A Use-Case Based BDI Agent Software Development Process

Jeffery M. Einhorn[1] and Chang-Hyun Jo[2]

[1] Department of Computer Science, University of North Dakota
Grand Forks, ND 58202-9015, USA
jeinhorn@cs.und.edu

[2] Department of Computer Science, California State University Fullerton
Fullerton, CA 92834-6870, USA
jo@ecs.fullerton.edu

Abstract. As computer software continues to grow increasingly complex with each passing year, researchers continue to try and develop means to simplify software development. In this paper, we propose a BDI agent software development process as the next evolution in software development. The goal of this research is to develop a process, which can be used to enable the creation of agent-based systems. This paper strives to present a practical software development process, which is useful to today's software engineer, by building upon current agent research and proven software engineering practices. Our BDI agent software development process is a systematic process, which enables the decomposition of a system into agents. The Belief-Desire-Intention Model is a fundamental ingredient to our development process. We utilize BDI as a natural method for describing agents in our development process. Our software development process utilizes several forms of use cases, which are useful for defining the architecture of a system in our process. We have also leveraged many other existing software development tools such as CRC cards, patterns and the Unified Development Process. We have made modifications to many of these existing tools so they can be used for agent-based development. Basically, our BDI agent software development process strives to model both the dynamic and static structure of the agents that make up the system. Once we have modeled the structure, which makes up the agents in the system the structure can then be created in software.

1 Introduction

Today we stand on the horizon of the next generation of software development methodologies. Agent-based software engineering will provide the next step forward in the effort to provide better tools for developing software that must meet the increasing demands, expectations and changes from customers [12] [24]. Agent-based software engineering decomposes a system into agents. These agents have control over both their state and behavior. Systems will contain many agents that can cooperate with other agents to provide the system's functionality. The research in the following sections will describe our new agent-based software development process.

The goal of this paper is to provide a software development process that software developers can use as a tool for constructing agent-based systems. Much work has been done on theory, but many of the theoretic approaches do not provide enough consideration to the desires of the software developers that will use this technology. In our approach we seek to balance the needs of the software developer with a solid approach to building agent-based systems.

--
Einhorn, Jeffery, M. and Chang-Hyun Jo, *A Use-Case Based BDI Agent Software Development Process*, Proc. of the 2nd International Workshop on Agent-Oriented Methodologies - OOPSLA-2003, Anaheim, CA, USA, 7-20, Oct. 26-30, 2003.

We propose a systematic agent-based software development process that is natural for software engineers to use. We will explain each step of our agent-based development process and provide an example of its use. It is our belief that this research is easier to understand if we provide real applications after we discuss the theory behind what should be done. This allows a software engineer to understand each step before moving on to the next step. The case study example is shown in the Appendix.

2 Background

There has been much debate on the definition of an agent or even an intelligent agent. The simplest definitions of an agent usually are described as an object with a goal or an entity that acts upon the environment it exists in [26]. Wooldridge and Jennings describe agents as having autonomy, proactiveness, reactivity and social ability [23]. In our research an agent-based system is a system that is made up of agents defined by a set of beliefs, desires and intentions (BDI) [5, 14, 21]. In our research entities become agents when we can assign beliefs, desires and intentions to them.

In the analysis phase of our agent-based software development process will strive to discover potential agents and the BDI's that make up our system. Defining the BDI's does border on design instead of analysis because we are describing how something will be done. In the design phase of our agent-based software development process we will assign the BDI's to software agents.

Our development process builds upon successful strategies that can be found in object-oriented development. We propose new methods for use in agent-based software development whenever previous tools found in other development processes such as UML, the Unified Development Process [4, 18] and use cases [6] prove inadequate for agent-based development.

Use cases are another tool that will be fundamental to our agent-based software development process. Use cases are a proven tool that helps drive the development process forward and helps capture the requirements of a system [6]. Use cases provide a functional approach to gathering requirements [18]. Jennings also supports functional analysis by describing it as more natural than data or object type analysis [13]. The functional approach will also be useful when building agent-based systems because it is necessary to gather requirements for agent-based systems. We will use a modified use case called an external use case for discovering the functions or services that our systems should provide. We will also use another kind of use case called an internal use case for identifying plans (intentions), goals (desires), and their beliefs from the system services discovered from the external use case.

In our agent-based development process we will first identify the services that our system should provide. The system can be thought of as an agent, since we will describe our entire system as an encapsulated entity, which will have state and behavior. After we have identified the services that our system will provide we can then identify the goals that are necessary to provide each service. Identifying the proper goals and assigning them to agents becomes a major focus of the agent-based software development process.

An agent's beliefs correspond to the knowledge an agent has about its environment. The desires of an agent can be described as the goals an agent can choose to achieve. An agent's intentions are the plans that will allow the fulfillment of a goal. In our agent-based software development process we define an agent as an entity that we can assign BDI to. In our development process we will identify the possible agents and the goals that will provide the system's functionality. In the process of discovering goals we will also assign beliefs and intentions to each goal. We define the software agents in the system as we assign BDI to candidate agents.

