5-2002

A BDI Agent Software Development Process

Jeffrey M. Einhorn

A BDI AGENT SOFTWARE DEVELOPMENT PROCESS

by

Jeffrey M. Einhorn
Bachelor of Arts, University of St. Thomas, 1999

A Thesis

Submitted to the Graduate Faculty

of the

University of North Dakota

in partial fulfillment of the requirements

for the degree of

Master of Science

Grand Forks, North Dakota
May
2002

This thesis, submitted by Jeffrey M. Einhorn is partial fulfillment of the requirements for the degree of Master of Science from the University of North Dakota, has been read by the Faculty Advisory Committee under whom the work has been done and is hereby approved.

[illegible] (Chairperson)

[illegible]

[illegible]

This thesis meets the standards for appearance, conforms to the style and format requirements of the Graduate School of the University of North Dakota, and is hereby approved.

[illegible]
Dean of the Graduate School

April 24, 2002
Date

PERMISSION

Title	An Agent BDI Software Development Process
Department	Computer Science
Degree	Master of Science

In presenting this thesis in partial fulfillment of the requirements for a graduate degree from the University of North Dakota, I agree that the library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by the professor who supervised my thesis work or, in his absence, by the chairperson of the department or the dean of the Graduate School. It is understood that any copying or publication or other use of this thesis or part thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of North Dakota in any scholarly use, which may be made of any material in my thesis.

Signature [illegible]

Date 4/23/2002

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

ACKNOWLEDGEMENTS

There are many people I owe thanks for aiding me in my research. First and foremost I would like to thank my advisor, Dr. Jo, for sharing his valuable time and experience in this research. Without Dr. Jo's continued patience, logical thinking and constant advice my thesis would have never reached completion. Special thanks are due to Dr. Thomas Wiggen and Dr. Thomas O'Neil for being my research committee members and for their comments and suggestions regarding this thesis.

I would like to thank the rest of the Computer Science faculty for making the Computer Science Department an excellent place for higher learning and intellectual growth. Special thanks go to Julie Kosmatka and Linda Kilichowski for helping with all my special requests that I needed in order to complete my graduate degree.

In addition I would like to thank Douglas Rand and Steve Etzell for providing their valuable time in previewing preliminary drafts of my thesis. I would like to thank the all people at Meridian for creating a company that values higher education.

Finally I would like to express my sincere gratitude to my wife Kristi for her love and understanding during my research. I would like to thank my parents for raising me in an environment that encourage learning and for their continued support of my goals.

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 thesis, 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 thesis 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. These are just some of the tools that provide valuable insight into the development of our BDI agent software development process.

In addition to describing our software development process, we will also provide a case study to clarify the description of our BDI agent software development process.

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.

CHAPTER I
INTRODUCTION

In this chapter we will describe why there is a need for our research and what we hope to accomplish with our research.

Reason for Research

The field of Software Engineering has continued to evolve ever since the first programming language was developed. The first software systems that were developed were relatively simple and could only accomplish simple tasks. A large reason for the simplicity of the software was because of the limitations of the hardware. The software developer had to be very creative to get the most out of the limited hardware resources. There were very few techniques to aid a software developer in creating software systems. Most programming was done in assembly language or other low level languages and there was very little code reuse. However, as each generation of computer hardware has become more powerful, more has been expected of the software. New software engineering techniques were developed, as a response to the increased demands for more complex software systems. These new tools were better able to leverage the increased capacities of the new computers. In addition these new tools helped manage the complexity of developing larger and increasing complex systems by allowing the software developer to work at a higher level of abstraction. Higher-level languages like FORTRAN were developed, which allowed scientists to focus more on the problem at hand and less on the low level details that were essential with previous languages. The development of programming languages, such as C, PL/1, COBOL, allowed programmers to leverage the advantages of reusable libraries that increase programmer productivity. However, users continued to demand more complicated software, so new software engineering practices such as structured programming were developed to help manage these complex systems. Structured programming tried to divide large systems into smaller blocks based on the systems functions. Still software became too complex to manage easily and thus, object-oriented software engineering practices were developed, so that software could be encapsulated into reusable pieces that communicated with each other by passing messages. Even object technologies could not keep the ever-growing software complexity under control. So new techniques such as Use Cases [Cockburn 2001], CRC Cards [Bellin and Simone 1997], Patterns [Gamma et al. 1995], UML [Fowler 2000] and Unified Development Process [Booch et al. 1999] were developed to try and keep the complexity in check. It is clear from looking at the past trends that the software development tools will continue to evolve to leverage the advances provided by new generations of hardware.

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. 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. Any agent-based software engineering process must increase programmer productivity if it is ever to have any success. When we say it should increase programmer productivity this includes providing a process that allows software developers to more readily construct complex systems. The research in the following chapters will describe our new agent-based software development process. Our agent-based software development process does not seek to reinvent the wheel, but rather to build upon and expand proven existing tools and methodologies for use in our process.

Research Goal

The goal of this thesis 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.

Systems built from agents provide a natural way to describe and build complex systems [Jennings 2000]. It has been noted in a paper [Wooldridge and Jennings 1999] that agents are mostly based in computer science and only have a slight AI element. We share this idea and thus will build upon existing successful software tools for constructing our agent-based software development process. This approach will allow us to both leverage existing research and create a development process that is familiar to practicing software engineers.

Regardless of what processes are used for developing software, two basics steps usually take place. The first step in software development is analysis. In analysis we describe the problem in order to get a clear understanding of what must be done. The second step in software development is design. During design we develop a solution to the problem we described during analysis. These general concepts can be found in many of the successful software development processes that have been created. Agent-based software development processes will also need to provide tools to aid analysis and design if it is to be successful. The agent-based software development process that we describe will provide techniques and methods to aid in both analysis and design of complex systems.

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.

CHAPTER II
BACKGROUND

The key objective of this research is to facilitate the development of an agent-based software development process. In order to comprehend this research we must first decide on what exactly defines an agent. This chapter will describe what defines an agent. We will also describe important tools like use cases in this chapter. We will also describe the BDI (Belief-Desire-Intention) model briefly. In understanding the tools and methods that we propose will allow us lay groundwork for our agent-based development process. Finally, we will describe and compare other research that we found useful to our own research.

Definition of an Agent

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 [Wooldridge 2000]. Wooldridge and Jennings describe agents as having autonomy, proactiveness, reactivity and social ability [Wooldridge and Jennings 1995]. In particular autonomy requires that agents have their own thread of control. Agents need to be able to function independently of any other agents in the system. Thus any agent system will be fundamentally multithreaded, which means agents will have their own thread of control. Agents are proactive in the sense that they have a goal and can modify their behavior in order to achieve that goal. Agents must be able to react and respond to changes in the environment. Agents do not just pursue their goals in a bubble, but must work in a constantly changing environment. It is clear that agent-based systems will have many agents. In order for agent-based systems to work, agents must be able to communicate and work with other agents.

This research will focus on providing software solutions by building software systems made up of agents. We will provide a simple definition of an agent for our research. Entities become agents when we can assign beliefs, desires and intentions to them. From this point on in this research we will refer to beliefs, desires and intentions as BDI [Bratman 87].

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.

For the purpose of our research it is necessary to provide a definition of an agent-based system. In our research an agent-based system is a system that is made up of agents [Jo 2001]. Each agent is defined by a set of BDI. In addition agents have control over both their state and behavior. At this time it is important to make the distinction that we do not require all the agents that make up our system to be intelligent 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 prove inadequate for agent-based development. It is useful to compare the differences between objects and agents, before we propose our agent-based development process.

Objects are described as having state and behavior. Objects have control over their state in the sense that it can change, but their behavior remains constant. Agents can also be described as having state and behavior. However, agents have control over both their state and their behavior [Weiss 1999]. Thus, Agents can change both their state and behavior at any time. We will use the belief-desire-intention (BDI) model [Rao 1995] to help define agents in our system.

We take a goal-oriented approach in our agent development process because it is a natural extension of object-oriented software development. We can readily describe the functions or services that our system should provide. In our agent-based development process we describe a goal or set of goals that will provide the service for the system.

