Patterns in Agent-Oriented Software Engineering

Jürgen Lind

Ackerstraße 1
D-81541 München, Germany
jli@agentlab.de

Abstract. In this paper, I will show how the now popular concept of software patterns can be used in agent-oriented software engineering. To this end, I will present a possible structure of a pattern catalog for agent-oriented patterns and introduce a pattern description scheme that accounts for the specific needs of agent patterns. Then, I will provide two examples for how this scheme can be used to describe actual agent patterns.

1 Introduction

Patterns are everywhere. In all fields of our daily life, we use existing patterns to solve recurring problems or we develop new patterns for future use. In most cases, however, we do so subconsciously and are thus not aware of this powerful technique. Although todays software practice has shown that patterns are useful tools, no rationale for the transition from cognitive aspects into software engineering has been provided. Therefore, I will take a short detour to cognitive science at the beginning of this paper and discuss some mental aspects that are involved in using and developing solutions for recurring problems. We will, however, not completely ignore computer science aspects but instead explain the effects of the cognitive mechanisms that are introduced on computer programming and software engineering.

First of all, we need a model of the human memory such that we can establish a link between the abstractions on the cognitive level and those on the domain level. According to [Anderson, 1983], the human memory is organized in hierarchical levels of abstraction using so-called *cognitive units* as the building blocks. Whereas the knowledge on low levels of abstraction is represented with little abstraction from the pure sensory impressions, knowledge on higher levels of abstraction uses a more complex representation: *schemata*. Schemata are cognitive units with a rich internal structure [Détienne, 1990] that consists of several *slots* (variables) that can be instantiated with *slot fillers* (values). Partially instantiated schemata are *prototypes* for a particular concept, fully instantiated schemata are *exemplars* of this prototype. These exemplars can be ordered according to their semantic distance to their common prototype [Spada, 1990]. In the software engineering domain, schemata can be classified according to three different classes [Détienne, 1990]: *programming schemata*, *application domain schemata* and *discourse schemata*.

Programming schemata are either *variable plans* or *control-flow plans*. A variable plan, for example, contains the semantic knowledge about the concept of a counter variable that is used in loops and a typical control-flow plan is the programmers knowledge about the general process of iterating over list of arbitrary elements. In a real program, the software engineer will need both schemata (amongst others) to implement a concrete function e.g. to sum up a list of integer values.

Application domain schemata represent the engineers background knowledge about the application domain. It is a crucial task in the software engineering process to match the application domain knowledge and the programming g domain knowledge in order to develop or to understand a software system for the application domain.

Discourse schemata, finally, enable the software engineer to reason and communicate about programs besides functional aspects. These schemata include knowledge about general principles and conventions such as the convention that a variable should reflect its function or that particular variable names are used for certain tasks (e.g. `i` and `j` are typical variable names for loop variables).

For larger units of knowledge with many different interrelationships, schemata are not suited because of their rather descriptive nature. A more dynamic approach to the representation of large-scale cognitive units are *mental models* [Johnson-Laird, 1983]. These mental models contain declarative and procedural knowledge about a well-defined field and are often individualized scientific theories, e.g. about electricity [Spada, 1990]. Humans usually maintain a wide variety of these mental models and construct new models on demand, e.g. in the process of understanding the behavior of a complex system. The newly created models must be consistent with the existing models because these are resistant against changes and can only be revised with a certain learning effort [Spada, 1990].

The schemata discussed in the previous paragraphs are purely cognitive structures that normally have no external representation. To make the knowledge that is captured in the schemata available for re-use, we thus need a mechanisms to externalize the contained knowledge. Enter patterns.

Generally speaking, a pattern is a general solution to a specific recurring design problem. It explains the insight and good practices that have evolved to solve the problem and it provides a concise definition of common elements, context, and essential requirements for a solution. Patterns were first used in [Alexander, 1979] to describe constellations of structural elements in buildings and towns.

In [Lind, 2001], I have argued that software design is a general design task in that it is concerned with arranging a collection of primitive elements according to a given design language in order to achieve a particular goal [Grenno and Simon, 1988]. Examples for general design tasks can be found in technical disciplines such as architectural design or electrical circuit design, but also in cultural areas, e.g. in music composition or in writing an essay. Thus, design tasks are complex tasks that entail multiple subtasks that draw on different knowledge domains and a variety of cognitive processes [Pennington and Grabowski, 1990].

Although these subjects appear to be rather different in their nature, they nonetheless share two fundamental activities that are exercised during the overall design process [Pennington and Grabowski, 1990]. The first general aspect is *composition*, i.e. the process of developing a design by describing associations between the structural elements of the design. In terms of a software engineering process, this step maps *what* a program should achieve onto a detailed set of instructions that specify *how* these requirements are implemented in a particular programming language.