In studying the research that was been done in the area of agent software systems we have found two general types of works to be useful. The first is the research that has been done to solve problems from a software engineering perspective. Research into such tools as CRC cards [2], UML diagrams [8], use cases [6] and software patterns [9] [18] have been invaluable for use in constructing object-oriented systems [3]. Agent UML is one of the pioneer work in extending UML for agent development [19]. In our research we have modified several of the tools that have proved successful for object-oriented software construction for use in agent-based software systems.

The second area of research is in the agent theory. Jennings, Wooldridge and others [23, 11, 24, 25, 26, 7, 13, 14, 20] provide research into the theory of agent software development. Depke et al. [7] takes the approach of describing a system using roles. It provides a brief description of a development process based on roles. They also make the necessary additions to UML in order to provide diagrams to describe their process. Wooldridge, Jennings and Kinny [25] also talk about defining a system based upon its organization. They state by looking at the roles played by agents in the system you can then model the system based upon those roles. Instead of focusing on roles we focus on describing the beliefs, desires and intentions for each agent.

Michael Wooldridge has published several research documents on agent theory and agent software development techniques [23, 24, 25, 26]. Wooldridge presents a detailed BDI architecture, which is designed for building BDI agent systems [26]. Rao and Georgeff [21] provide a paper describing a BDI architecture for use in building agents. They formalize their work using BDI logic and provide a model that can be used to describe BDI agents. Their research provides us with a better understanding of BDI and how it can be formally described. One of the most related works we found was Kinny's work [16, 17]. They similarly view the agent system from external viewpoint and internal viewpoint. Their work describes the framework necessary for agent-based modeling very well. However, our modeling views and captures external view and internal view using our own coherent methods and techniques. Our technique using two different kinds of use cases provides a very systematic way to develop BDI agent-based systems.

3 BDI Agent Development Process

3.1 Brief Process Description

A salient point in our research is the use of BDI [21] for describing agents. BDI provides us with a clear view of what makes up an agent. We will assign beliefs, desires and intentions to each agent. Our process will provide the tools that will be necessary to systematically build agent-based software systems.

Figure 1 provides a high level view of how we will use BDI in our agent-based development process. Figure 1 describes a general approach of how an agent BDI attributes are discovered in our BDI agent software development process. In the beginning of our development process we use external use cases, which are general plans indicating how a specific service can be provided from an external point of view. We then refine these plans into goals using internal use cases. The internal use cases decompose a service into one or more goals. In addition the internal use cases also provide a more precise description of each goal and its corresponding plan. After we have discovered a goal and described a plan for each goal we need to discover the beliefs that will be necessary for each goal to be completed. The beliefs are determined for each goal by analyzing each goal's plans and determining what beliefs will be necessary for its completion. Now that we have described a complete BDI we can assign it to an agent. Before we discuss each step of our development process in detail it is useful to take a high level view of the entire BDI agent software development process. Our process stresses a goal-oriented approach for developing agent-based systems. Use cases play an important role in discovering the goals that will be necessary to provide the services for our system.

Figure 2 is a diagram of the artifacts that will be created during our BDI agent development process. The arrows show the general order of creation for the artifacts in our process. It is important to understand that the artifacts can be created in any order that is useful to the developer. The arrows represent a loose order that we suggest for artifact creation at this time. The BDI agent software development process begins with the initial problem statement, which describes what the system should do. Next we create the enterprise software assessment in order to discover how this problem fits within the scope of the current enterprise. Then the brief external use cases are created to define the services that the system should provide. At this point in the process we have defined, from an external point of view, the services that our system is required to provide. External use cases present scenarios that provide plans, from an external viewpoint, that will provide the services for our system. By analyzing the artifacts we have created so far we create the conceptual agent list. Next an agent relation diagram is created for any external use cases that may provide insight into how a conceptual system might work. By looking at our conceptual agent list and our agent relation diagram we can apply agent patterns to create the candidate agent list. During this phase of development we will discover a majority of the possible agents that our system can be decomposed into. At this point in the development process we are beginning to shift from analysis to design.

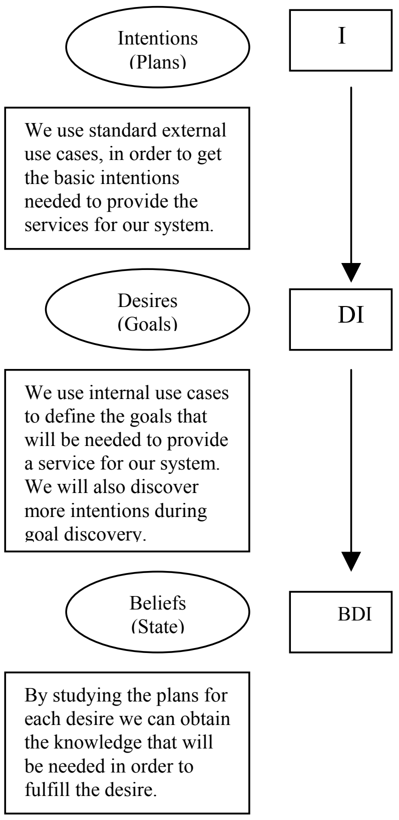

Intentions (Plans)
I
We use standard external use cases, in order to get the basic intentions needed to provide the services for our system.
Desires (Goals)
DI
We use internal use cases to define the goals that will be needed to provide a service for our system. We will also discover more intentions during goal discovery.
Beliefs (State)
BDI
By studying the plans for each desire we can obtain the knowledge that will be needed in order to fulfill the desire.