The Unified Development Process [Booch et al. 1999] is a tested tool [Larman 2002] for building robust object-oriented systems and we studied many of the successful strategies like use cases and UML that are described in the Unified Development Process [Booch et al. 1999]. Due to the differences between object-oriented development and agent-based development the artifacts found in Unified Development Process often require modification for use in agent-based development. We use the term artifact to describe the items that are created during the different steps of a development process. When object-oriented tools prove inadequate we create new agent-based tools that better describe the software system.

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 [Cockburn 2001]. Use cases are an essential part of successful object-oriented software development processes like the Unified Development Process [Booch et al. 1999]. Use cases provide a functional approach to gathering requirements. Object-oriented software systems can be described by using a functional approach. This has been proven by Craig Larman's software development process [Larman 2002]. Jennings also supports functional analysis by describing it as more natural than data or object type analysis [Jennings 2001]. 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.

Use cases are not object-oriented or even agent-oriented in nature, but they provide valuable insight into the requirements of a problem. We will provide some slight modifications to the use cases to increase their usefulness to our agent-based software development process. We will use the modified use cases to discover the requirements of our system. Once we understand the systems requirements we can use them to design an agent-based system.

It is important to assign the proper responsibilities to objects when developing an object-oriented system. Describing the objects and their interactions is a fundamental need in OOA/D. 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. In the following chapters we will describe our agent-based software development process and give a case study demonstrating its use.

Belief Desire Intention (BDI)

We will use BDI [Bratman 87] [Rao et al. 95] as a tool for defining agents in our development process. The discovery of the BDI that will characterize each agent provides us with several advantages. First of all the BDI provides a natural way to describe an agent. Secondly there are existing systems that successfully use BDI [Wooldridge 2000]. There is a large amount of BDI research that can be leveraged in the creation of agent-based systems [Bratman 87] [Rao et al. 95] [Wooldridge 2000].

The BDI model provides a set of guidelines for describing an agent. We will describe our interpretation of the BDI guidelines for the purpose of this research. 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. Finally, 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.

Other Research

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 [Bellin and Simone 1997], UML diagrams [Fowler and Scott 2000], use cases [Cockburn 2001] and software patterns [Gamma et al. 1995] [Larman 2002] have been invaluable for use in constructing object-oriented systems [Booch 1994]. 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 [Wooldridge and Jennings 1995, Iglesias et al. 1998, Wooldridge and Jennings 1999, Wooldridge et al. 1999, Wooldridge 2000, Depke 2001, Jennings 2001, Jo 2001, Petrie 2001], provide research into the theory of agent software development. Both of these types of research proved useful in developing our agent-based software development process.

Grady Booch's book "Object-oriented Analysis and Design with Applications" provides a detailed description of object-oriented analysis and design for software development [1994]. While object-oriented software development provides many new tools to abstract problems into individual objects, large systems are still difficult to manage. In our research we looked to understand the advantages of object-oriented software development first in order to suggest improvements and changes for use in the agent-based software development process.

Kent Beck [Beck 2000] provides a new software development process that centers on building software from the programmer's prospective. Many programmers don't feel they have the time to learn complex development processes. Kent Beck addresses this issue by presenting a type of development process based on the activity of writing code, which is fundamental to the creation of software systems. A software process that provides techniques that aid in creation of code appeals to software developers who spend most of their time creating code. Kent Beck's extreme programming process combines several different practices in order to provide a way to build software successfully [Beck 2000]. Some of the key features found in the extreme programming process are programming in pairs, writing test cases first and implementing only a few features at a time [Beck 2000]. We found Kent Beck's research interesting because he takes a fresh approach to the development process. Kent tries to identify some of the key hurdles to software development and then explains how his process deals with these obstacles. In our research we tried to capture the spirit of Kent Beck's software development process by creating a process that also kept the needs of the software developer in mind.

David Bellin and Susan Simone describe the use of CRC cards to assist in the discovery of classes in an object-oriented system [Bellin and Simone 1997]. Their research provides an extremely detailed discussion of the use of CRC cards as aids, which allow teams to use CRC cards for class discovery. We feel that the CRC cards lack enough structure to be used as a reliable tool for the discovery of software objects or agents. CRC cards are useful for describing the static structure of an agent or object. In our process we have developed a modified CRC type artifact called a BDI agent card, for use in our agent-based software development process. The BDI agent card can clearly represent the static structure of an agent in our process.

Alistair Cockburn's book "Writing Effective Use Cases" explains the use of use cases for software development [2001] in great detail. Use cases are useful tools that allow software developers to naturally identify the functions of a system. Use cases are neither object-oriented nor agent-based, but are functional in nature. We have created modified versions of use cases for use in our agent-oriented development process. These modified agent use cases will help discover the possible goals and services that we will need in our system.

"Applying UML and Patterns", by Craig Larman, describes a development process for constructing object-oriented systems. His research provides a process that can be followed by developers to construct object-oriented systems. Larman's development process leverages many tools to aid in software development such as use cases, UML and patterns. Our development process attempts to provide the same type of practical development process for building agent-oriented systems.

The Gang of Four research on design patterns describes solutions to common problems encountered in developing software [Gamma et al 1995]. They provide three general categories of design patterns based upon their purposes. The three types of designed patterns presented by the GOF are Creational, Structural and Behavioral. Creational patterns provide solutions for object creation. Structural patterns provide guidelines for laying out the proper structure of objects and behavioral patterns can be used to help design object interactions properly. These design patterns provide a way for software developers to leverage the previous experience of other software developers.

We suggest some basic patterns for agent-based software development in this research. Patterns are particularly useful when solving complex problems and they are useful for managing complexity in our agent-oriented development process. Whenever the same problem is solved over and over again we will try and develop a pattern that can be used to solve that problem.

The research paper "A Catalog of Agent Coordination" proposes the use of patterns in agent-oriented development [Hayden et al. 1999]. They provide a brief description of a broker agent pattern. We feel that a pattern similar to a broker agent may be useful in agent-oriented development. In our research we try to discover other several common patterns that we could apply to our case study.

The paper "Improving the Agent-Oriented Modeling Process by Roles" takes the approach of describing a system using roles [Depke et al. 2001]. 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. We find their role-based process is difficult too follow. Wooldridge, Jennings and Kinny [1999] 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. Currently it seems unlikely that a system based on constantly changing roles can be easily described or implemented. In our process we have tried to provide a natural way of decomposing a software system into agents. Instead of focusing on roles we focus on describing the beliefs, desires and intentions for each agent.

"Analysis and Design of Multiagent Systems using MAS-CommonKADS" [Iglesias et al. 1998] is a paper that discusses software development process based upon knowledge engineering. The knowledge-base approach they present lacks the rigid approach that we feel a software developer needs in order for a process to be valuable to them. In our research we describe the problem using agents that have beliefs, desires and intentions. We feel that the knowledge engineering research could be leveraged in describing the beliefs of an agent. Further more, we seek to present a process such that each successive artifact builds upon the previous artifacts.

Michael Wooldridge has published several research documents on agent software development techniques [Wooldridge and Jennings 1995, Wooldridge and Jennings 199, Wooldridge 2000, Wooldridge et al 1999]. His approach is based in agent theory. In his book "Reasoning about Rational Agents" Wooldridge presents a detailed BDI architecture, which is designed for building BDI agent systems [Wooldridge 2000]. Wooldridge's research describes agent theory in great detail. In our research we try to apply modern software engineering principles to the agent theory presented by Wooldridge, Jennings and others. In integrating the theory described by Wooldridge and modern software engineering principles we describe a practical development tool that software developers can use for the creation of agent-based systems.

Rao and Georgeff [Rao and Georgeff 1995] 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.

Nicholas Jennings has published several research documents that support the advantages of agent-based software engineering [Jennings 2000, Jennings 2001]. This research lays the foundation for building agent-based systems. Jennings argues that agent-based systems provide a natural and useful way to build complex systems. In our research we provide the beginnings of a process that can be used to develop agent-oriented systems.

CHAPTER III
BDI AGENT DEVELOPMENT PROCESS

This chapter describes our BDI agent development process in detail. A sample case study is included, which will clarify the explanation for each artifact.

Brief Process Description

A salient point in our research is the use of BDI [Rao et al. 1995] 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 agents 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.