The second, and equally important aspect, is *comprehension*. Comprehension means to take a particular design and to understand the associations between its structural elements. The input for this process may be a design that was produced by a third party, but often it is a design that was developed by the same person. Now, why should it be necessary for a designer to understand something that he or she has developed? Simply because it is almost impossible to anticipate all implicit relations that are introduced as side-effects of one explicit design decision. For example, creating a new function for a particular purpose may have the side-effect that other, already existing functions can be simplified by using the newly created function. For the software engineer, the process of understanding a design is to map *how* a program implements a specification to *what* this specification entails.

The most important property of general design tasks is the evolutionary nature of the entire process. The design process is not sequential in that it proceeds from one intermediate product to the next until the design is completed, Rather, the process involves frequent revisions of previous decisions, re-structuring of the design elements or exploration of tentative solutions for particular sub-problems. Therefore, the design process often starts with constructing a kernel solution and then incrementally extending this solution until it meets the initial requirements [Kant and Newell, 1984] [Ratcliffe and Siddiqi, 1985]. In software development, the kernel solution is often retrieved by re-using existing code fragments and the applying a series of repeated modifications to these fragments until the target system is constructed [Green, 1990].

In this section, I have provided a general outline for why patterns are a suitable tool in software development as they are general solutions for recurring design patterns and software development is a general design task. Now, we will come to the main part of this paper and discuss the potential use of software patterns for intelligent agents and multiagent systems. We will start with a review of existing approaches in patterns in general and agent patterns in particular before we turn to the development of a patterns catalog structure for agent-oriented patterns. Then, I will propose a pattern description scheme that can be used for agent patterns and I will subsequently provide some brief examples on how the pattern description scheme can be used in practice.

2 Related Work

Several templates for software design patterns have been developed. The first attempt to bring the idea of software patterns into the broad public was undertaken by the ”Gang of Four” (GoF) [Gamma et al., 1994]. Their template is used for patterns that describe constellations of classes that solve common software design problems of relatively small scope. A similar approach is taken by a group of Siemens employees [Buschmann et al., 1996] who also provide a section. that discusses component structures of relatively large scope. What is common to these two approaches is the fact that they deal with very general problems that are to be found in almost any (sufficiently large) software system. In their second volume [Schmidt et al., 2000], the Siemens people therefore deal with patterns for large scale problems in greater detail.The idea of Software Architecture as a field of software development that can specifically benefit from a pattern oriented approach is also discussed in [Shaw, 1995]. Due to the great success of the early attempts to semi-formalize the description of software patterns, international conferences are conducted every year to enhance the information interchange in the pattern community; the best known of these conferences is the [PLoP, 2001].

Bringing together agents and other fields of software engineering might be difficult as the advantages of agent technology are still not widely recognized. In [da Silva and Delgado, 1998], for example, the entire agent approach is presented as a singular patterns among others. Clearly, this view on agents is much too limited and coarse grained and more elaborate pattern schemata for agents are necessary.

In the agent world itself, several attempts to introduce patterns and pattern languages have been made. In [Kendall et al., 1997], for example, a concrete pattern for layered agent architectures is presented using am ad-hoc pattern description scheme and without providing and outline for the organization of a agent-oriented pattern catalog. Similarly, [Wooldridge and Jennings, 1998] provides a collection of best practices but without adhering to a particular description scheme. In [Kendall, 1998a] and [Kendall, 1998b], a large collection of agent interaction schemata and generic role models is presented where each role and its relations to other roles are expressed in terms of CRC cards. Further approaches to introducing patterns into the agent world are to be found in [Aridor and Lange, 1998] or [Yasuyuki Tahara and Honiden, 1999]; an interesting approach for using agent patterns in a real world scenario is described in [Knublauch and Rose, 2001].

In summary, however, one must clearly admit that the idea of developing a standardized pattern catalog for agent-based systems has not been investigated in depth and that such an attempt is necessary in order to advance the development of agent-based applications.

3 Pattern Catalog Structure

The first problem in developing a pattern catalog for software patterns is to find an adequate structure of this catalog such that the patterns are grouped together in meaningful categories. In [Gamma et al., 1994], for example, the patterns that are discussed are classified into three classes: *creational patterns* deal with object creation, *structural patterns* describe the composition of classes and objects and *behavioral patterns*, finally, capture interaction between classes and objects. The approach taken in [Buschmann et al., 1996] differentiates between three categories of patterns: *architectural patterns*, *design patterns* and *idioms*. Architectural patterns describe fundamental structural organization schemata for software systems; this structure contains a set or predefined subsystems with well-defined responsibilities and it includes rules and guidelines for organizing the relationships between them. Design patterns, on the other hand, are schemata for refining subsystems or components or the relations between them. These patterns describe commonly recurring communication structures that solve general design problems within a particular context. Idioms, finally, are low-level patterns specific to a particular programming language. Idioms describe how particular aspects of a pattern are implemented in that language.