Figure 1. BDI Discovery in our BDI Agent Model

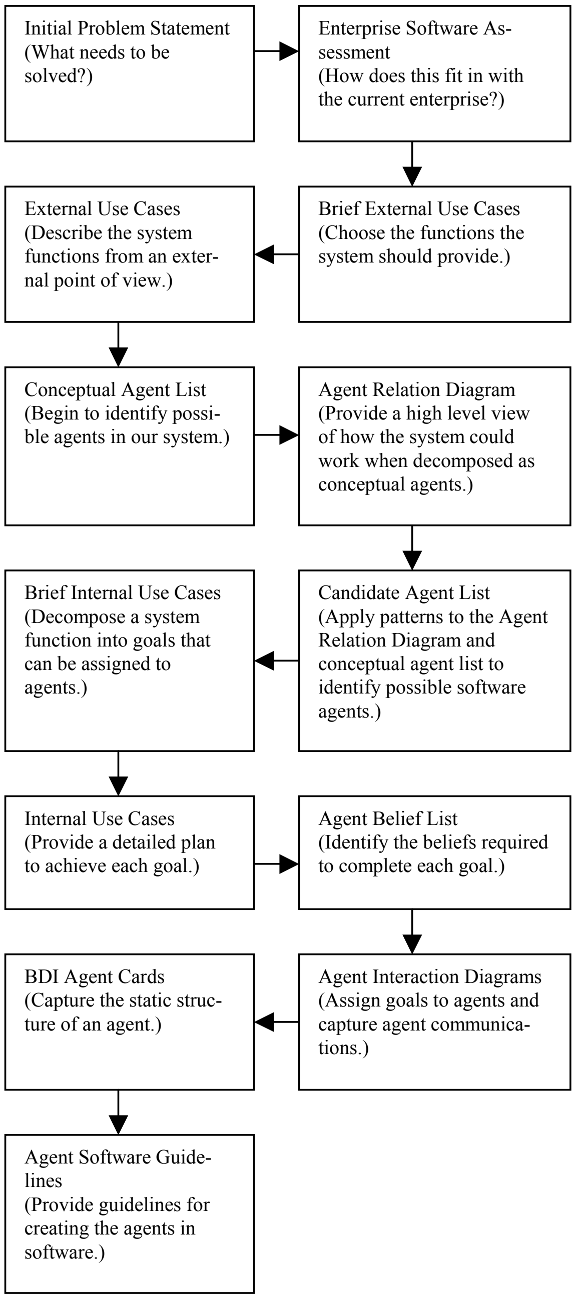

Initial Problem Statement (What needs to be solved?)
Enterprise Software Assessment (How does this fit in with the current enterprise?)
Brief External Use Cases (Choose the functions the system should provide.)
External Use Cases (Describe the system functions from an external point of view.)
Conceptual Agent List (Begin to identify possible agents in our system.)
Agent Relation Diagram (Provide a high level view of how the system could work when decomposed as conceptual agents.)
Candidate Agent List (Apply patterns to the Agent Relation Diagram and conceptual agent list to identify possible software agents.)
Brief Internal Use Cases (Decompose a system function into goals that can be assigned to agents.)
Internal Use Cases (Provide a detailed plan to achieve each goal.)
Agent Belief List (Identify the beliefs required to complete each goal.)
Agent Interaction Diagrams (Assign goals to agents and capture agent communications.)
BDI Agent Cards (Capture the static structure of an agent.)
Agent Software Guidelines (Provide guidelines for creating the agents in software.)

Figure 2. BDI Agent Development Process

--
Einhorn, Jeffery, M. and Chang-Hyun Jo, *A Use-Case Based BDI Agent Software Development Process*, Proc. of the 2nd International Workshop on Agent-Oriented Methodologies - OOPSLA-2003, Anaheim, CA, USA, 7-20, Oct. 26-30, 2003.

The brief internal use cases decompose a system service into one or more goals that can then be assigned to agents. Next we create the internal use cases, which provide a detailed plan for achieving each goal. The agent belief list is created by studying each goal's plan and identifying the beliefs that will be required in order for that plan to be completed. At this point in our development process we have described each belief, desire and intention in detail.

The final phase in our development process centers on describing the agents in such a way that they can be created in software. The assignment of each BDI to agents becomes a fundamental activity during this phase. Agent interaction diagrams are created to facilitate the assignment of each BDI to agents in our system. It is important to note that many agent interaction diagrams may be created before we decide upon a BDI agent relationship. During the creation of the agent interaction diagrams we may discover new insight into our system, which may require us to modify the internal use cases to reflect this new understanding.