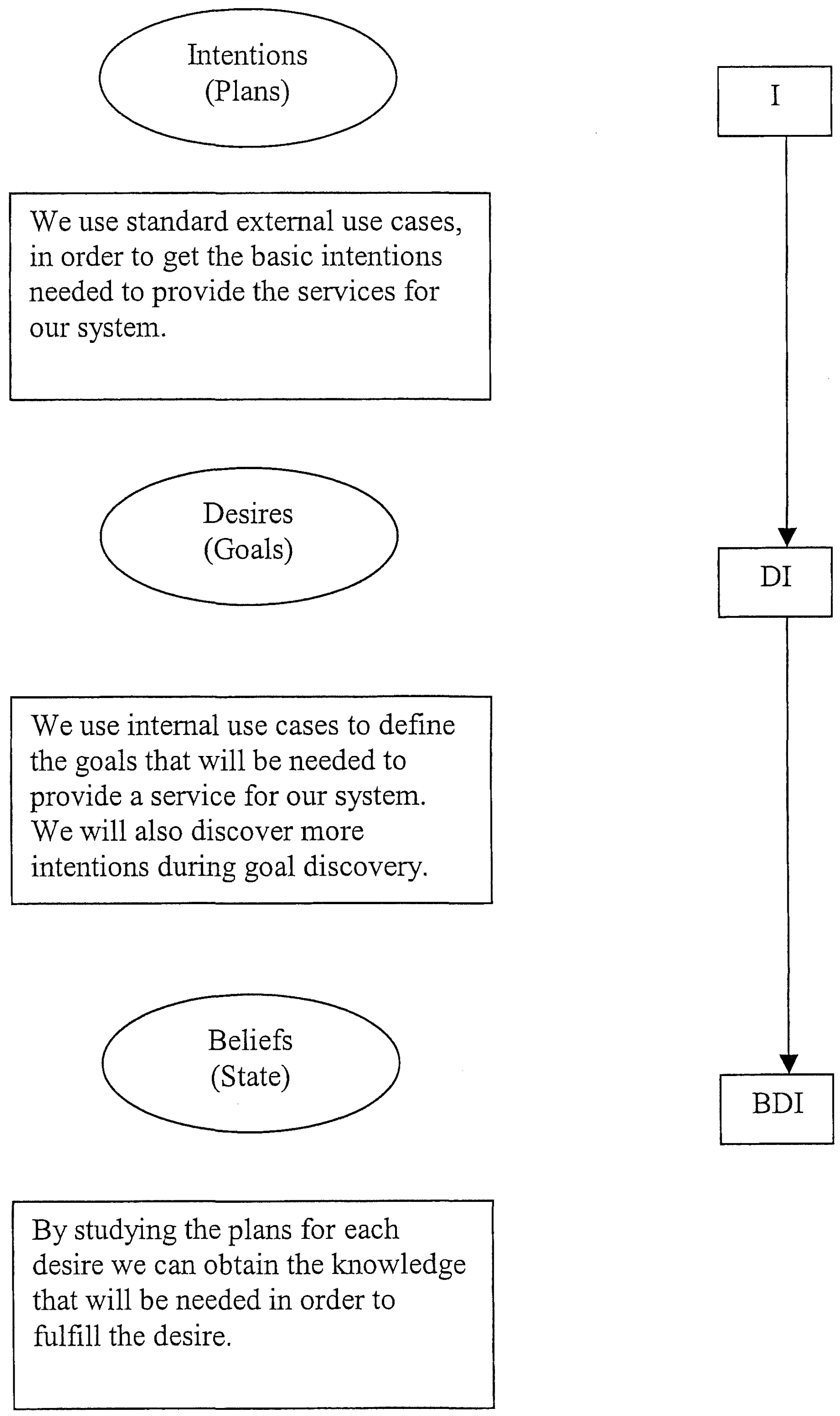
Intentions
(Plans)
I
We use standard external use cases, in order to get the basic intentions needed to provide the services for our system.
Desires
(Goals)
DI
We use internal use cases to define the goals that will be needed to provide a service for our system. We will also discover more intentions 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 Agent Model.

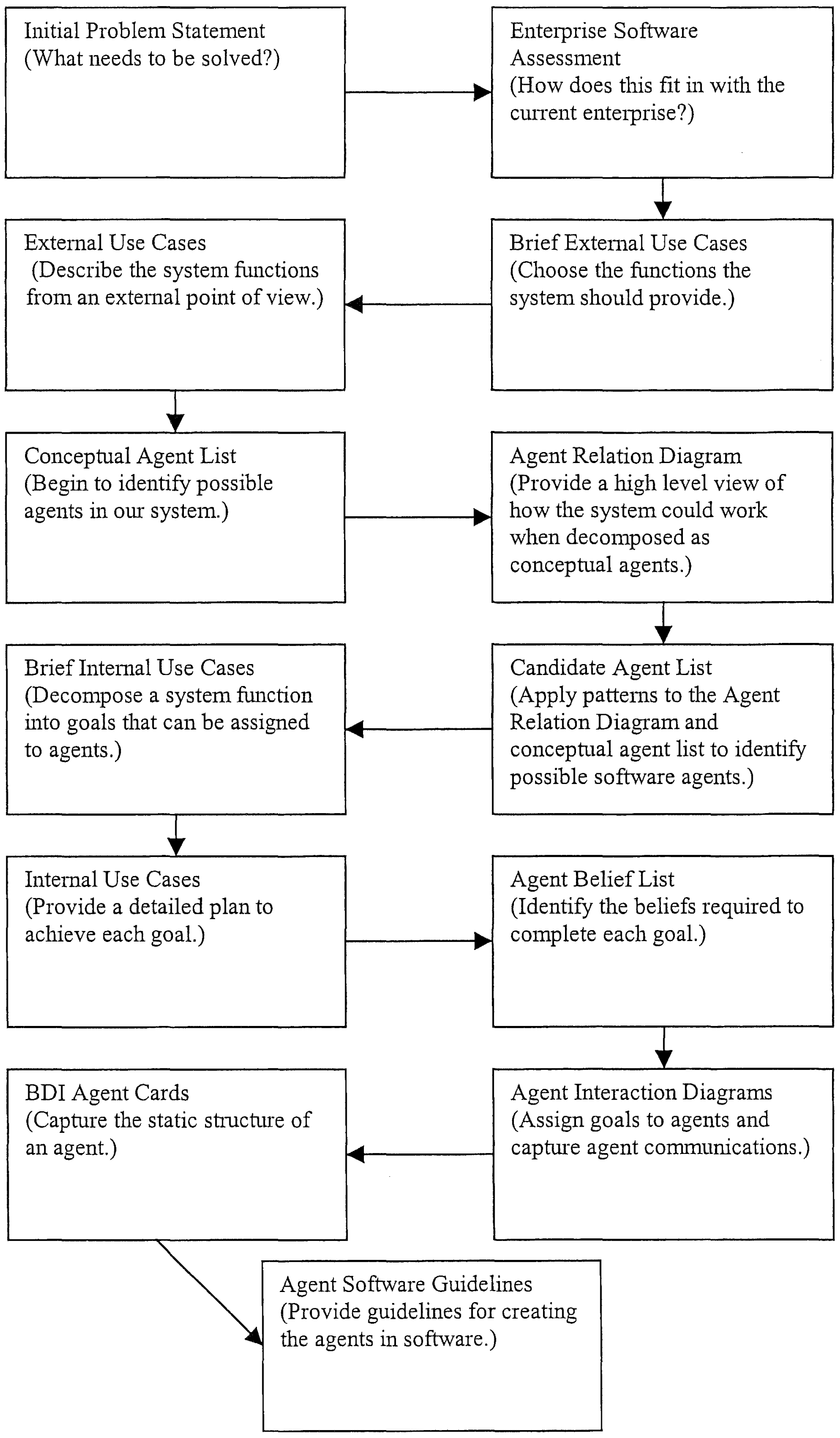
Initial Problem Statement
(What needs to be solved?)
Enterprise Software Assessment
(How does this fit in with the current enterprise?)
Brief External Use Cases
(Choose the functions the system should provide.)
External Use Cases
(Describe the system functions from an external point of view.)
Conceptual 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 Software Development Process.

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 create 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.

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.

We have taken a brief tour of the complete BDI agent software development process. We will now systematically describe our process in much greater detail. The following section will analyze each artifact in the order that is presented in Figure 2. We will present a detailed discussion of each artifact that can be created in our process and provide a case study providing a practical view of an actual artifact once it is created.

