A UML-based Software Engineering Methodology for Agent Factory

Rem Collier
University College Dublin
Ireland
rem.collier@ucd.ie

Gregory O'Hare
University College Dublin
Ireland
gregory.ohare@ucd.ie

Colm Rooney
University College Dublin
Ireland
colm.rooney@ucd.ie

Abstract

This paper presents the Agent Factory Development Methodology, an Agent-Oriented Software Engineering (AOSE) methodology that employs a synthesis of the Unified Modelling Language (UML) and Agent UML to support the development of multi-agent systems. We illustrate the use of this methodology, through a simple case study and briefly compare it to some other well-known AOSE methodologies.

1 Introduction

With the continuing emergence of the Agent-Oriented paradigm, there is an urgent need to understand how we might best support developers assigned the task of building an agent-oriented application. This need must be addressed from two perspectives: (1) through the creation of software engineering artefacts (methodologies, tools, architectures etc.) that support the development and deployment of these applications, and (2) through the construction of exemplar applications that act as case-studies illustrating best practices for the use of these artefacts.

The work presented in this paper is concerned with the former of these perspectives, namely the creation of software engineering artefacts that support the development and deployment of agent-oriented applications. Specifically, we introduce the a cohesive methodology that supports the design, implementation, and deployment of agent-oriented applications using *Agent Factory* (AF) [3].

Figure 1, presents a schematic of the AF framework, a four-layer framework that combine: a purpose-built agent programming language known as AF-APL, a distributed FIPA-compliant [5] Run-Time Environment, an integrated Development Environment that supports the implementation and debugging of agents written in AF-APL, and finally, a software engineering methodology that defines a structured approach to the use of the lower layers. This framework is implemented in Java.

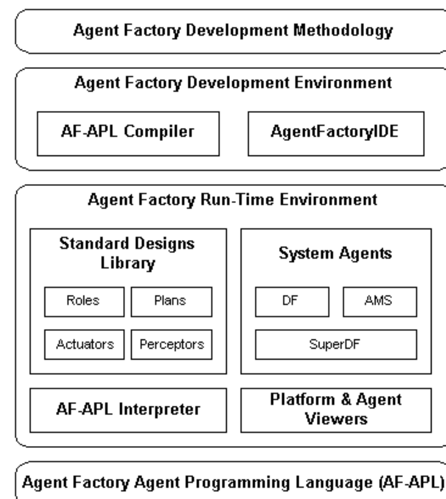

Figure 1. The Agent Factory Framework

To this end, section 2 describes the AF Development Methodology, section 3 highlights the use of this methodology through an exemplar case study of an agent-based internet chat system, and finally, section 4 introduces some related work and presents some concluding remarks. For a more detailed treatment of AF see [3] [4] [11].

2 The Development Methodology

One of the primary objectives behind AF is the development of a cohesive software engineering methodology that delivers structured support for the design, implementation, and deployment of multi-agent systems. In designing this methodology, we sought to address a number of objectives: (1) to employ, where possible, pre-existing industry recognised design notations; (2) to focus upon the definition of visual notations; (3) to use models that promote design reuse; and (4) to maintain a strong link between design and implementation, opening the way for automated code generation. We describe the resultant methodology below.

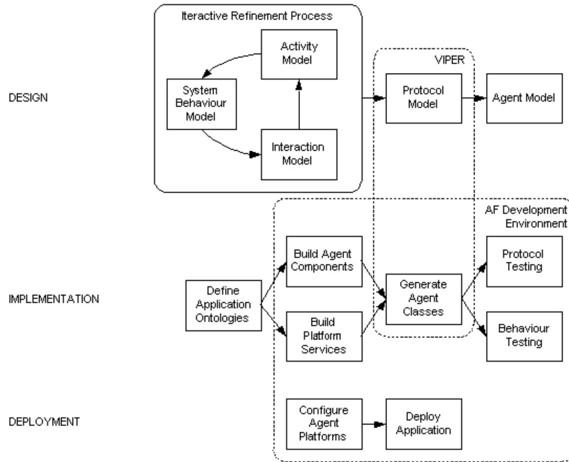

Figure 2. The AF Development Methodology

2.1 The Design Phase

The design phase is concerned with the translation of system requirements into a well defined model of the target system that can be easily implemented using agent technologies. Specifically, we require this methodology to adhere to a number of objectives as set down earlier in this paper. In response to these objectives, we have built the AF Development Methodology around five key models that employ a combination of Unified Modelling Language (UML) and Agent UML [1] diagrams. Figure 2 presents a diagrammatic overview of this methodology.