The agent interaction diagrams are a useful tool for describing the dynamic structure or communication that takes place between agents. BDI agent cards are created to capture the static structure for each agent. The BDI agent cards and the agent interaction diagrams can be created in parallel. It is imperative that both the BDI agent cards and the agent interaction diagrams share a consistent architecture of the system. Thus a major change in the agent interaction diagrams can often lead to a major change in the corresponding BDI agent cards and vice versa. We have now created the artifacts that will be used to create the agents in software. The actual creation of the agents in software is beyond the scope of this current research, but represents an interesting problem to study in the future.

3.2 Initial Problem Statement

The initial problem statement is the first step in our BDI agent software development process. The initial problem statement describes the problem that needs to be solved by the system. Software developers should create the initial problem statement based upon talking with the customer and by reading any documents describing the problem. Both the customer and the developer should discuss the initial problem statement together. It is important that both customers and developers agree upon a high level view of the system. Customer and developer agreement, upon the initial problem statement, provides a solid start to the development process. Without customer and developer agreement, the system may not meet the needs of the customer and the system will be doomed to failure from the start. The proper creation of the initial problem statement will help the developer understand exactly what the system should do from the customer's standpoint.

3.3 Enterprise Software Assessment

The enterprise software assessment provides a brief overview of how a possible solution would fit in with an enterprise's current systems. We should ask ourselves two different questions during the enterprise software assessment. First we should ask what environment would the system most likely be deployed in. Then we should ask how could we leverage existing software to aid in the development of the system.

The enterprise software assessment addresses the issues of what environment the system will be deployed in and suggests how existing software could be leveraged in the creation of a new system. By looking at the previous issues we gain a better idea of how the system might be implemented for the customer.

3.4 Brief External Use Cases

The brief external use cases are used to identify the services that our system should provide. The creation of the brief external use cases is a valuable tool that can be used in the creation of the external use cases. We will most likely create many brief external use cases before we decide on the ones that best represent the services that our system should provide from an external point of view. When we talk about an external point of view, we mean that we consider the system as a black box and we focus on the users, also known external agents, interaction with this encapsulated system.

3.5 Detailed External Use Cases

We will use a drill down approach throughout our development process as we create artifacts to provide detailed descriptions of previous artifacts. The drill down approach allows us to manage the complexity of a system by focusing on a small number of items at a time. When we need to look at a potentially complex problem like all of the services our system should provide, we use a simple artifact like the brief external use case, which lets us focus on what services our system should provide instead of the details on how such services should work. The external use cases then focus on one service at a time providing a detailed description, from an external viewpoint, of how a service could be provided.

In the brief external use case section we decided what services our system should provide and we gave a brief description of each service. Now we must explore each service in more detail in order to discover a plan that would provide the service. When choosing a service to analyze in greater detail it is important to consider a variety of items. The following two sentences are an example of the types of questions we found useful when deciding which services to analyze. “Is this service critical to the success of the system?” “Will this service help us understand any architectural requirements that our system might have?” We want to focus on the difficult and important services early in the development process. Focusing on the complex services allows us to identify major stumbling blocks early in the development process. It is a well-accepted software engineering view that it is easier to make major architecture changes during the beginning phases of software development.

3.6 Conceptual Agent List

After we finish creating an external use case we should update our conceptual agent list. If this is our first external use case then we will need to create a conceptual agent list. We find conceptual agents by using linguistic analysis. We identify the nouns and words that could possibly be used as nouns, in our written artifacts for the system. The artifacts include any external use cases, brief external use cases, enterprise software assessments, initial problem statements and any other documents describing the system.

Our conceptual list will contain many nouns that may not be agents. They may be only objects or nothing at all. The simplest definition of an agent is an object with a goal. In our software development process conceptual agents only become agents when they are assigned BDI. While we are only interested in decomposing the system into agents, it is useful to identify all the possible nouns, since they will prove useful in providing a starting point for the possible agents that may participate in the system. It is also very likely that we may undercover new agents during the development process. Whenever we find a new conceptual agent we should add it our list.

3.7 Conceptual Agent Relation Diagram

The conceptual agent relation diagram provides a conceptual view of how a service might be provided for a system. The conceptual agent relation diagram shows some conceptual agents and their possible relationships with each other in the system. Agents that are represented by an oval are external to a system and agents represented in a rectangle are internal agents. The arrows in the diagram describe the direction of communication and the label near the arrows indicates the goal of the communication. The key purpose of the conceptual agent relation diagram is to give the developer insight into what some of the internal agents might be. However, the conceptual agents do not directly map to the software agents of the system. By looking at the conceptual agent diagram and by applying the agent software patterns we can create an updated conceptual agent list, which we will call the candidate agent list.

3.8 Agent Patterns

Agent patterns help leverage previous experience in system development and apply it to our current system. Our agent patterns will build upon the object-oriented design pattern work done by Gang of Four [9] and Larman [18]. We have also looked at agent design patterns by Hayden, Carrick and Yang [10] and Aridor and Lange (1998). We will talk about three general types of agent patterns.