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. The following case study is an example of an initial problem statement for a Notice Management System.

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). The customer wishes to use these notices as a warning that they may need to take action in order to respond to an event. Our business would like the interface to be fast and easy to use in order to minimize both the time and cost to the forecaster in creating the notices. We do not want the forecaster to have to worry about the delivery of the notices. Instead we would like to develop a system that will automatically deliver the notices to the districts once they are created. The forecasters job is identifying when to create a notice. The forecaster will use an interface to create the notices. The system should be able to format and deliver the notices, created by the forecasters, as needed. The customer often wants the notices delivered in a variety of formats (web pages, faxes or both). The customers usually want the notices delivered to each district where the notice is valid.

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. When looking at the software that exists we should first consider the software that is already readily available to customer. Just one of the many advantages of using software that already exists is the existing software most likely already works properly in the enterprise. Another important advantage for using existing software is that any software we will create will probably require maintance. Once we have identified the existing software we can propose how our system will fit in with the existing software. It is cost prohibitive to the customer to have software developed from scratch. As a result software developers should strive to maximize any software that has already been created. If software needs to be created from scratch it is important to develop software that will work with the other software already available in the enterprise. If people must use the software it should behave similarly to current software to reduce the amount of training necessary for the new software.

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. The following case study is an example of a current business outlook for our Notice Management System.

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. We currently have an internal system set up called the notifier that can watch the database for different kinds of weather products to be inserted. We can use the notifier to signal the system when new notices are created.

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 actors, interaction with this encapsulated system. The format for a brief external use case is fairly simple. Each brief external use case is identified by a "name" and is followed by a "description" that describes a scenario for this use case. The following case study lists the brief external use cases that where created for the Notice Management System.

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.

Name: ViewNotice
Description:
A forecaster wishes to view the notices that are currently valid. The forecaster starts the interface and selects the region to view notices in. The forecaster is able to easily see where valid notices are and can bring up the details of a notice as desired.

Name: Start
Description:
The system manager needs to start the system.

Name: Stop
Description:
The system manager needs to be able to stop the system.

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.

The external use cases are based on the fully dressed use cases described by Larman [2002]. We use the distinction "external" because the external use cases are oriented around the external actors and their interaction with the system. By developing an external use case we gain a better understanding of how the each service could be provided. The external use cases provide a description of how a service will be provided from an external actor's viewpoint. We will create an external use case for each service that we feel the need to understand better.

In the following case study we have created external use cases for the services createNotice, viewNotice and start. At this time there was no need to create an external use case for stop because we did not feel that the external use case for stop would provide any new insight into the operation of the system.

Case study: External Use cases

External use case: CreateNotice
Primary Actors: Forecaster, System, District
Stakeholders:
-Forecaster wants fast and accurate entry of the notices.
-Customer wants accurate and timely delivery of the warnings to districts
-Districts are interested in taking appropriate action for each warning.
-Company wants to satisfy customer interests in a cost effective manner.
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:
1) A customer requests, from the company, that they receive notices of certain kinds of weather events.
2) The company directs the forecaster to create the notices for the customers.
3) A Forecaster recognizes the need to create a notice.
4) Forecaster starts the notice creation interface.
5) Forecaster selects the proper customer to issue a notice for.
6) Forecaster creates the text of the notice.
7) Forecaster submits the finished notice to the system.
8) The system recognizes that a notice needs to be delivered.
9) The system formats the product for delivery.
10) The notice is delivered in the proper format to each district.
11) Forecaster repeats steps 5-6 as needed.

Extensions:
a) System fails:
-any work that hasn't been submited by the forecaster should be lost.
-restart the interface and recreate the notice.

Special Requirements:
-Once the system is loaded it must have a very quick response time (less than a sec or two from the forecasters perspective)

External use case: ViewNotice
Primary Actors: Forecaster, System
Stakeholders:

-Forecaster wants view current valid notices.
-Company wants forecaster to be able to easily view the valid notices.
Preconditions: Forecaster decides to view a notice.
Sucess/Postcondition: Forecaster is able to view the contents of a notice.
Scenario:
1) The forecaster wants to view current valid notices.
2) The forecaster starts the view notice interface.
3) The forecaster selects the customer to view notices for.
4) The interface indicates to the forecaster what districts have valid notices.
5) The forecaster can choose a district and view the valid notice for that district.

Extensions:
5a)
-If if we try and view a notice in a district that currently doesn't have any valid notices, then we should get a message, which says "No valid notice available".

External use case: Start
Primary Actor: System Administrator, System
Stakeholders:
-The System Administrator wants to be able to start and stop the notice delivery system as desired.
-Company wants System Administrator to be able to easily manage the system.
Preconditions: System Administrator decides to start the system.
Sucess/Postcondition: The system is started.
Scenario:
1) The System Administrator wishes to start the notice management and delivery system.
2) The System Administrator starts the notice management and delivery system.
3) The System starts the proper components to enable management and notice delivery.

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 [Abbot 83]. 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. The following case study lists the conceptual agents of the Notice Management System.

Case Study: Conceptual Agent List

Notice	Weather
Notifier	District
System	Web Page
Forecaster	Fax
Customer	Delivery

Conceptual Agent Relation Diagram

The conceptual agent relation diagram (CARD) 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.

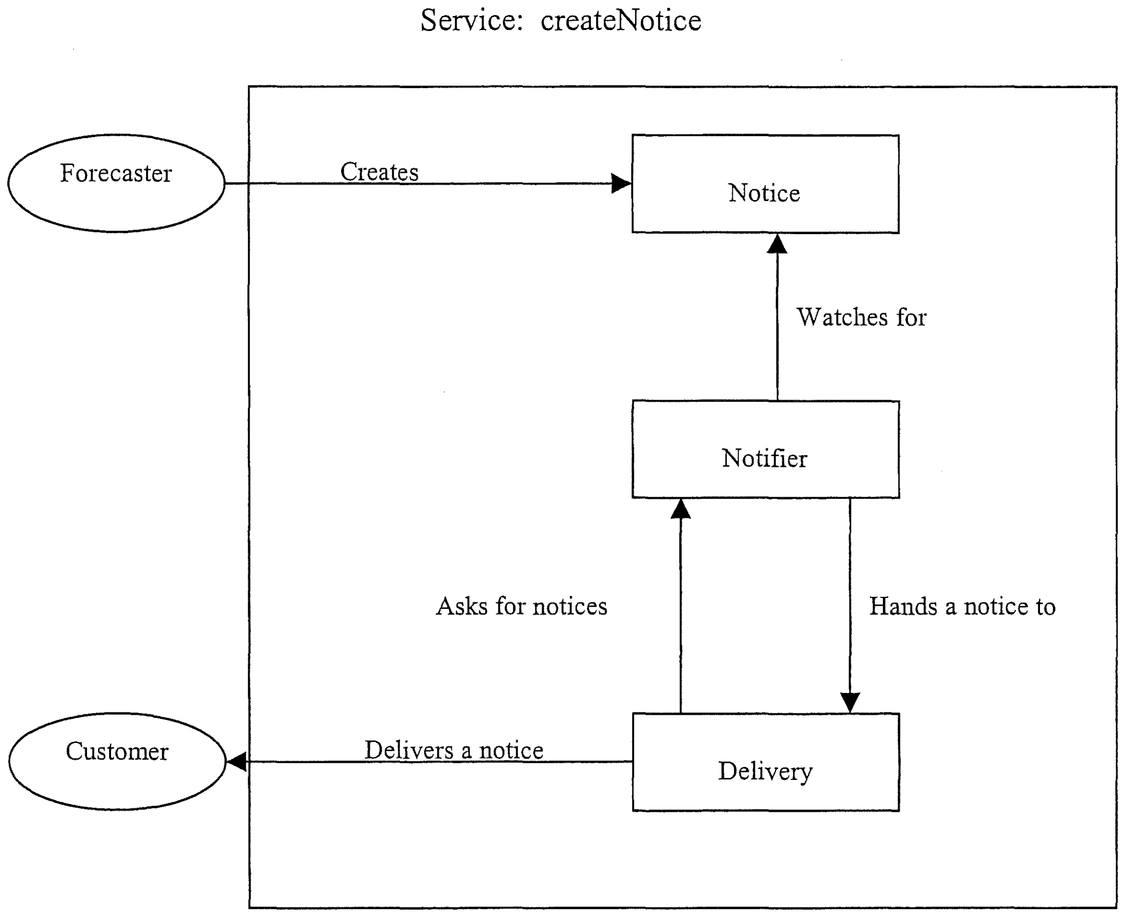

Figure 3. Conceptual Agent Relation Diagram. The conceptual agent relation diagram provides a conceptual view of a possible system. The oval circles represent external agents and the boxes represent internal agents. There is a large rectangle that surrounds all thee internal agents, which represents the system boundary.