Central to these models is the notion of an *role*. Within the methodology, roles are used to abstract from fixed agent entities towards the discrete patterns of activity and interaction that will occur in the system. Our rationale for this is that we expect different agents to play the same role during the lifetime of the system. Furthermore, we are able to gain a degree of decoupling between agent implementations and the associated interaction patterns. This allows us to encode best practices for agent interactions as design patterns that may be implemented in a number of ways. This is in line the Foundation for Intelligent Physical Agents (FIPA) efforts at the standardisation of agent interactions (e.g. the Contract Net protocol).

2.1.1 The System Behaviour Model

The first model we develop is the *System Behaviour Model* (SBM). A system behaviour is any distinct set of activities and / or interactions that take place during the operation of the system. For example, in mobile computing system, key system behaviours may include user movement updates, service registration, and service activation.

The principle activity associated with the SBM is the identification of both the key system behaviours and the types of role that the agents will play while engaged in them. In particular, our model distinguishes between two types of system behaviour: (1) *interaction-oriented behaviours* are system behaviours that are associated with two or more roles; and (2) *activity-oriented behaviours* are system behaviours that are associated with a single role.

The SBM is formalised using a customised form of the UML Use Case Diagram. Use Case Diagrams are a well-understood approach for modelling how external actors interact with a software system. From an agent-oriented perspective, we adopt the view of actors as agents that are playing a specific role, and use cases as the behaviours that one associates with these roles . Formally, we customise the UML Use Case Diagram by employing two stereotypes: the <<role>> stereotype, which identifies actors that represent roles that will be played by agents, and the <<role-use-case>> stereotype, which identifies use cases that occur between agents that are engaged in specific roles.

Upon completion of an initial SBM, the various use cases are organised by behaviour type. These behaviours are then analysed further through the *Interaction Model* (section 2.1.2) and the *Activity Model* (section 2.1.3).

2.1.2 The Interaction Model

The Interaction Model (IM) expands on the SBM through the analysis of the interactions that will occur within each of the interaction-oriented system behaviour. Specifically, for each use case, we identify a number of *interaction scenarios*, each of which describes a potential set of interactions that may take place. Typically, each use case is with one standard scenario (i.e. the one in which everything works out well), together with a number of alternate scenarios that describe variations on the standard scenario. For each scenario, we identify the types of message that the agents send to one another, while playing a role, and the order in which the agents send them.

Given the objectives to employ, where possible, visual design notations, we use a customisation of UML Collaboration Diagrams, similar to that described in [9], to represent individual interaction scenarios. Specifically, we introduce two stereotypes: the <<role>> stereotype, which identifies the objects that represent the agents that are playing specific roles; and the <<fipa-acl>> stereotype, which constrains the valid message types to the FIPA ACL performatives specified in the FIPA 2000 standards [5].

The purpose of this model is to support the expansion of the initial SBM. As such, it is possible (and in fact expected) that this stage of the process will highlight deficiencies in the initial SBM. Consequently, we expect that the initial IM will be iteratively refined as the analysis progresses. How-

ever, it is expected this model will ultimately reach a level of stability, at which point, it is transformed into a set of protocols within the *Protocol Model* (section 2.1.4).

2.1.3 The Activity Model

The Interaction Model (section 2.1.2) facilitates the expansion of the scenarios that underpin interaction-oriented system behaviours. While this type of behaviour is predominant within agent-oriented applications, it is not the only type of behaviour, there are also activity-oriented system behaviours. To cater activity-oriented behaviours, a third model, known as the *Activity Model* (ActM), is introduced. In contrast with the IM, this model focuses on the activities that underpin the system behaviours.