The first category of agent patterns is *agent identification patterns*. Agent identification patterns help us discover what additional software agents the system may require in addition to the conceptual agents that we have already discovered. The second category of agent patterns is the *agent creational pattern*. The creational patterns are based upon the creational patterns discussed by the Gang of Four [9]. These agent creational patterns help provide guidelines for who should create the agents. The last type of agent pattern is the *agent goal assignment patterns*. The agent goal assignment patterns will provide guidelines for assigning goals to agents.

In proposing agent identification patterns we try to identify common agents that will be found in agent software systems. We will talk about three different agent identification patterns. The first agent identification pattern is the *manager agent pattern*. The manager agent proposes that we abstract the interaction between internal and external agents to a single manager agent. Large software systems may use several manager agents to communicate with the different subsystems that make up the system. When a system has multiple manager agents it is often useful to create a special kind of manager agent know as a delegation agent. The delegation agent simply receives external agents' messages and forwards them to the proper internal manager agent. The second identification pattern is the *service pattern*. The service pattern provides an agent to represent a certain kind of service that needs to be available to the entire system. An example of a service pattern might be a system, which needs to provide database access to the internal agents in the system. Since several different agents need access to the database we can create a single database agent to provide database service to the internal agents. The third type of identification agent we would like to talk about is the *broker pattern* [10]. The broker pattern suggests a broker agent can be used to abstract a service from the agent that provides that service. If we have several different agents that provide similar services they can register their services with the broker agent. Agents that want to use those services then communicate with the broker agent. The broker agent will choose the proper service agent to handle the agent's request.

Agent creational patterns suggest who should create an agent. The first agent creational pattern is the *long-lived agent*. The long-lived agent needs to be available whenever the system is running. The long-lived agent should be created when the system is started and shutdown when the system quits. The following creational patterns are based upon the design pattern work done by Larman [18]. An agent A should be created by agent B if agent A is the only used by agent B. An agent A should be created by B if agent B has agent A's initialization data. Agent B should create agent A if it aggregates, contains, records or closely uses agent A.

The last category of agent patterns that we will describe is agent *goal assignment patterns*. One could argue that agent creation patterns are really agent goal assignment patterns because we are describing who has the responsibility to create an agent. However, creation of agents is such an important and difficult step we feel that it deserves its own classification of patterns and we have separated agent creational patterns from agent goal patterns. The following agent goal assignment patterns are based upon the design patterns described by Larman [18]. The low coupling pattern suggests that we should try to assign goals so coupling between agents remains low. By keeping the coupling between agents low, we increase the self-sufficiency of the agents, which reduces the complexity of the system by abstracting behavior into a single agent. It is also desirable for our agent's goals to exhibit high cohesion. Our agent's goals should be similar in nature. An agent with vastly different goals can be a sign of an overly complex agent and may indicate the need for the agent to be decomposed into several smaller agents. The notifier pattern (Aridor and Lange 1998) suggests that it is common for agents to ask other agents to notify them of events. The notifier agent will watch for a certain events and notify the proper agents.

3.9 Candidate Agent List

The candidate agent list contains all the potential agents that we can use in designing our system. We may do several iterations of creating an agent relation diagram, applying agent identification patterns and then creating brief internal use cases, in order to discover the possible agents for the system. It is important to look at the conceptual agent list and think about what patterns may be applied to other agents that we didn't include in our agent relation diagrams. By applying the patterns to agents in our conceptual agent list we may discover a better architecture for our system that otherwise we may have been missed. The candidate agent list contains all the conceptual agents in addition to the new agents we discover. It is important to note that not every agent we may discover will become a software agent in our final system. This candidate agent list is developed as a resource to be used when creating the internal use cases and in the creation of the agent interaction diagrams.

3.10 Brief Internal Use Cases

The *brief internal use cases* attempt to decompose a service into one or more *goals*. They are also the first step in preparing to create the internal use cases. In order to create the brief internal use cases we read the external use cases and identify potential *goals* and a simple *plan* to complete each goal. These goals are used to provide the services for the system.

We recommend creating many different brief internal use cases when trying to decompose a service. The brief internal use cases can be quickly created and provide an excellent tool for discovering the architecture for the system. After deciding on the brief internal use cases for the system any extraneous brief internal use cases can be discarded. The main purpose of the brief internal use cases is to define a decomposed service as one or more manageable *goals*, which can then be assigned to the proper agents in our system.

When deciding on the creation of goals we need to keep in mind that agents can only communicate with each other through goals. Thus whenever agents must communicate with each other we must create goals for them to do so. Ideally we would like the goals to be as abstract as possible because fewer goals are usually easier to work with. However, the goals should not provide too much functionality because they should be reusable as well. The goal assignment patterns provide a general description of what is desirable in a properly decomposed goal.

3.11 Detailed Internal Use Cases

The *internal use cases* focus on describing the detailed *plan* for each *goal* that will provide the service we are currently looking at. Each internal use case name represents a conceptual goal. Every goal has a plan that can include other goals, which in turn have their own plans.

The main goal of the internal use case is the detailed description of goals and the plans that will complete each goal. The internal use cases follow a standard use case format with a few modifications. The internal use cases all belong to a particular service that they are providing. In an internal use case the service that each goal belongs to is indicated by the service line, which is in bold so it can be easily identified. In addition to belonging to a service internal use cases can have intentions that are actually goals with their own plans called sub-goals.