For agent patterns, I suggest using a catalog structure that follows the view oriented approach presented in [Lind, 2001]. The views that are introduced there are used to model the entire system from different perspectives where each of these perspectives captures a set of related aspects. The views that are proposed are as follows:

Interaction Interaction is a fundamental concept for a system that consists of multiple independent entities that coordinate themselves in order to achieve their individual as well as their joint goals. In this view, interaction within the target system is seen as a generalized form of conflict resolution that is not limited to a particular form such as communication. Instead, several generic forms of interaction exist that can be instantiated in a wide variety of contexts. The developer is encouraged to analyze the target problem with respect to the applicability of these generic forms before designing new forms. The most popular example for interaction is of course a communication protocol, simply because communication protocols have been studied for quite some time. However, multiagent systems that simulate physical environments or real physical multiagent systems such as robots or machines have many other possibilities of interaction besides communication and these forms of interaction must be allowed for in a general purpose method as well.

Role The role view determines the functional aggregation of the basic problem solving capabilities according to the physical constraints of the target system. A role is an abstraction that links the domain dependent part of the application to the agent technology that solves the problem under consideration. In my view, an agent consists of one or more role descriptions and an architecture that is capable of executing these role models which makes it important to aggregate the basic capabilities according to physical constraints.

Architecture (system, agent, agent management) The Architecture view is a projection of the target system onto the fundamental structural attributes with respect to the system design. The major aspects that are dealt with in this view are the system architecture as a whole and – due to the size and complexity of this particular aspect – the agent architecture. The system architecture is described according to various aspects and includes things such as agent management or database integration. The required agent architecture is characterized according to the requirements of the problem to be solved and it is strongly recommended that the system developer should at first try to select one of the numerous existing architectures before trying to develop a new architecture from scratch.

An important aspect that has to be dealt with in this view is to find the appropriate segregation between agents and objects. Just because agents provide a means for structuring a problem does not mean that they are necessarily the best means to do so [Collins and Ndumu, 1998]. Sometimes, it is better to implement particular abstractions as ordinary objects and thereby increase the system performance by avoiding the inevitable overhead associated with turning an object into an agent.

Society A society is a structured collection of entities that pursue a common goal. The goal of this view is to classify the society that either pre-exists within the organizational context of the system or that is desirable from the point-of-view of the system developer. According to this classification and to well defined quality measures for the performance of the target society that depend on application specific aspects, a society model is developed that is consistent with the roles within the society and that achieves the defined goals.

To illustrate how the quality measure affects the desirable society structure, consider, for example, Internet trading. In order to achieve the best trade, the number of participants in the trading process should be rather high in order to increase the chance of finding a profitable trade. On the other hand, a high number of participants also increases the computational and communicational overhead and thus a clustering of trading agents would increase the communicational and computational efficiency of the system. The final structure of the agent society (flat or clustered) thus depends on the quality measures (quality of the solution vs. efficiency).

System This view deals with systems aspects that affect several of the other views or even the system as a whole. The System view, for example, handles the user interface that controls the interaction between the system and the user(s) whose the task specific aspects are usually the input specification and the output presentation whereas task independent aspects deal with the visualization of the system activities in order to enable the user to follow the ongoing computations and interactions. Other aspects that are described in this view are the system-wide error-handling strategy, performance engineering and the system deployment once it has been developed.

Task In the Task view, the functional aspects of the target system are analyzed and a task hierarchy is generated that is then used to determine the basic problem solving capabilities of the entities in the final system. Furthermore, the nonfunctional requirements of the target system are defined and quantified as far as possible. Note that this view does not assume that a multiagent approach is used for the final system and therefore provides a rather high-level analysis of the problem.

In the case of a compiler application, for example, the basic functional requirement is that the system translates a program specified in a high-level language to a particular assembly language. The quality of the resulting code or the maximal tolerable time for the compilation are nonfunctional requirements and the basic problem solving capabilities are for example lexical analysis or code generation.

Environment In this view, the environment of the target system is analyzed from the developers perspective as well as from the systems perspective. These two perspectives usually differ as the developer has global knowledge whereas the system has only local knowledge. In the RoboCup domain [Noda, 1995], for example, the developer has access to the complete state of the system and its environment and this state is completely deterministic from this point-of-view. From the perspective of the individual agent within the system, on the other hand, only parts of the environment are accessible and the state transitions appear to be nondeterministic because of ongoing activities that cannot be perceived by the agent.

The advantage of using this collection of views as the categories of the agent patterns is that it adheres to a decomposition of the target software system that has shown to be quite effective in developing agent-based applications [Lind, 2001]. Still, the above collection of views should serve solely as a starting point as it is not yet clear whether a sufficient number of patterns can be found for each view. If it is difficult to find patterns for a particular view, the best idea is probably to abandon the respective view as a pattern category as it may be too volatile. A sufficient degree of stability is a prerequisite for successful pattern extraction.