The ActM is underpinned by the concept of an *activity scenario*. In contrast with interaction scenarios, activity scenarios focus on the activities that the various agents perform while playing associated roles. Initially, this type of scenario was devised for single agent behaviours. However, it has also proved valuable when analysing certain activity intensive multi-agent behaviours. Consequently, the ActM contains at least one activity scenario for each activity-oriented system behaviour, and zero or more activity scenarios for each interaction-oriented system behaviour. In addition, each system behaviour may be associated with multiple activity scenarios. This is because there may be a number of ways that a given behaviour can be realised.

We formalise the ActM through the customisation of UML Activity Diagrams in a fashion similar to that presented in [9]. Specifically, we customise this diagram through the introduction of a <<role>> stereotype, which associates swimlanes with roles.

Again, it is expected that, during the formation of the ActM, various deficiencies in the current design will be identified. As a result, the model will be iteratively refined as the analysis progresses. However, once the model reaches a suitable level of stability, work on the ActM ceases, and the final model becomes an input into the Agent Model (section 2.1.5).

2.1.4 The Protocol Model

Once the first three design models have reached a suitable level of stability, the design process switches from identifying roles, interactions, and activities to their formalisation through two models: the *Protocol Model* (PM) and the *Agent Model*. This section describes the former model.

The PM represents a formalisation of the IM (section 2.1.2). Specifically, the PM refines the various interaction scenarios into a set of protocols that describe how the agents will interact, and which encapsulates each of the alternate scenarios associated with a given system behaviour. Each

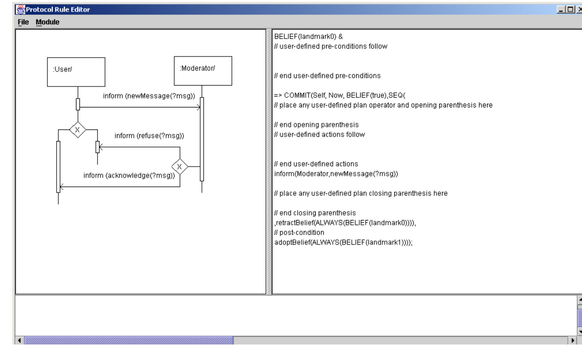

Figure 3. VIPER & Send Message Protocol

protocol specified within this model is defined using Agent UML Sequence Diagrams [1].

It is expected that each interaction-oriented system behaviour will map onto exactly one primary sequence diagram. However, this diagram may itself may onto a number of secondary sequence diagrams that arise from a number of additional refinement activities that can (optionally) take place during the formulation of the model. These activities include: (1) identification and extraction of common interactions within the protocols, the reformulation of these common interactions as *template protocols*; and (2) re-factoring of protocols to make use of any appropriate template protocols, including both those extracted through the previous activity and any known agent design patterns (e.g. Contract Net, Dutch Auction, etc.).

One side effect of this re-factoring is that some re-factored protocols may be linked to system behaviours that have associated activity scenarios. It is vital that the designer be aware of such dependencies, and must re-factor these scenarios to reflect the updated protocols.

Tool-based support for protocol creation is provided via the VIPER [11] visual protocol editor (see figure 3). VIPER performs two jobs within this methodology: (1) it supports visual editing of Agent UML Sequence Diagrams, and (2) it automatically generates agent code based upon these diagrams.

2.1.5 The Agent Model

The Agent Model (AgtM) switches the focus of the design process from the role, interactions, and activities that are necessary to deliver system behaviours to the agents that will exist in the deployed system. Specifically, this model moves the design from a behaviour-centric view to an agent-centric view of the system.

Within this alternate view of the system, we focus upon two concepts: roles, and *agent classes*. Specifically, agent classes represent the types of agents that will be deployed

in the final system, and consequently how the various roles will be *implemented*. We allow a many-to-many correspondence between roles and agent classes. That is, we allow each agent class to be associated with many roles, and each role to be associated with many classes.