Sub-goals are essentially the same as goals in all regards, except a sub-goal helps provide a goal instead of a service like regular goals. The creation of these sub-goals often occurs during the creation of the internal use cases because the internal use case construction provides the developer with a better understanding of the architecture of the system. The increased knowledge of system allows the developer to suggest the creation of sub-goals that provide a better decomposition of the service they provide. The sub-goals are advantages because they describe an intention in more detail, but still keep the goals they extend from becoming over complicated.

We describe a sub-goal as extending from its parent goal. These sub-goals are goals the parent goal can use in order to meet their own goal. A sub-goal is noted in an internal use case by the "extends" identifier. Immediately after the "extends" identifier is the name of the parent goal it belongs to and the intention or intentions it expands upon. It is often difficult to create every internal use case until we begin to create the agent interaction diagrams.

During the creation of the agent interaction diagrams new goals or sub-goals can be discovered. It is especially common to discover creational goals during the creation of agent interaction diagrams. When these new goals are discovered during the creation of the agent interaction diagrams it is useful to go back and create or update the internal use cases to reflect the decisions made when creating the agent interaction diagrams. This is useful because the internal use cases are used to define much of the static structure that is found in the BDI agent cards.

3.12 Agent Belief List

The *agent belief list* provides a list of beliefs that are needed to carry out each goal and sub-goal that is listed in our internal use case scenario or plan. Our BDI agent software development process centers on the idea of goal discovery and the assigning of those goals and their corresponding beliefs and intentions to agents. At this point in the development process we have defined the goals and each goal's intentions. In addition to assigning goals to agents we must discover the *beliefs* that are needed to complete each goal.

Goals require certain knowledge in order to be fulfilled, we call this needed knowledge beliefs. The agent belief list shows what beliefs each goal requires in order to be fulfilled. The agent belief list contains the name of every goal in our system, which is then followed by the beliefs and a reason describing why each belief is necessary. We order each set of beliefs under the bold title of service that each goal provides. The reason we group the beliefs by service is to limit conflicts that may occur from two different services that have similarly named goals.

3.13 Agent Interaction Diagrams

During the creation of the agent *interaction diagrams* we *assign goals* to agents and describe how the agents *communicate* with each other in order to provide a service. The internal use cases give us a rough idea of what agents might work together to provide the services for the system. The agent interaction diagrams are different than the internal use cases in the fact that we are now focused on assigning goals to agents or who will do what, instead of understanding how it will be done, which is the focus of the internal use case. If we make changes in the interaction diagrams that change how the internal use cases work, we must update the internal use cases to reflect these changes. When assigning goals to agents we can use the agent goal assignment patterns to aid us.

Agents are depicted by the words that are at the top of the vertical lines in our agent interaction diagrams. Agent communication is depicted by lines with arrows and labeled with goal that is being invoked. Internal agents are represented with rectangles and external agents are represented with ovals.

3.14 BDI Agent Cards

The *BDI agent cards* can be created in parallel with the Agent Interaction Diagrams. We can use the BDI agent cards as a way to bring all the different parts of an agent together into a single entity. The BDI agent cards are based upon the object-oriented design cards called CRC cards [2]. The BDI agent cards and the agent interaction diagrams represent the architecture of the system.

The BDI agent cards document the *static* architecture of the system and the agent interaction diagrams detail the *dynamic* collaboration between the agents of our system. By creating the BDI agent cards we are able to describe the static structure of the agents in a single artifact. After the creation of the BDI agent cards and the agent interaction diagrams all the *goals* will be assigned to agents.

During this phase in the development process a key action will be making sure both the BDI agent cards and the agent interaction diagrams reflect the same architecture for the system. Often a change to one BDI agent card or agent interaction diagram will cause a change in the other and vice versa. The BDI agent cards and agent interaction diagrams completely define the architecture of our system, which can then be used to create our system in software.

4 Conclusions

This research lays the groundwork for a BDI agent software development process. The BDI agent software development process that we propose is designed to be usable by today's software developer. Our process is not overly complex, but is designed to be a systematic process for developing agent-based systems. In this research we have proposed both our BDI agent software development process and provided a case study to clarify the use of the process for agent software development.

There are several salient points in our BDI agent development process. In our BDI agent development process we describe agents as those entities that we assign BDI too. There are two key activities that place in our BDI agent software development process. These key activities are the *discovery of agents* and the *discovery of the BDI* for each agent. Not only do we use traditional tools, but we also propose new tools to identify the potential agents in our system.

There are many options for future research. This research is just the first iteration in the development of a process for developing agent-based systems. This process can be further refined and additions can be made to improve areas that prove difficult to use when constructing agent-based systems. Agent patterns represent a promising area of research that can aid in building agent-based systems by leveraging solutions to common problems found when constructing agent-based systems. A programming language that is specifically designed to simplify the creation of agent-based systems in software can be created [15]. This BDI agent software development process could also be extended to better describe artificial intelligence elements that may be required for creating intelligent agents. The field of agent-based software engineering is still relatively young, but it holds great promise for the future development of complex systems.