4 A Pattern Description Scheme

Pattern description schemata are collections of aspects that, when taken together, fully capture a software pattern. The major advantage of such schemata is that they introduces a structured way to understand, explain and reason about patterns. Furthermore, they allows for a more effective communication among software engineers because a particular pattern description scheme provides the language to talk about patterns. In the existing literature, several general purpose schemata have been proposed; in Appendix A, I have briefly summarized two of them. As we can see, these two schemata are rather similar, suggesting that there might exist something as a "canonical" scheme that an be used for different pattern catalogs.

As valuable as the schemata summarized in the appendix are, I do not think that a single, general pattern description scheme is adequate for all categories of patterns, simply because it is too general. Furthermore, I also believe that pattern schemata for agents must be more complex than "normal" patterns because the usually deal with problems of coarser granularity. Therefore, I suggest to split up the pattern description scheme into two parts: a general part that deals with the generic properties of a pattern and a view-specific part that handles those aspects that are characteristic for the view the pattern belongs to two. In Table 1, I have summarized the resulting scheme, the view specific aspects will extend the last point mentioned there.

Name	A crisp name that captures the essential idea underlying the pattern
Aliases	, also known as
Problem	What problem is solved by the pattern
Forces	Which aspects of the problem are the forces that led to the development of the pattern? What are the prerequisites for using the pattern?
Entities	The entities that participate in the pattern. The name of the slot is chosen to avoid technical terms such as "class" or "object" to avoid a premature limitation on a particular implementation.
Dynamics	How do the entities of the pattern collaborate to achieve their goal.
Dependencies	Does the pattern require any specific environment before it can be applied?
Example	A simple, abstracted example for how to use the pattern.
Implementation	Hints on how the pattern may be implemented.
Known Uses	Examples of systems where the pattern has been applied successfully.
Consequences	What are the consequences of using the pattern? Does the pattern determine design decisions in other places of the system?
See Also	References to other patterns that solve similar problems or that can be beneficially combined with this pattern. Also, where are potential conflicts with other patterns.
View category specific fields	Additional fields that are specific for a particular category and that do not make sense for other categories.

Table 1. Description Fields in MASSIVE

5 Examples

In this section, I will demonstrate how the pattern description scheme from Table 1 can be used for agent-oriented software patterns. Due to the limited space, however, I shall restrict myself to some introductory remarks only.

5.1 Agent Architecture Patterns

The first example that we will discuss in this section stems from the architecture view and deals with the agent architecture. According to [Lind, 2001], the agent architecture defines a structural model of the components that constitute an agent as well as the interconnections of these components together with a computational model that implements the basic capabilities of the agent. In addition to the general purpose pattern fields, we will use the following view-specific fields that are used to capture the characteristics of a particular agent architecture.

Resource limitations This aspect of the characterization describes the resources that are available to a single agent within the multiagent system. If an agent has a very limited amount of processor time or memory space, it is impossible to use an agent architecture that requires, say, the resources of a Unix process. However, this point can also be viewed from a different angle. If the individual agent has to deal with very complex problems, a simple architecture may not be able to cope with the resource requirements of the architecture because it was not designed for heavy weight problems.

Control flow The aim of this requirement is to characterize the control flow that is needed within the agent. First of all, the designer should decide whether a sequential flow of control is sufficient or if the agent is required to do several things at the same time and thus needs some parallel action execution model. In the second case, a concurrent architecture that in most case is much more complex then a sequential architecture must be chosen.

Second, the designer must decide about the required flexibility of the control flow. In a more static setting, the flow of control can be explicitly hard-coded into the architecture while in a dynamic context, the flow of control is likely to undergo changes and must therefore be described implicitly e.g. in plan scripts that are interpreted at run time and that can be changed while the agent is in operation.

Knowledge handling The knowledge representation within the target system is defined in the Task view. In the Architecture view, the knowledge structures that are defined there must be characterized in order to decide which architectural features are necessary to effectively handle these structures.

First, it is important if the agents knowledge is stored explicitly in a knowledge base or is it encoded implicitly into the agent code. Second, the knowledge structures may be represented in a symbolic manner using some sort of logical formulae or in a sub-symbolic form e.g. in its simplest form as collection of values or more elaborate in the form of a neural network.

Reasoning capabilities The reasoning capabilities of the agents define the most important property and often determine the overall complexity of the agent architecture. For example, an agent may be forced to plan its actions if it is not a purely reactive agent, or it may use some utilitarian reasoning mechanisms to chose among several possible actions. Another important issue is the ability for an agent to learn from past experiences or the agent may be used to fulfill special tasks such as theorem proving etc.