To achieve this change of view, we perform three steps: (1) we list each of the roles specified in the SBM, and for each role, list the associated protocols; (2) we associate each role with one or more *agent classes*; and finally, (3) we relate each agent class to the associated set of activities specified within the Activity Model.

Perhaps the most subjective aspect of the AgtM is the selection of which activities to associate with a given agent class. As stated in section 2.1.3, a system behaviour can be associated with a number of activity scenarios. This one-to-many relationship occurs in recognition of the possibly that a system behaviour can be realized in a number of ways (two obvious alternatives are direct realisation versus delegation). The designer must choose which of the potential activity scenarios a given agent class should employ when realising the corresponding system behaviour.

This final model is formalised using a UML Class Diagram that has been customized to include: a <<role>> stereotype and an <agent-class>> stereotype. The <<role>> stereotype represents roles, and takes the form of a box that contains two compartments: the first compartment contains the stereotype followed by the role identifier, and the second compartment contains a list of protocol identifiers. Conversely, the <<agent-class>> stereotype represents agent classes, and takes the form of a box that contains three compartments: the first compartment contains the stereotype followed by the agent class identifier, the second compartment contains a list of protocols (not these specified in the associated roles), and the third compartment contains a list of activity identifiers.

2.2 The Implementation Phase

The second phase of the AF Development Methodology (see figre 2) is the implementation phase. This phase contrasts significantly with the earlier design phase, in that it is tied closely to a specific agent development framework, while the design phase is not.

Central to the implementation phase is the fabrication of a set of agent classes. These classes are implemented in AF-APL [3], an Agent-Oriented Programming language in which agents are mental entities that are modelled using mental attitudes, in this case: *beliefs* and *commitments*. Beliefs describe, using a first-order logic representation language, the current state of the agent and its environment, and commitments describe the current (and future) activities that the agent has decided to perform. Finally, decisions are modelled through a set of commitment rules that map situations (a conjunction of positive and negative beliefs) onto commitments. These rules are checked repeatedly within a sense-deliberate-act cycle.

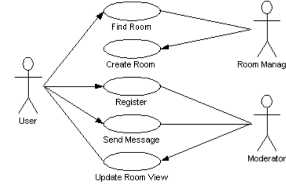

Figure 4. ChatterBot System Behaviour Model

The beliefs of an agent form an internal model of both itself and its environment. Agent interaction can often involve an agent communicating information held within these beliefs to another agent. To ensure that these internal models are coherent, and to facilitate information dissemination between agents, the process commences with the definition of any application-specific *ontologies*. Within AF, an ontology is realised as a well defined mapping of logical predicates to target domain relations. For example, a mobile computing ontology may specify that the predicate position(?lat, ?long) represents a users position in latitude and longitude.

Upon completion of an initial set of ontologies, the next activity to be carried out is the generation of any custom *agent components* required by the final system. The principle (but not only) agent components employed within AF are perceptor and actuator units. Perceptor units are Java classes that encapsulate specific sensing abilities (e.g. monitor the users location), and which, convert raw data into beliefs. Conversely, actuator units are Java classes that encapsulate specific primitive abilities that an agent may directly execute (e.g. update user profile). The developer identifies potential perceptor and actuator units by reviewing the activities specified in the Activity Model.

In tandem with the construction of agent components, the developer also builds any custom *platform services* required by the final system. A platform service is a service that is deployed on a agent platform as specified by the FIPA-standards [5]. Typical platform services include: message transport, migration, and persistence services.