References

1. Beck, K. Extreme Programming Explained-Embrace Change, *Addison-Wesley*, (2000).
2. Bellin, David and Simone, Susan, The CRC Card Book, *Addison-Wesley*, (1997).
3. Booch, G., Object-Oriented Analysis and Design with Applications, *Addison Wesley*, (1994).
4. Booch, G., Rumbaugh, J., Jacobson, I., The Unified Software Development Process, *Addison-Wesley*,(1999).
5. Bratman, M. E., Intention, Plans, and Practical Reason, *Harvard University Press*, (1987).
6. Cockburn, Alistair, Writing Effective Use Cases, *Addison-Wesley*, (2001).
7. Depke, Ralph, Heckel, Reiko, and Kuster, Jochen, Improving the Agent-Oriented Modeling Process by Roles, *AGENTS'01*, (June 2001), 640-647.
8. Fowler, Martin and Scott, Kendall, UML Distilled Second Edition: A Brief Guide to the to the Standard Object Modeling Language, *Addison-Wesley*, (2000).

9. Gamma, E., R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object Oriented Software, *Addison-Wesley*, (1995).
10. Hayden, Sandra, Carrick, Christina, and Yang, Qiang, A Catalog of Agent Coordination Patterns, ACM Press, (1999), 412-413.
11. Iglesias C. A., Garijo M, Gonzalez J. C., and Juan R. Velasco, Analysis and Design of Multiagent Systems using MAS- CommonKADS, *4th Int. Workshop on Agent Theories, Architectures, and Languages (ATAL-97)*, volume 1365 of LNAI, Springer-Verlag, (July 24-26, 1998), 313-328.
12. Jennings, N. R., On agent-based software engineering, *Artificial Intelligence,* v.117, (2000), 277-296.
13. Jennings, N. R., An Agent-Based Approach for Building Complex Software Systems, *CACM*, 44(4), (April 2001), 35-41.
14. Jo, Chang-Hyun, A Seamless Approach to the Agent Development, *ACM SAC 2001*, (2001), 641-647.
15. Jo, Chang-Hyun and Allen J. Arnold, Agent-Based Programming Language (APL), *ACM SAC 2002*, Madrid, Spain, (March, 2002), 27-31.
16. Kinny, D., Georgeff, M., and Rao, A. A Methodology and Modelling Techniques for Systems of BDI agents. Proc. of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Vol. 1038): 56-71, Springer, 1996.
17. Kinny, D. and Georgeff, M., Modelling and Design of Multi-Agent Systems, 3rd International Work shop on Agent Theories, Architectures, and Languages (ATAL-96), Springer, Lect. Notes in AI, 1996.
18. Larman, Craig, Applying UML and Patterns: Second Edition, *Prentice-Hall*, (2002).
19. Odell, J., Parunak, H. V. D., and Bauer, B., Extending UML for Agents, Proc. of the Agent-Oriented Information System Workshop at the National Conf. on AI, (AOIS Workshop at AAAI 2000), (2000).
20. Petrie, Charles, Agent-Based Software Engineering, *Agent-Oriented Software Engineering, Lecture Notes in AI*, *Springer-Verlag,* (2001), 58-76.
21. Rao, Anand S. and Georgeff, Michael P., BDI Agents: From Theory to Practice, *Australian Artificial Intelligence Institute*, (1995).
22. Weiss G., editor, Multi-Agent Systems, *The MIT Press*, (1999).
23. Wooldridge, M. and Jennings, N. R., Intelligent Agents: Theory and Practice, *Knowledge Engineering Review, Cambridge Univ. Press,* 10(2), (1995), 115-152.
24. Wooldridge, M. and Jennings, N. R., Software Engineering With Agents: Pitfalls and Pratfalls, *IEEE Internet Computing*, (May-June 1999), 20-27.
25. Wooldridge, M., Jennings, N. R., and Kinny, D., A Methodology for Agent-Oriented Analysis and Design, *Autonomous Agents 1999,Seattle, WA,* (1999), 69-76.
26. Wooldridge, M., Reasoning about Rational Agents, *The MIT Press: Cambridge, MA*, (2000).

Appendix: A Case Study

The *case study* is to show how to use our process suggested here to develop a BDI agent-based application such as the *Notice Management System* in the *Weather Forecasting System*. Because of the limitation of the paper length, a detailed *case study* is provided at http://www.ecs.fullerton.edu/~jo/research.

Case Study: Initial Problem Statement

A customer would like to receive special notices of certain types of weather events. The business will direct the forecasters that they need to create these new notices. We need to develop a tool that will aid the forecasters in providing notices to districts inside a state. The system should be able to provide notices for a variety of events (frost, severe-weather, freezing rain).

… (*omitted*) …

Case study: Enterprise Software Assessment

We currently have a system that stores all our weather products in a database. The notices could be added to a database as a new weather product. Once a notice is received in the database we could provide another process that will handle the delivery and formatting of a notice.