Autonomy The degree of autonomy that is required by the agent defines how the agent interacts with its environment. A reactive agent simply responds to external stimuli by reproducing a pre-defined behavior when a particular stimulus is given by the environment. A pro-active agent, on the other hand, can become active without external trigger and then perform some action that satisfies the goal. Pro-active agents are usually more complex and their behavior is not always predictable.

User interaction The more interaction the agent has with the user, the more elaborate the user interface has to be in order to provide convenient means for input and output data. Furthermore, an advanced user interface agent will perform user profiling and try to learn the users preferences from his or her input/output behavior.

Temporal context This aspect characterizes the agents lifetime. Obviously, an agent with only a limited activation time will need another form of persistence mechanism – if any at all – then a long-running agent. Persistence refers to the ability of the agent to maintain knowledge structures over time and over unavoidable down-times due to service failures such as hardware or software crashes. However, the amount of information that is collected by the agent over its lifetime can become very large and must be handled in an effective manner. Thus, the architecture must provide means to manage the data handling process.

Decision making This attribute characterizes the way in which the agent comes to its decisions during its reasoning processes. While some authors claim that rationality is an inherent property of any agent [Russell and Wefald, 1991], [Russell and Norvig, 1995], there are others who consider architectures that support emotional decision making as an alternative [Burt, 1998], [André et al., 1999]. There are two main fields for a potential application of emotional architectures. First, they can become valuable tools to implement lifelike characters and avatars that represent a human user in networked environments. Second, the notion of emotions can be used to express complex heuristics for advanced software agent in a natural way. However, the development of the basic technology is still in its beginnings and does not play a relevant role until now. Still, the developer of a particular application may want to consider these ideas if they are appropriate for the problem in question.

We will no see a concrete example on how this scheme can be used to describe a particular agent architecture pattern. I have chosen the InteRRaP architecture as presented in [Müller, 1996] because it is well documented and therefore the pattern description scheme can be applied easily. Obviously, though, a complex thing such as a generic agent architecture requires a lot more space then that available in this paper. Therefore, the following paragraphs are solely intended to demonstrate how one might start with a pattern that captures the InteRRaP architecture; this is illustrated by the dots that appear in almost any field.

Name InteRRaP
Aliases none

Problem InteRRaP defines an agent architecture that supports situated behavior where the agents are able to recognize unexpected events and react timely and appropriately to them. Second, the InteRRaP architecture enables the agents to show goal-directed behavior in a way that the agent decides which goals to pursue by which means. Third, InteRRaP is designed in a way that the agents can act under real-time constraints and act efficiently with their resources. Fourth, InteRRaP agents must be able to interact with other agents in order to achieve common goals.

...

Forces InteRRaP was originally designed to bridge the gap between reactivity and deliberation on th one hand and interaction and coordination on the other. InteRRaP is rather heavy-weight architecture that should not be used in a system context with many agents with little resources. Ideally, only a single instance of an InteRRaP agent should run on a single computational node. Additionally, as InteRRaP implements a BDI architecture [Rao and Georgeff, 1995], it must be possible to model the problem domain in terms of BDI concepts.

...

Entities The major architectural abstraction of the InteRRaP agent architecture is *layering*. InteRRaP consists of three layers that serve different purposes:

Behavior Based Layer (BBL) This layer implements the reactive behavior of the agent, i.e. this layer reacts to external requirements without any explicit reasoning, thus it reacts very fast.

Local Planning Layer (LPL) This layer performs the planning process of an individual agent, it is also responsible to monitor the plan execution of the agents current plan.

Social Planning Layer (SPL) This layer is responsible for the coordination with the other agents within a multiagent system. The coordination with the other agents is achieved with explicit negotiation protocols.

These layers mainly serve two purposes. First, the represent different functionalities that compete for the resources of the agent. Second, the layering allows for better conceptual abstractions in terms of the agents knowledge representation and by restricting access to particular pieces of information to a particular layer.

...

Dependencies none

Example [left out for brevity]

Implementation [left out for brevity]

Known Uses Examples for the use of InteRRaP agent architecture can be found in [Müller, 1996] and [Jung, 1999].

Consequences InteRRaP agents in their strict form rely on modeling the agent behavior and the domain knowledge in terms of specific concepts that stem from the knowledge and plan representation schemata used. This can easily become a complex task even for domains with limited complexity. Alternatively, a light-weight version of InteRRaP agents should be used that does not strictly implement the original design and that uses a custom knowledge and plan representation scheme, making it easier to implement domain specific concepts. An example for this approach can be found in [Lind et al., 1999].

...

See Also BDI agents [Bratman et al., 1987], layered architectures [Kendall et al., 1997]

Control flow The control flow of an InteRRaP agent is guided mainly by two principal ideas: *Bottom-up activation* where control is shifted up a layer if the current layer is not competent to deal with the situation (and, of course, a higher layer exists); and *Top-down execution* where each layer uses operational primitives defined at the next lower layer to achieve its goals.