Once the agent components and platform services have been constructed,the final development activity is started, namely, the generation of AF-APL code that implements the agent classes specified in the Agent Model. In particular, AF realises agent classes as text files, entitled role files, that uses a ”rle” extension, and which contain AF-APL code. Reuse of AF-APL code is supported through a USE_ROLE construct similar to the #include construct of C. When converting our design into AF-APL code, we assume a one-to-

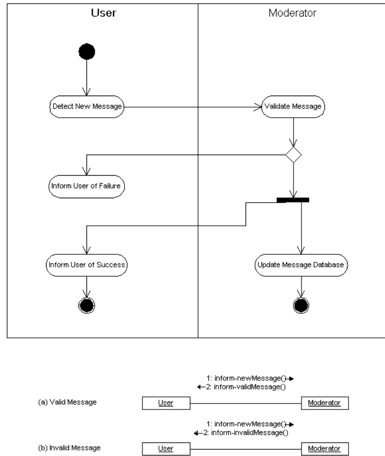

Figure 5. Activity and Interaction Scenarios

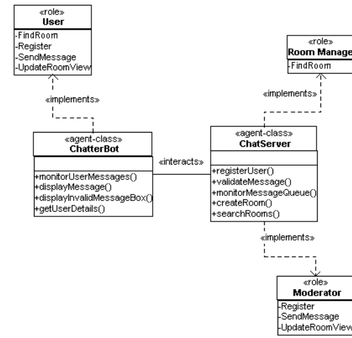

Figure 6. The ChatterBot Agent Model

one mapping from agent-classes to role files (i.e. role files do not map onto roles as specified in the design, although ongoing work aims to redress this imbalance). Each role file identifies a set of actuators and perceptors that correspond to the specified activities, and a number of commitment rules that describe how those activities should be used.

Finally, the implementation phase concludes with the testing of the implementation. Two types of testing are undertaken: protocol tests evaluate the correctness of the agent interaction protocols, and behaviour tests evaluate the correctness of specific agent behaviours (i.e. that in a given situation, the agent does the expected set of tasks). Once testing is complete, the implementation phase draws to a close, and the system is deployed.

3 Case Study

To illustrate our development methodology, we now present the ChatterBot case study. ChatterBot is an agent-based internet chat system, the basic premise of which is that users are represented by an interface agent that registers the user with a particular chat room, sends messages to that room on behalf of the user, and views any messages posted to the room by other users. When analyzing this system, three key roles were identified: the User role, which is responsible for sending and viewing messages, the Moderator role, that is responsible for validating messages associated with a given room, and the Room Manager role, which is responsible for creating new rooms and managing existing rooms. We formalise this analysis within the System Behaviour Model (SBM) as illustrated in figure 4.

The second phase involves the expansion of each system behaviour specified in the SBM first within context of the Interaction Model, and then later within the context of the Activity Model. the of the interaction to build collaboration diagrams for each of the tasks that describe the interactions that should take place between the participants.

By way of illustration, we expand upon the Send Message system behaviour. Our initial expansion of this behaviour is through the Interaction Model. Specifically, we define two interaction scenarios for this behaviour: namely where the User role sends a message that the Moderator role considers to be valid, and where the User role sends a message that the Moderator role considers to be invalid. Figure 5 contains a visual representation of these scenarios.

To help to clarify how the two interaction scenarios arise, we further expand upon the Send Message system behaviour through the Activity Model. Figure 5 presents activity scenario that results from this second expansion. Informally, this scenario indicates that the User role detects a new message has been entered, informs the Moderator role of the new message. The Moderator then validates the message and informs the User role of the success or failure of the validation activity.

As the System Behaviour, Interaction, and Activity models become more stable (i.e. as a general agreement on how the system behaviour emerges), work on these three models stops, and the designer commences work on the Protocol Model. Specifically, the PM is a formalisation of the IM, in which the various interaction scenarios are collapsed into a single protocol. This process is illustrated in figure 3, which presents the Send Message protocol. This protocol is based on the interaction scenarios described in figure 5.

Upon completion of the PM, the final step of the design process involves the formulation of the Agent Model. This final model switches from a behaviour-oriented view of the system to an agent-oriented view of the system. Specifically, this model presents a view of the roles within the system, and the set of agent classes that will implement those roles (section 2.1.5). Figure 6 presents the agent model for ChatterBox. As can be seen in this model, the three roles are

implemented via two agent classes: the ChatterBox class implements the User role, and the ChatServer class implements the Moderator and Room Manager roles.

At this point, the design of ChatterBox is complete and all that remains is for the developers to implement the system. As indicuated in section 2.2, this involves the design of a number of agent components and the implementation of the ChatterBox and ChatServer agent classes within AF-APL. Further details of this process are not described here.