We only create conceptual agent relation diagrams for services that we feel will provide insight into the possible software agents that may exist in our system. In the following case study we create an agent relation diagram for the service createNotice. Figure 3 is the agent relation diagram for the createNotice service. The diagram of the createNotice service provides a general view of how a conceptual system might provide the createNotice service.

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 [Gamma et al. 1995] and Larman [2002]. We have also looked at agent design patterns by Hayden, Carrick and Yang [1999] 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 [Gamma et al. 1995]. 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. The manager agent can handle any communication between the internal and external agents. The manager agent can locate the proper internal agent based on its interaction with the external agents. The manager agent also removes the need for the external agents to know detailed information about internal agents. 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 [Hayden et al.]. 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 [2002]. 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 [2002]. 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 is based upon work done by Aridor and Lange [1998]. The notifier pattern 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.

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.

After looking at Figure 3, the agent relation diagram for createNotice, and applying our agent identification patterns we discover a number of potential agents. We see that we have two external actors that interact with our internal agents and we suggest manager agents that handle the interaction between the internal and external agents. We also discover that the delivery agent wants to be notified of any new notices and we would create a notifier agent, but we have already discovered that agent. We notice that several of our agents will need to interact with the database and add the database agent to our list. Upon further review the Delivery agent may need help in delivering notices, so we suggest a deliveryservice agent. We think that it may be possible that we want several different agents to register their delivery services with a broker agent. For example the Web Page and Fax agents may want to provide their services through a DeliveryBroker agent. Table 1 describes the candidate agent list for our Notice Management System.

Table 1. Candidate Agent List.

Agent	Reason
Notice	Conceptual Agent List
Weather	Conceptual Agent List
Notifier	Conceptual Agent List
District	Conceptual Agent List
System	Conceptual Agent List
Web Page	Conceptual Agent List
Forecaster	Conceptual Agent List
Fax	Conceptual Agent List
Customer	Conceptual Agent List
Delivery	Conceptual Agent List
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.
DeliveryManager	We recommend this agent based on applying the manager pattern.
Database	We discover this agent by applying the service pattern. The service pattern provides a single agent that is available to all the internal agents in our system. Many agents will need to get and store information in the database and we may provide a single database agent to handle this.
DeliveryService	We recommend this agent based on applying the service pattern.
DeliveryBroker	We identify this possible software agent based on the broker pattern. We have several possible agents such as Fax and Web page, which provide the function of delivering notices to the districts. The broker could provide a single agent that decides which agent to use for notice delivery.
SystemManager	This agent is recommended from looking at the start brief internal use case. The system manager will ensure the proper agents are created at the systems initialization.

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.

The following case study lists the brief internal uses cases for our Notice Management System. The brief internal use cases are grouped under the service that they should provide. The service name is in bold in order to easily distinguish which brief internal use cases belong to which service.

Case Study: Brief Internal Uses 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.

Name: WatchForNoticesToDeliver
Description:
The notifier is started and asked to watch the database for new notices. When a new notice arrives it is handed to the interested party.

Name: DeliverNoticesToDistricts
Description:
A notice arrives for delivery. The notice contains the information about whom it should be delivered to. The database is checked on how to properly format the notice for delivery. The database is checked on how to format the notice properly. The notice could be delivered as a fax, web page or both. We also want to add the ability to deliver the notice in new formats.

Service: ViewNotice
Name: ViewNotice
Description:
The forecaster decides to view notices and starts the view notice interface to do so. The NoticeManager provides a view notice interface to the forecaster. The forecaster indicates the state it wishes to view valid notices in. The interface indicates which districts have valid notices to the forecaster. The forecaster selects a district to view a valid notice for. The valid notice is retrieved from the database and displayed to the forecaster.

Service: Start
Name: Start
Description:
The System Administrator decides to start the notice delivery system. The SystemManager creates the proper agents that will be needed in order for the system to allow management and delivery of notices.

Service: Stop
Name: Stop

Description:
The System Administrator decides to shutdown the system.

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. We can create sub-goals whenever we feel it is advantageous to do so. When creating a sub-goal we must balance encapsulating functionality in its own sub-goal verses providing so many sub-goals that they are cumbersome to work with. 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.

The following case study lists the internal use cases for our Notice Management System. The internal use cases are structured similarly to the brief internal use cases. Some of these internal use cases where not created until after the creation of some of the agent interaction diagrams. The Create goal that can be found under the start service in our internal use cases is an example of new use case that was discovered during the creation of the agent interaction diagrams.

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.
-NoticeManager handles the interaction with the forecaster and desires proper creation and maintenance of notices.
-Notice contains all the information about a notice.
Preconditions: Forecaster has identified a need to submit a notice for an area.
Postcondition: Notice is delivered/saved to the database.
Scenario (intentions):
1) Forecaster asks the NoticeManager agent to create a Notice.
2) The NoticeManager agent provides an interface to the forecaster for notice creation.
3) NoticeManager agent properly formats and submits the notice to the database.

Internal use case: Create
Extends: CreateNotice, intention 2
Actors: NoticeManager, Notice
Preconditions: We need to create a notice.
Success/Postconditions: A notice is created.
Scenario (intentions):
1) The notice interface is started for notice creation.
2) The NoticeManager gets the State from the user.
3) The NoticeManager gets the district from the user.
4) The NoticeManager gets the notice text from the user.
5) The NoticeManager passes the DistrictIds and the notice text to the Notice.
6) A new notice is created.
7) The NoticeManager now has a notice.

Internal use case: Submit
Extends: CreateNotice, intention 3
Actors: NoticeManager, Database
Preconditions: The database receives a notice to save.
Success/Postconditions: The notice is saved.
1) The NoticeManager sends a notice to the database to be saved.

Internal use case: WatchForNoticesToDeliver
Actors: Notifier, Delivery, DeliveryService
Stakeholders:
-The notifier watches the information that is inserted into the database.
-The delivery agent wants to know when/what/who it should deliver.
-The deliveryservice agent will format the notice as specified and deliver it.
Preconditions: The system needs to recognize when notices should be delivered.
Sucess/Postcondition: The system recognizes that a notice should be delivered.

Scenario (intentions):
1) The delivery agent creates the notifier.

2) The delivery agent asks the notifier to watch the database for notices, which are a type of weather product.
3) The delivery agent listens to notifier for notices.
4) The notifier watches the database for notices to be entered.
5) The notifier gives the delivery agent a notice.
6) The delivery agent sends a request to the delivery service agent to deliver the notice.

Internal use case: DeliverNoticesToDistricts
Actors: DeliveryService, Notice
Preconditions: DeliveryService agent has received a notice to be delivered and how it should be delivered.
Sucess/Postcondition: The notice is delivered in the proper format to the proper districts.
Scenario (intentions):
1) The deliveryservice agent checks the notice for who it should be delivered too.
2) The deliveryservice agent then checks the database on how to properly format the notice for delivery.
3) The deliveryservice agent formats the notice for delivery.
4) The deliveryservice agent delivers the notice as a fax, web page or both.

Service: ViewNotice
Internal use case: ViewNotice
Service: ViewNotice
Actors: NoticeManager, Forecaster, Notice
Stakeholders:
-Forecaster wants fast viewing of valid notices.
-NoticeManager handles the interaction with the forecaster and desires proper valid notice viewing.
-Notice contains all the information about a notice.
Preconditions: Forecaster desires to view a notice area.
Postcondition: Forecaster is able to view the contents of a valid notice.
Scenario (intentions):
1) Forecaster asks the NoticeManager agent to view a Notice.
2) The NoticeManager agent provides an interface to the forecaster for notice viewing.