Each layer runs in a (potentially) infinite control loop that is independent from the other layers and that uses the bottom-up activation to hand over the flow of control to the next higher level and town-down execution either to receive operations to execute or to demand the execution of commands from the lower level.

...

Resource limitations As the InteRRaP architecture is rather complex, the agents that are implemented using the architecture will be rather heavy-weighted. Therefore, the computational resources that are available for a single agent must not be too limited in order to allow for a satisfactory performance of the agent. As the final resource requirements depend on a particular implementation of the generic architecture, no further details with respect to resources are possible.

Knowledge handling The InteRRaP agent architecture has a layered knowledge base where the beliefs of an agent are stored hierarchically; this restricts the amount of available information to a particular control layer.

The knowledge is stored using a knowledge representation scheme that consists of *Concepts*, *Types*, *Attributes Features* and *Relations*.

...

Reasoning capabilities InteRRaP uses the *Assertional Knowledge Base (AKB* to represent the agents current knowledge, its beliefs and its goals in terms of the concepts discussed in the **Knowledge handling** section of this pattern. The AKB offers three types of interface services: *assertional services* that allow to assert new beliefs into the knowledge base, i.e., to create instances of concepts and relations, and to change the values of attributes of existing concept and relation instances, *retrieval services* that provide access to beliefs that are actually stored in the AKB, and *active information services*, finally, that offer a possibility to access information from the knowledge base upon demand.

...

The main reasoning capabilities of InteRRaP agents are based on a planning mechanism known as *planning from second principles* [Schank and Abelson, 1977], [Bratman et al., 1987]. The planner is viewed as a black box, that is given a problem description and that then returns a plan that solves the problem. In InteRRaP, planning and plan execution are interleaved to cope with the dynamic environment of the agents. InteRRaP agents have a *plan library* that represent collections of plans for particular domain specific problems. The plans are represented in a special language allows the agent developer to link domain specific patterns of behavior with an abstract plan.

...

Autonomy The degree of autonomy exhibited by InteRRaP agents is domain dependent.

User interaction [generic architecture, not applicable]

Temporal context [generic architecture, not applicable]

Decision making [generic architecture, not applicable]

5.2 Interaction Protocol Patterns

The second example that we will discuss in this section are patterns for the interaction view. Here, we have the situation that the most important view-specific aspects of an interaction pattern are already covered in the part of the pattern description that deals with general aspects. These parts are the *roles*, the *messages* and the *temporal ordering* of the message exchange. In the general description scheme, the roles and the messages can be discussed in the "entities" section and the temporal ordering of the messages is part of the "dynamics" of the pattern. However, in order to point out the specific meanings of the three major concepts, I think it is a better idea to remove the fields from the general part of the description scheme and to add the three new fields to the view-specific part. That way, the special semantics of the filed names are emphasized.

To demonstrate how a concrete interaction protocol pattern might look, we will now briefly present a pattern description for the well-known Contract-net protocol as introduced in [Smith, 1980]. Remember that the following paragraphs serve as example only and therefore, the field contents contain introductory aspects only. Furthermore, we will only consider the simple Contract-net variant without task refinements etc.

Name Contract-net

Aliases none

Problem Allocate a particular task to one out of several potential contractors while minimizing the local costs.

Forces The Contract-net protocol is a one-to-one copy of the behavior shown by participants in market. The idea is to have the participants calculate their local cost for performing a particular task and then announcing the resulting cost to a manager who decides on the task assignment. This leads to an optimal resource allocation in the case of independent tasks. If inter-dependencies between tasks exist, the task allocation process can get caught in local minima.

Dependencies Using the Contract-net protocol for a task allocation process requires that there exist several potential contractors with comparable abilities. For optimal performance, the Contract-net also requires independent tasks.

Example [omitted]

Implementation See [Lind, 2001], Appendix C.

Known Uses See [Lind et al., 1999].

Consequences If the tasks are not independent from each other, the Contract-net protocol is likely to lead to sub-optimal solutions as it can get stuck in local optima. To escape from local optima, additional mechanisms may be necessary (see *See Also* below).

See Also Simulated Trading [Bachem et al., 1993] as used in [Lind et al., 1999].