4 Discussion and Conclusion

The methodology presented in this paper represents one of a number of potential approaches to fabricating multi-agent systems. Gaia [12] is another methodology, which supports the analysis and design phases. Gaia differs from the our methodology in that it is primarily form-based (i.e. not a visual methodology), employs a non-standard design notation, and does not provide any support for the implementation of the designs it produces. In contrast, the MESSAGE methodology [2] does employ a visual design notation, and there is tool-based support for the implementation of multi-agent system. However, while MESSAGE employs the same meta-modelling language as UML, MESSAGE diagrams bear little resemblance to UML diagrams, which are well understood within the software industry. A third methodology is that outlined in [6]. Like our methodology, Heinze's methodology does employ UML use cases and activity diagrams, and does specify the link between design and implementation. However, Heinze's methodology does not account for multi-agent interactions, nor is there any tool-based support for the development process.

In summary, this paper presents a visual agent-oriented software engineering methodology that is founded upon the industry standard UML design notation. Furthermore, the models underpinning this methodology facilitate both design reuse, and automated partial code generation. Specifically, partial tool-based support for the methodology currently exists, in the form of VIPER, a visual protocol editor [11]. Finally, due to space constraints, this paper has illustrated our methodology through a trivial case study. However, this methodology has been employed in the development a number of large scale agent-based applications including the WAY System [7], Gulliver's Genie [10] and the award-winning ACCESS Architecture [8].

References

[1] B. Bauer, J. P. Muller, and J. Odell. Agent uml: A formalism for specifying multiagent interaction. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*. Springer Verlag, 2001.

[2] G. Caire, W. Coulier, F. J. Garijo, J. Gomez, J. Pavon, F. Leal, P. Chainho, P. E. Kearney, J. Stark, R. Evans, and P. Massonet. Agent oriented analysis using message/UML. In *AOSE*, pages 119–135, 2001.

[3] R. W. Collier. *Agent Factory: A Framework for the Engineering of Agent-Oriented Applications*. PhD thesis, Department of Computer Science, University College Dublin, 2001.

[4] R. W. Collier, G. M. P. O'Hare, T. D. Lowen, and C. F. B. Rooney. Beyond prototyping in the factory of agents. In *Proceedings of 3rd International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS 2003)*, Prague, Czech Republic, 2003.

[5] FIPA. Fipa 2000 standards. URL: http://www.fipa.org.

[6] C. Heinze, M. Papasimeon, and S. Goss. Specifying agent behaviour with use cases. In *Proceedings of the Pacific Rim International Workshop on Multi-Agent Systems (PRIMA 2000)*, pages 128–142, 2000.

[7] T. Lowen, G. O'Hare, and P. O'Hare. Mobile agents point the way: Context sensitive service delivery through mobile lightweight agents. In C. Castelfranchi and W. Johnson, editors, *Proceedings of First International Joint Conference on Autonomous Agents and Multi-Agent Systems Conference (AAMAS 2002)*, Bologna, Italy, 2002. AAAI Publishers.

[8] C. Muldoon, G. OHare, D. Phelan, R. Strahan, and R. Collier. Access: An agent architecture for ubiquitous service delivery. In M. Klusch, A. Omicini, S. Ossowski, and H. Laamanen, editors, *Cooperative Information Agents VII*, LNAI 2782. Springer Verlag, 2003.

[9] J. Odell, H. V. D. Parunack, and B. Bauer. Extending uml for agents. In *Proceedings of AOIS Workshop at AAAI 2000*, 2000.

[10] G. O'Hare and M. O'Grady. Gulliver's genie: A multi-agent system for ubiquitous and intelligent content delivery. *Computer Communications*, 26(11):1178–1187, 2003.

[11] C. F. B. Rooney, R. W. Collier, and G. M. P. O'Hare. Viper: Visual protocol editor. In *Proceedings of COORDINATION 2004*, Pisa, Italy, 2004.

[12] M. Wooldridge, N. R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.