Internal use case: View
Extends: ViewNotice, intention 2
Actors: NoticeManager, Notice, Forecaster, Database
Preconditions: We need to view a notice.
Success/Postconditions: A notice is viewed.
Scenario (intentions):
1) The notice interface is started for notice viewing.
2) The NoticeManager gets the State from the user.
3) The NoticeManager gets the valid notices from the database.

4) The NoticeManager provides an interface with the districts that have valid notices.
5) The Forecaster selects a district to view a notice for.
6) The NoticeManager displays the notice contents to the Forecaster.

Internal use case: GetValidNotices
Extends: viewNotice, intention 3
Actors: NoticeManager, Database
Preconditions: The database receives a request for valid notices in a state.
Success/Postconditions: All the valid notices are returned.
Scenario (intentions):
1) The NoticeManager requests all the valid notices for a state from the Database.

Service: Start
Internal use case: Start
Actors: SystemManager, Database, Delivery
Preconditions: The system is started.
Success/Postconditions: The proper services are started to enable management and notice delivery.
1) The SystemManager creates the database to provide database access to the various agents of the system.
2) The SystemManager creates the delivery agent to enable the delivery of notices.

Internal use case: Create
Extends: Start, intention 2
Actors: SystemManager, Delivery, Notifier, DeliveryService
Preconditions: The delivery agent is created.
Success/Postconditions: The proper agents are created which allow the delivery agent to deliver notices.
1) The delivery agent creates the Notifier so it can be notified of new notices.
2) The delivery agent creates the DeliveryService agent, which it will hand the notices that need to be delivered to.
3) The delivery agent gives a list of notices to the Notifier, which it wishes to be notified for.
4) When the delivery agent receives a notice from the Notifier it hands the notice to the DeliveryService for delivery.

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. The following case study lists the beliefs that are needed for each goal in our Notice Management System.

Case Study: Agent Belief List

Service: CreateNotice
Goal: CreateNotice
Belief: NoticeDB
Reason: We need to know the database to submit notices to.

Goal: Create
Belief: StateDB
Reason: Agent needs to provide a list of districts for a state to the user.

Goal: Submit
Belief: NoticeDB
Reason: Agents needs access to the Notice DB to insert new notices.

Goal: WatchForNoticesToDeliver
Belief: NoticeDB
Reason: Agent needs to watch for new notices entering the NoticeDB.

Goal: DeliverNoticesToDistricts
Belief: StateDB

Reason: Agent formats the notice for delivery based upon how the states desire it delivered.

Service: ViewNotice
Goal: ViewNotice
Belief: StateDB
Reason: We need a district id for the notice we wish to view.

Goal: View
Belief: Notice
Reason: We need a notice to view.

Goal: GetValidNotices
Belief: StateDB, NoticeDB
Reason: We need a list of district ids to check for valid notices for those districts.

Service: Start
Goal: Start
Belief: Delivery, Database
Reason: This goal needs to know which delivery and database services to start.

Goal: Create
Belief: Notifier, DeliveryService
Reason: This goal needs to know, which Notifier and DeliveryService agents to start.

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.

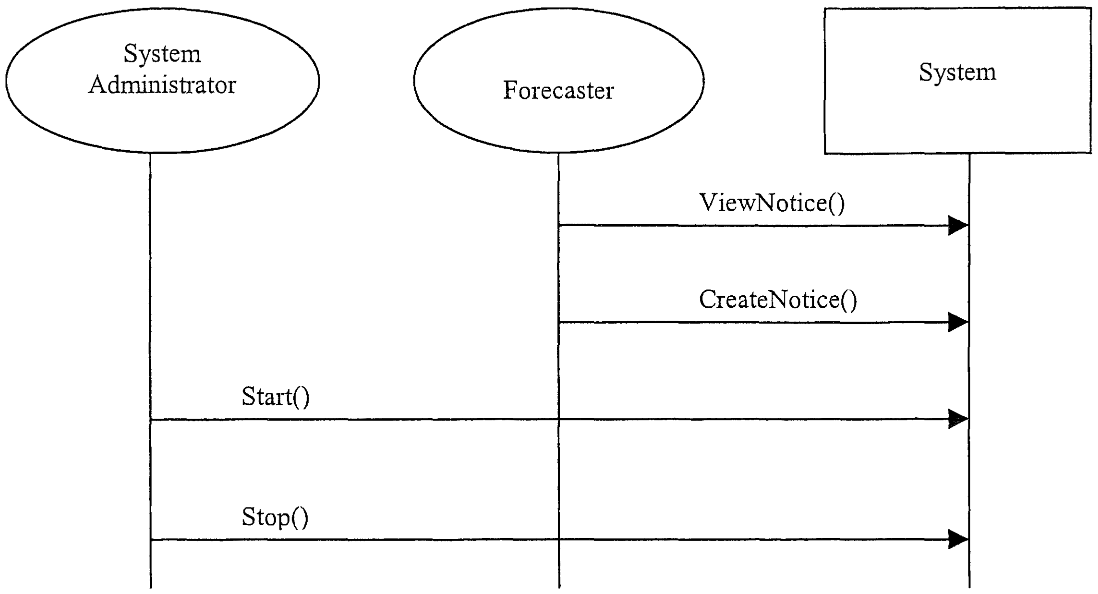

Figure 4. System Services Interaction Diagram.

Figure 4 is a variation of the agent interaction diagram that shows the services that will be provided by our system. The services that our system should provide can be extracted from the titles of the brief external use cases. This system service interaction diagram provides the developer with a visual picture of the external services that the system will provide. In our notice management case study we have described the CreateNotice, ViewNotice and Start services in detail. We choose not to provide a detailed documentation of the Stop service because it does not seem to provide any useful insight into the system architecture.

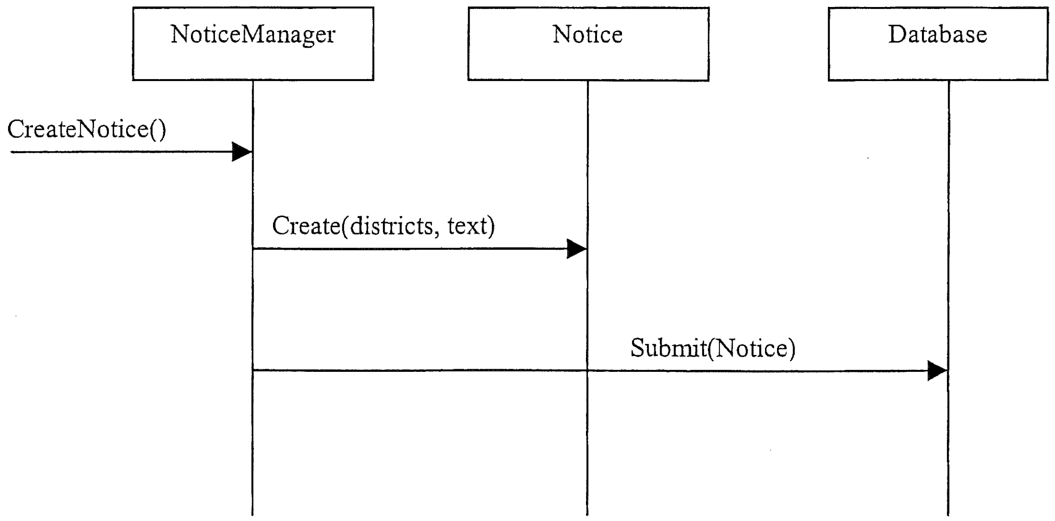

Figure 5. Agent Interaction Diagram: NoticeManager.CreateNotice(). The Forecaster invokes the CreateNotice goal of the NoticeManager. The NoticeManager then uses the Notice.Create() goal in order to create a Notice. The NoticeManager uses the Database.Submit(Notice) goal in order to store the notice in the database.

The agent interaction diagram in Figure 5 describes the internal actions that take place when an external agent (a Forecaster in this case) invokes the CreateNotice goal that can be found in Figure 4. The internal use cases for the CreateNotice goal are used for the creation of Figure 5. The CreateNotice goal should not be confused with the CreateNotice service. The CreateNotice service embodies all the goals listed under the CreateNotice service in our brief internal use cases.

Figure 6 details the internal actions that take place when the Start goal is invoked. We choose to have the SystemManager create both the Database agent and the Delivery agent in Figure 6. The Database agent meets the description of the long-lived agent pattern because it needs to be available to many different agents inside our system and it needs to be available for the entire life of the system. We have the SystemManager create the Delivery agent because we want notices to be delivered as soon as the system is started.