Roles The Contract-net protocol has two roles: the *manager* and the *bidder*. The manager is responsible for putting together a task description that tells the potential bidders what to do. In closed agent societies, the task descriptions are usually given is some proprietary format; in open agent societies, some sort of standard must be applied. The second task of the manager is to determine the set of possible bidders. This can be achieved by broadcasting the task announcements to all possible receivers or by performing a pre-selection of already known candidates. While the first approach ensures that all potential contractors are reached, it is not very resource effective and

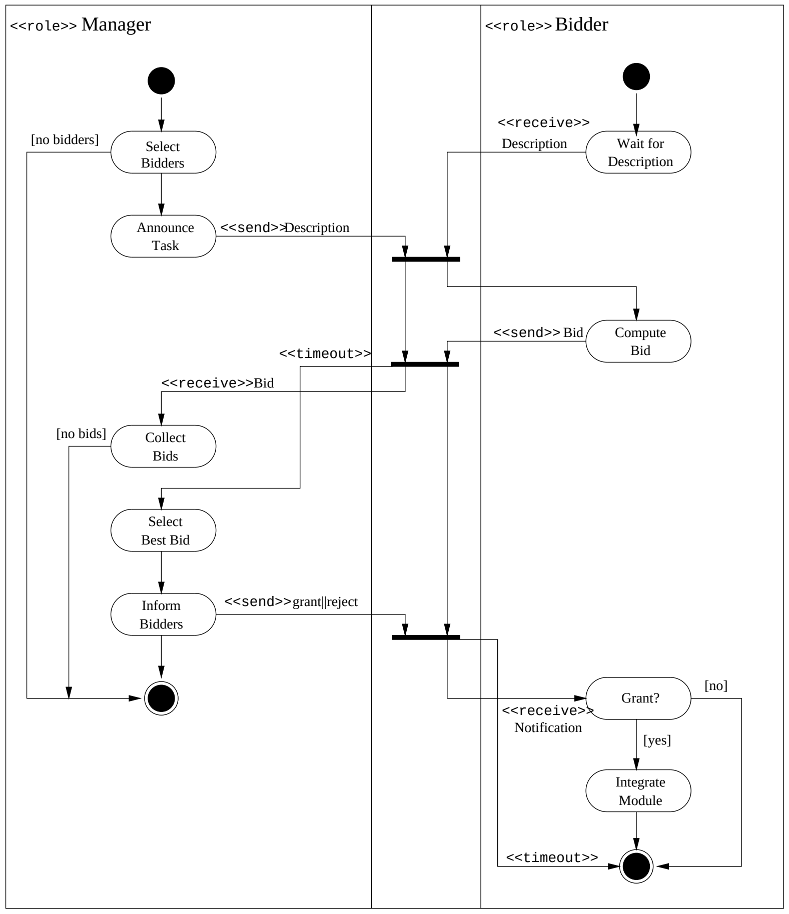

Fig. 1. Contract-net

can lead to unnecessary computations on the bidders side. Thus, a pre-selection of candidates is usually advantageous. The task of a bidder is to compute the cost for performing the announced task and to send an offer (if the bidder can fulfill the task at all) to the manager. If the manager grants the task to a particular bidder, the bidder guarantees task completion within the constraints provided by the task announcement.

Messages The Contract-net has four messages: *call for bids*, *bid*, *grant* and *reject*. The *call for bids* message is issued by the manager and contains the task description. The bidders reply to this message with *bid* message if they can perform the task. If not, no message is sent to the manager. After completing the bid collection phase, the manager selects the best bid and sends a *grant* to the successful bidder; the other bidders receive *reject* messages.

Temporal Ordering See Figure 1.

6 Conclusion

Software pattern have shown to be useful tools in many areas of software development; in this paper, a starting point for introducing patterns into agent-oriented software engineering has been presented. The approach integrates with the MASSIVEmethod but it is not limited to that particular development method. The basic idea that was discussed is to develop a pattern catalog that is structured according to the view-oriented decomposition of the final system as it is suggested in the MASSIVEmethod.

In forthcoming research, the suggested view system must be evaluated for its usefulness as a pattern catalog structure and the pattern description scheme must be analyzed with respect to its adequacy for capturing all relevant aspects of agent patterns. The most important task, however, is to start to collect agent patterns and to make them widely available.

References

[Alexander, 1979] ALEXANDER, C. (1979). *The timeless way of building*. Oxford University Press.

[Anderson, 1983] ANDERSON, J. R. (1983). *The Architecture of Cognition*. Harvard University Press, Cambridge, MA.

[André et al., 1999] ANDRÉ, E., M., K., GEBHARD, P., ALLEN, S., AND RIST, T. (1999). Integrating models of personality and emotions into lifelike characters. In *Affect in Interactions Towards a New Generation of Interfaces*.

[Aridor and Lange, 1998] ARIDOR, Y. AND LANGE, D. B. (1998). Agent design patterns: Elements of agent application design. In *Proceedings of the Second International Conference on Autonomous Agents*.

[Bachem et al., 1993] BACHEM, A., HOCHSTÄTTLER, W., AND MALICH, M. (1993). The Simulated Trading Heuristic for Solving Vehicle Routing Problems. Technical Report 93.139, Mathematisches Institut der Universität zu Köln.

[Bratman et al., 1987] BRATMAN, M. E., ISRAEL, D. J., AND POLLACK, M. E. (1987). Toward an architecture for resource-bounded agents. Technical Report CSLI-87- 104, Center for the Study of Language and Information, SRI and Stanford University.