… (*omitted*) …

Case study: Brief External Use Cases

Name: CreateNotice

Description:

A forecaster identifies the need to submit a notice or notices in a region. The forecaster starts the notice interface. The forecaster selects the state to submit notices in. The forecaster then selects the districts to submit notices to. The forecaster then creates and submits the notice to the system. The system recognizes that a notice needs to be delivered. The system formats the notice properly for delivery and then delivers the notice properly.

… (*omitted*) …

Case study: (Detailed) External Use cases

External use case: CreateNotice

Primary Actors: Forecaster, System, District

Stakeholders:

-Forecaster wants fast and accurate entry of the notices. … (*omitted*) …

Preconditions: Forecaster has identified a need to submit a severe weather warning for an area.

Sucess/Postcondition: A notice is delivered to the district and a copy is saved.

Scenario:

A customer requests, from the company, that they receive notices of certain kinds of weather events.

… (*omitted*) …

Extensions: … (*omitted*) …

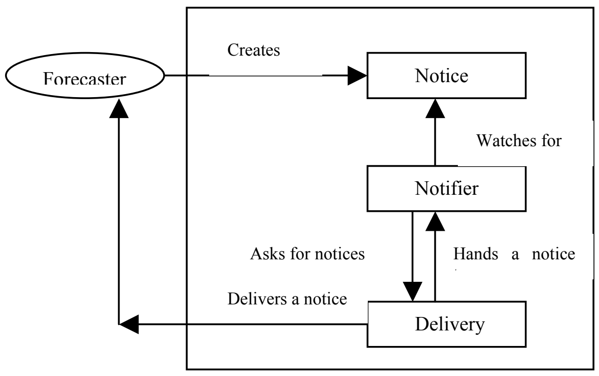
Special Requirements: … (*omitted*) …

Case Study: Conceptual Agent List

Notice	Weather
Notifier	District
System	Web Page
Forecaster	Fax
Customer	Delivery

Case Study: Conceptual Agent Relation Diagram

Service: createNotice



Case Study: Candidate Agent List

Agent	Reason
Notice	Conceptual Agent List
Weather	Conceptual Agent List
… (*omitted*) …	… (*omitted*) …
NoticeManager	By studying the agent relation diagram for the createNotice service we can see that the external Forecaster agent is communicating directly with the notice agent. By applying the manager pattern, we identify the possible need for a NoticeManager.
… (*omitted*) …	… (*omitted*) …

--

Einhorn, Jeffery, M. and Chang-Hyun Jo, *A Use-Case Based BDI Agent Software Development Process*, Proc. of the 2nd International Workshop on Agent-Oriented Methodologies - OOPSLA-2003, Anaheim, CA, USA, 7-20, Oct. 26-30, 2003.

Case Study: Brief Internal Use Cases

Service: CreateNotice

Name: CreateNotice

Description:

The forecaster has identified a need to submit a notice for an region. The forecaster starts the notice interface. The forecaster selects the proper state to issue a notice for. The forecaster enters the notice text. The notice is formatted and submitted and is stored in the database.

… (*omitted*) …

Case Study: Internal Use Cases

Service: CreateNotice

Internal use case: CreateNotice

Actors: NoticeManager, Forecaster, Notice

Stakeholders:

-Forecaster wants fast and accurate creation of the notices.

:

Preconditions: Forecaster has identified a need to submit a notice for an area.

Postcondition: Notice is delivered/saved to the database.

Scenario (intentions):

Forecaster asks the NoticeManager agent to create a Notice.

The NoticeManager agent provides an interface to the forecaster for notice creation.

NoticeManager agent properly formats and submits the notice to the database.

… (*omitted*) …

Case Study: Agent Belief List

Service: CreateNotice

Goal: CreateNotice

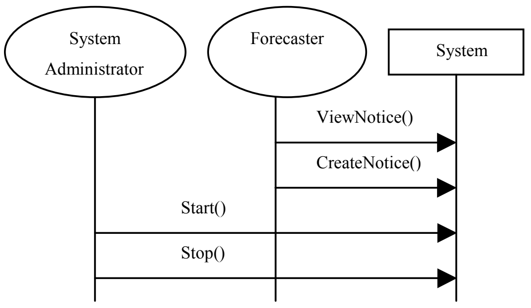
Belief: NoticeDB

Reason: We need to know the database to submit notices to.

… (*omitted*) …

Case Study: Agent Interaction Diagrams

System Services Interaction Diagram



Case Study: BDI Agent Cards

Agent: NoticeManager

BDI list:

Desire: CreateNotice

Pre-condition: Forecaster decides to create a Notice.

Belief: NoticeDB

Post-condition: Notice is saved in the database.

Collaborators: Forecaster (external), Notice

Intentions:

Forecaster asks the NoticeManager agent to create a Notice.

The NoticeManager agent provides an interface to the forecaster for notice creation.

NoticeManager agent properly formats and submits the notice to the database.

… (*omitted*) …

--

Einhorn, Jeffery, M. and Chang-Hyun Jo, *A Use-Case Based BDI Agent Software Development Process*, Proc. of the 2nd International Workshop on Agent-Oriented Methodologies - OOPSLA-2003, Anaheim, CA, USA, 7-20, Oct. 26-30, 2003.