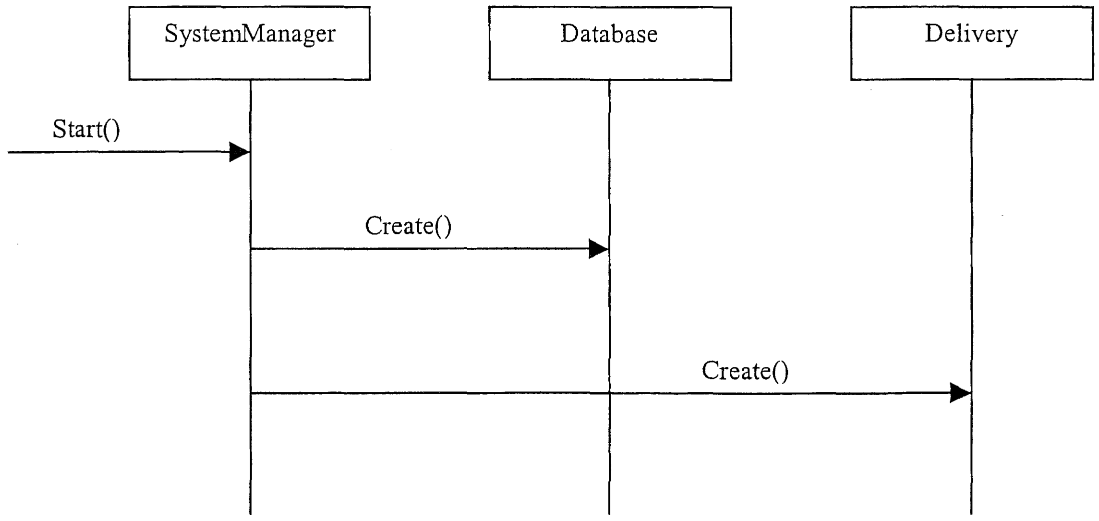

Figure 6. Agent Interaction Diagram: SystemManager.Start(). When the SystemManager.Start() goal is invoked by the SystemAdministrator then the SystemManager invokes the Create() goals for the Database and Delivery agents.

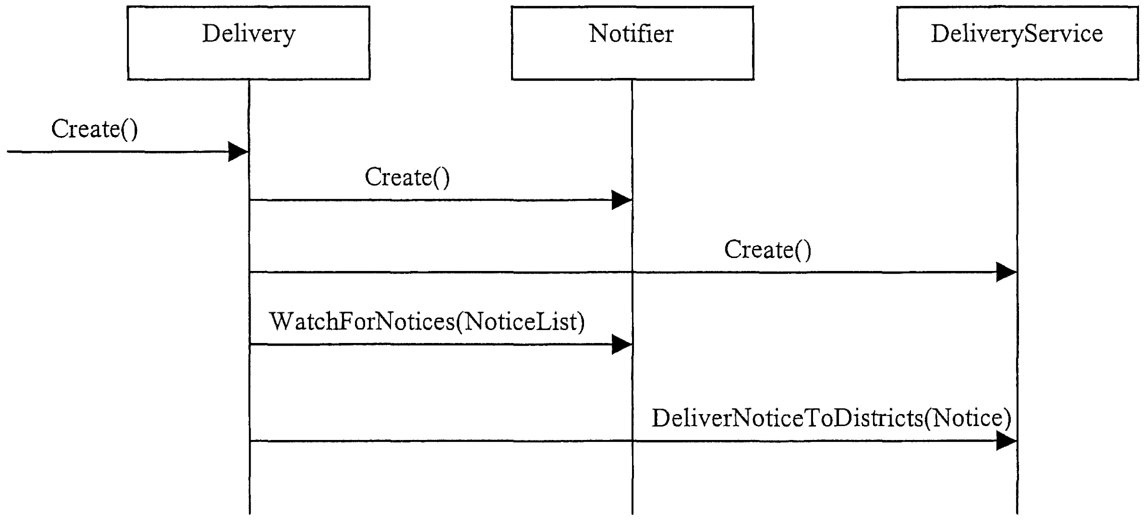

Figure 7. Agent Interaction Diagram: Delivery.Create(). When the Delivery agent receives a create request from the SystemManager it invokes the Notifier.Create(), DeliveryService.Create() and Delivery.WatchForNotices(NoticeList) goals. The Notifier will transparently pass new notices to the Delivery agent, described by the WatchForNotices(NoticeList) goal, which will then invoke the DeliveryService.deliverNoticeToDistricts(Notice) goal.

Figure 7 is a more detailed description of the actions that take place when the SystemManager invokes the Create goal of the Delivery agent in Figure 6. The Delivery agent creates the Notifier agent, so it can be receive new notices from the system. The Delivery agent creates the DeliveryService agent, which it will use to deliver any notices it receives. The Notifier.WatchForNotices(NoticeList) goal is invoked which tells the Notifier which notices we wished to be notified of. The DeliveryService.DeliverNoticeToDistricts(Notice) goal is invoked whenever a notice is received from the Notifier.

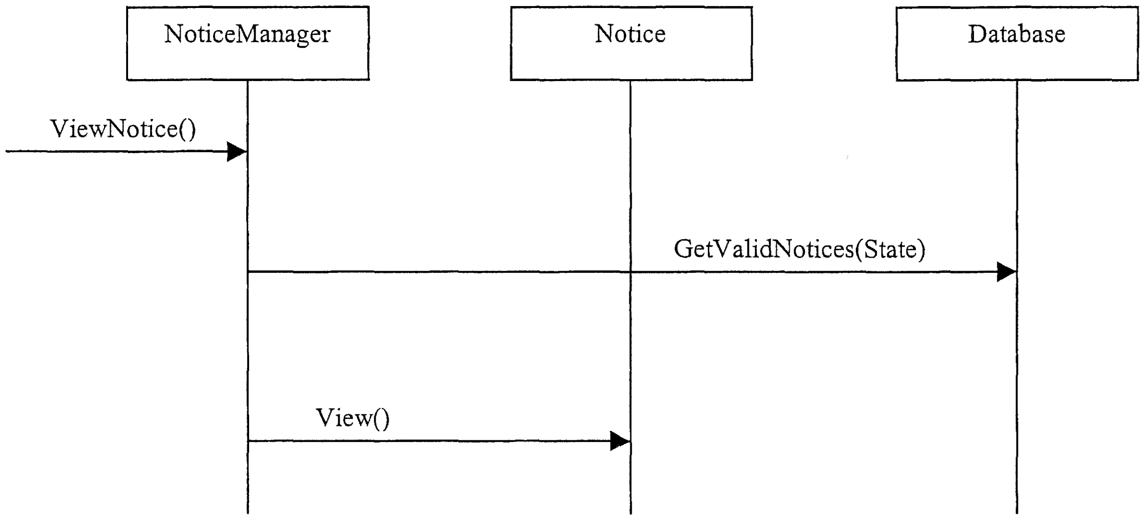

Figure 8. Agent Interaction Diagram: NoticeManger.ViewNotice(). The NoticeManager.ViewNotice() goal is invoked by the Forecaster. The NoticeMangar invokes the Database.GetValidNotices(State) goal and receives a list of Notices. The NoticeManager can then invoke the Notice.View() on each Notice.

Figure 8 is an agent interaction diagram that provides a more detailed description of what happens when an external agent invokes the systems ViewNotice goal. The Database.GetValidNotices(State) allows the NoticeManager to get the list of valid notices for a state from the database. The Notice.View() goal allows the NoticeManager to view the contents of each notice.

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 [Bellin et al. 1997]. 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. Describing an agent's static structure with a single artifact provides a valuable tool that can be used for constructing the agent in software. 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. The following case study lists the BDI agent cards that we created for our Notice Management System.

Case Study: BDI Agent Cards

Agent: NoticeManager
BDI list:
1) 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:
1) Forecaster asks the NoticeManager agent to create a Notice.
2) The NoticeManager agent provides an interface to the forecaster for notice creation.
3) NoticeManager agent properly formats and submits the notice to the database.

