serving as communication interface between an agent and the actual object. Since in our agent model we do not define the internal structure of the object, the generated class includes only the description of the interface and has no actual code. The member functions of the class are the ones corresponding to the manipulations in the visual model. The rules are the following:

– The name of the object class is the name of the object in the visual model.

– The new class is located in the same package with the agent that is manipulating the object. Note that the package contains the agent class, it's behaviour classes and classes for objects manipulated by the agent. Design of systems where several agents manipulate the same object should be avoided.

A **MANIPULATION** is mapped in JADE code as part of the behaviour code of the agent and as a function of the object class. The rules are the following:

– A public member function is added to the object class.

– Name of the function is taken from the name of the manipulation in the visual model.

– The return type is implemented according to the return type and the return variable defined in the visual model:

– If no return type and variable are specified then the function is defined as “void”.

– If return type is specified then a new instance of the type is returned. The type should support constructor with no parameters.

– If a return variable is specified but the type is omitted then String is assumed.

– Parameters of the manipulation function are implemented using the type specified in the model. If the type of a parameter is not specified then String is assumed.

Implementation of the manipulation in the agent’s behaviour code is straightforward:

– Define an instance of the class of the manipulated object in the “action()” method of the behaviour code. Use names o1, o2, o3, etc., for naming of the instance variable.

– If the member function requires parameters then these must be defined. If parameter type is not specified then String is assumed. A comment is added to the generated code to remind programmer to assign correct values to the variables.

– Call the member function. If the function returns a value then the value is assigned to a new variable of the corresponding type. If the return type is omitted then String is assumed.

Mapping Summary

The mapping of a diagram onto the actual program code is broken down to the mapping of different agents, objects, messages, and other components included in the diagrams. We have covered all the defined components already and Table 1 summarizes the mapping rules of visual components on the diagrams onto the source code.

Table 1. Summary of the mapping of diagram components onto the source code

Diagram component	Generated source code components
Diagram Title	Used as “conversation ID” of ACL messages
Agent	Agent package; Agent class; MessageHandler behaviour class
Object	Object class
Lifeline	Not implemented in code
Behaviour	Behaviour class
Message	Part of behaviour code of the sender agent; Part of behaviour code of the receiver agent; Part of MessageHandler behaviour code of the receiver agent
Manipulation	Part of behaviour code of the agent; Member function of the object class
Internal action	Comment in the source code of the behaviour method Action()

5. CONCLUSIONS

The paper shows that agent technologies have been developed so far that it is possible to define methods supporting agent-based software development from the analysis through design to the final implementation. The methods introduced above have been tested on a case study of a library lending system. The case study includes the description of the business rules applicable in the lending system, the visual model based on the rules, and the software code based on the model. It is possible to visually follow the exchange of messages between agents in the JADE runtime environment and to see that the pattern matches with the one presented in the visual model.

Our work is based on a limited list of concepts and on sequence diagrams only. Investigations are needed to add more concepts to introduce their notation and implementation rules into the program code. In addition, more visual diagrams from UML and also from elsewhere should be adjusted for representing the business rules in agent-based systems.

REFERENCES

1. Odell, J., Parunak, H., and Bauer, B. Representing agent interaction protocols in UML. In *Proc. AAAI Agents 2000 Conference*. Barcelona. Forthcoming.
2. Bauer, B. *Extending UML for the Specification of Agent Interaction Protocols*. Foundation for Intelligent Physical Agents, Munich, 1999.
3. Nwana, H. S. Software agents: An overview. *Knowl. Eng. Rev.*, 1996, **11**, 205–244.
4. Wagner, G. *Foundations of Knowledge Systems with Applications to Databases and Agents*. Kluwer, Boston, 1998.
5. Finin, T., et al. *Draft Specification of the KQML Agent-Communication Language*. The DARPA Knowledge Sharing Initiative External Interfaces Working Group, Baltimore, 1993.
6. JADE *Online Documentation*. University of Parma, CSELT S.p.A, 2000. http://sharon.cselt.it/projects/jade/.
7. FIPA 97 *Specification, Version 2.0, Part 2, Agent Communication Language*, Foundation for Intelligent Physical Agents, Geneva, 1998. http://www.fipa.org.
8. Taveter, K. and Tamm, B. A new approach to the modeling, design, and implementation of business information systems. In *Proc. Fourth IEEE International Baltic Workshop on DB and IS (BalticDB&IS'2000)*. Vilnius. Forthcoming.
9. *UML Summary, version 1.1*. Rational Software Corporation, Santa Clara, 1997. http://www.rational.com/UML.
10. Hay, D. and Healy, K. A. *GUIDE Business Rules Project, Final Report*. The Business Rules Group, Seattle, 1997. http://businessrulesgroup.org/first_paper/br01c0.htm.
11. *UML Notation Guide, version 1.1*. Rational Software Corporation, Santa Clara, 1997. http://www.rational.com/UML.

AGENDIPÕHISE TARKVARA DISAIN

Margus OJA, Boris TAMM ja Kuldar TAVETER

Artiklis on püütud siduda agendipõhise tarkvara väljatöötamise etappe ühtseks tervikuks, sealjuures on defineeritud agendipõhise tarkvara komponendid, nende visuaalne notatsioon ja reeglid komponentide realiseerimiseks programmikoodis. Agendipõhise tarkvara komponentide defineerimisel on lähtutud mitmete autorite töödest. Komponentide tähistuse väljatöötamisel on kasutatud universaalset modelleerimiskeelt UML. Programmikoodi näited on koostatud Java keeles JADE arenduskeskkonnas. On esitatud kolmetasandilise tarkvara (agenditasand, objektitasand ja binaarne tasand) arenduse mudel. Defineeritud komponendid ja visuaalne mudel kuuluvad agenditasandile, genereeritud Java kood objekti-tasandile. Kolmandat, binaarset tasandit artiklis ei käsitleta. Agendipõhise tark-vara kasutamine on õigustatud kindlalt piiritletud tingimustes, kui tarkvara koosneb autonoomsetest komponentidest.