2)Desire: ViewNotice
Pre-conditions: Forecaster desires to view a notice area.
Belief: StateDB
Post-condition: Forecast is able to view the contents of a valid notice.
Collaborators: Forecaster (external)
Intentions:
1) Forecaster asks the NoticeManager agent to view a Notice.
2) The NoticeManager agent provides an interface to the forecaster for notice viewing.

Agent: Notice
BDI list:
1)Desire: Create
Pre-condition: NoticeManager needs to create a new notice.
Belief: StateDB
Post-condition: Notice is created.
Collaborators: NoticeManager, Database
Intentions:
1) The notice interface is started for notice creation.
2) The NoticeManager gets the State from the user.
3) The NoticeManager gets the district from the user.
4) The NoticeManager gets the notice text from the user.
5) The NoticeManager passes the DistrictIds and the notice text to the Notice.
6) A new notice is created.
7) The NoticeManager now has a notice.

2) Desire: View
Pre-condition: We need to view a notice.
Belief: Notice
Post-condition: A notice is viewed.
Collaborator: NoticeManager
Intentions:

1) The notice interface is started for notice viewing.
2) The NoticeManager gets the State from the user.
3) The NoticeManager gets the valid notices from the database.
4) The NoticeManager provides an interface with the districts that have valid notices.
5) The Forecaster selects a district to view a notice for.
6) The NoticeManager displays the notice contents to the Forecaster.

Agent: Delivery
BDI list:
1) Desire: Create
Preconditions: The delivery agent is created.
Belief: Notifer, DeliveryService
Success/Postconditions: The proper agents are created which allow the delivery agent to deliver notices.
Collaborators: SystemManager, Delivery, Notifier, DeliveryService
1) The delivery agent creates the Notifier so it can be notified of new notices.
2) The delivery agent creates the DeliveryService agent, which it will hand the notices that need to be delivered to.
3) The delivery agent gives a list of notices to the Notifier, which it wishes to be notified for.
4) When the delivery agent receives a notice from the Notifier it hands the notice to the DeliveryService for delivery.

Agent: DeliveryService
BDI list:
1) Desire: DeliverNoticesToDistricts
Pre-condition: We receive a notice that needs to be delivered.
Belief: StateDB
Post-condition: Notice is delivered in the proper format to the proper districts.
Collaborator: Delivery
Intentions:
1) The deliveryservice agent checks the notice for which it should be delivered too.
2) The deliveryservice agent then checks the database on how to properly format the notice for delivery.
3) The deliveryservice agent formats the notice for delivery.
4) The deliveryservice agent delivers the agent as a fax, web page or both.

Agent: Database
BDI list:
1) Desire: Submit
Pre-condition: The database agent receives a notice to save.
Belief: NoticeDB
Post-condition: The database agent stores the notice properly.
Collaborator: NoticeManager

Intentions:
1) The NoticeManager sends a notice to the database to be saved.

2) Desire: GetValidNotices
Pre-condition: The database receives a request for valid notices in a state.
Belief: NoticesDB, StateDB
Post-condition: All the valid notices are returned.
Collaborator: NoticeManager
Intentions:
1) The NoticeManager requests all the valid notices for a state from the Database.

Agent: Notifier
BDI list:
1) Desire: WatchForNotices
Pre-condition: The system needs to recognize when notices should be delivered.
Belief: NoticeDB
Post-condition: Waiting to receive notices.
Collaborator: Notifier
Intentions:
1) The delivery agent creates the notifier.
2) The delivery agent asks the notifier to watch the database for notices, which are a type of weather product.
3) The delivery agent listens to notifier for notices.
4) The notifier watches the database for notices to be entered.
5) The notifier gives the delivery agent a notice.
6) The delivery agent sends a request to the delivery service agent to delivery the agent.

Agent: SystemManager
1) Desire: Start
Pre-condition: The system is started.
Belief: Database, Delivery
Collaborator: System Administrator (external)
Intentions:
1) The SystemManger creates the database to provide database access to the various agents of the system.
2) The SystemManager creates the delivery agent to enable the delivery of notices.

CHAPTER IV
CONCLUSION

This is the final chapter summarizes our research and proposes some potential areas of research for the future.

Summary of Research

Software developers are continually called upon to develop increasingly complex systems. Computer scientists are constantly working on new tools that can aid software developers in the creation of these increasingly complex systems. We believe that agent-base software development will be useful tool for the construction of complex systems. 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 enties 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 like noun phase identification, but we also propose new tools like agent patterns to identify the potential agents in our system.

In constructing the BDI for each agent we take a goal-oriented approach. By using modified use cases we decompose the services for our system into one or more goals. Once the goals have been defined we use other artifacts like internal use cases to define the plan for each goal and agent belief lists to define the beliefs for each goal. We discover agents by assigning each goal and its corresponding belief and plan to a candidate agent in our process. Agent interaction diagrams and agent patterns are useful tools that can aid the assignment of each BDI to the proper agent.

We define the final architecture of our system with agent interaction diagrams and BDI agent cards. The agent interaction diagrams define the dynamic structure of our system and the BDI agent cards document the static structure of our system. Once the final architecture has been defined with the proper artifacts the system is ready to be created in software.

Future Research

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. 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

Beck, K. 2000. Extreme Programming Explained-Embrace Change, *Addison-Wesley*, 2000.

Bellin, David and Simone, Susan, The CRC Card Book, *Addison-Wesley*, 1997.

Booch, G., Object-Oriented Analysis and Design with Applications, *Addison Wesley*, 1994.

Booch, G., Rumbaugh, J., and Jacobson, I., The Unified Software Development Process, *Addison-Wesley*, 1999.

Bratman, M. E., Intention, Plans, and Practical Reason, *Harvard University Press*, 1987.

Cockburn, Alistair, Writing Effective Use Cases, *Addison-Wesley,* 2001.

Depke, Ralph, Heckel, Reiko, and Kuster, Jochen, Improving the Agent-Oriented Modeling Process by Roles, *AGENTS'01*, 640-647, June 2001.

Fowler, Martin and Scott, Kendall, UML Distilled Second Edition: A Brief Guide to the to the Standard Object Modeling Language, *Addison-Wesley*, 2000.

Gamma, E., r. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object Oriented Software, *Addison-Wesley*, 1995.

Hayden, Sandra, Carrick, Christina, and Yang, Qiang, A Catalog of Agent Coordination Patterns, ACM Press, 412-413, 1999.

Iglesias C. A., Garijo M, Gonzalez J. C., and Juan R. Velasco, Analysis and Design of Multiagent Systems using MAS-CommonKADS, In M.P. Singh, A. Rao, and M.J. Wooldridge, editors, *Proc. 4th Int. Workshop on Agent Theories, Architectures, and Languages (ATAL-97)*, volume 1365 of LNAI, 313-328, Springer-Verlag, July 24-26, 1998.

Jennings, Nicholas R., On agent-based software engineering, *Artificial Intelligence,* volume 117, 277-296, February 2000.

Jennings, Nicholas R., An Agent-Based Approach for Building Complex Software Systems, *Communications of the ACM*, 44(4), 35-41, April 2001.

Jo, Chang-Hyun, A Seamless Approach to the Agent Development, *ACM SAC 2001*, Las Vegas, 641-647, March, 2001.

Larman, Craig, Applying UML and Patterns: Second Edition, *Prentice-Hall*, 2002.

Petrie, Charles, Agent-Based Software Engineering, *Agent-Oriented Software Engineering, Lecture Notes in AI*, *Springer-Verlag,* 58-76, 2001.

Rao, Anand S. and Georgeff, Michael P., BDI Agents: From Theory to Practice, *Australian Artificial Intelligence Institute*, April, 1995.

Weiss G., editor, Multi-Agent Systems, *The MIT Press: Cambridge, MA*, 1999.

Wooldridge, M. and Jennings, N. R., Intelligent Agents: Theory and Practice, *Knowledge Engineering Review, Cambridge Univ. Press,* 10(2), 115-152, June 1995.

Wooldridge, M. and Jennings, N. R., Software Engineering With Agents: Pitfalls and Pratfalls, *IEEE Internet Computing*, 20-27, May-June 1999.

Wooldridge, M., Jennings, N. R., and Kinny, D., A Methodology for Agent-Oriented Analysis and Design, *Autonomous Agents 1999, Seattle, WA,* 69-76, 1999.

Wooldridge, M., Reasoning about Rational Agents, *The MIT Press: Cambridge, MA*, 2000.