[Burt, 1998] BURT, A. (1998). Emotionally Intelligent Agents: The Outline of a Resource-Oriented Approach. In *Proceedings of the 1998 AAAI Fall Symposium Emotional and Intelligent: The Tangled Knot of Cognition*.

[Buschmann et al., 1996] BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. (1996). *Pattern-oriented Software Architecture Vol 1: A System of Patterns*. John Wiley & Sons.

[Collins and Ndumu, 1998] COLLINS, J. AND NDUMU, D. (1998). The ZEUS Role Modelling Guide. Technical report, BT, Adastral Park, Martlesham Heath.

[da Silva and Delgado, 1998] DA SILVA, A. R. AND DELGADO, J. (1998). The agent pattern: A design pattern for dynamic and distributed applications. In *Proceedings of the EuroPLoP'98, Third European Conference on Pattern Languages of Programming and Computing*.

[Détienne, 1990] DÉTIENNE, F. (1990). Expert programmers and programming languages. In *Psychology of Programming*. Academic Press Ltd., London.

[Gamma et al., 1994] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Green, 1990] GREEN, T. R. G. (1990). Programming languages as information structures. In *Psychology of Programming*. Academic Press Ltd., London.

[Grenno and Simon, 1988] GRENNO, J. G. AND SIMON, H. A. (1988). Problem solving and reasoning. In ATKINSON, R. C., HERRNSTEIN, R. J., LINDZEY, G., AND LUCE, R. D., editors, *Stevens Handbook of Experimental Psychology*, volume 2. Wiley.

[Johnson-Laird, 1983] JOHNSON-LAIRD, P. N. (1983). *Mental Models*. Cmabridge University Press, London.

[Jung, 1999] JUNG, C. G. (1999). *Theory and Pratice of Hybrid Agents*. PhD thesis, Universität des Saarlandes.

[Kant and Newell, 1984] KANT, E. AND NEWELL, A. (1984). Problem solving techniques for the design of algorithms. *Human-Computer Interactions*, 28:97–118.

[Kendall, 1998a] KENDALL, E. A. (1998a). Agent Analysis and Design with Role Models. Technical report, British Telecom. Volume I: Overview.

[Kendall, 1998b] KENDALL, E. A. (1998b). Agent Analysis and Design with Role Models. Technical report, British Telecom. Volume II: Role Models for Agent Enhanced Workflow and Business Process Management.

[Kendall et al., 1997] KENDALL, E. A., PATHAK, C. V., KRISHNA, P. M., AND SURESH, C. (1997). The Layered Agent Pattern Language. In *Proceedings of the Conference on Pattern Languages of Programs (PLoP'97)*.

[Knublauch and Rose, 2001] KNUBLAUCH, H. AND ROSE, T. (2001). Werkzeugunterstützte Prozessanalayse zur Identifikation von Anwendungsszenarien für Agenten. In JABLONSKI, S., KIRN, S., PLAHA, M., SINZ, E. J., ULBRICH-VOM ENDE, A., AND WEISS, G., editors, *Verteilte Informationssysteme auf der Grundlage von Objekten, Komponenten und Agenten*. GI Fachgruppe 1.1.6 Verteilte Künstliche Intelligenz.

[Lind, 2001] LIND, J. (2001). *Iterative Software Engineering for Multiagent Systems - The* MASSIVE *Method*, volume 1994 of *Lecture Notes in Computer Science*. Springer.

[Lind et al., 1999] LIND, J., BÖCKER, J., AND ZIRKLER, B. (1999). Optimising the Operation Management with a Multi-Agent Approach - Using TCS as an Example. In *Proceedings of the World Congress on Railway Research (WCRR)*, Tokyo.

[Müller, 1996] MÜLLER, J. P. (1996). *The Design of Intelligent Agents: A Layered Approach*, volume 1177 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

[Noda, 1995] NODA, I. (1995). Soccer Server: A Simulator of Robocup. In *Proc. of AI symposium 1995*. JAPANESE SOCIETY FOR ARTIFICIAL INTELLIGENCE.

[Pennington and Grabowski, 1990] PENNINGTON, N. AND GRABOWSKI, B. (1990). The tasks of programming. In *Psychology of Programming*. Academic Press Ltd., London.

[PLoP, 2001] PLoP (2001). Pattern languages of programs. `http://jerry.cs.uiuc.edu/~/plop/`.

[Rao and Georgeff, 1995] RAO, A. S. AND GEORGEFF, M. (1995). BDI Agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, San Francisco, CA.

[Ratcliffe and Siddiqi, 1985] RATCLIFFE, B. AND SIDDIQI, J. A. (1985). An empirical investigation into problem decomposition strategies used in program design. *International Journal of Man-Machine Studies*, 22:77–90.

[Russell and Norvig, 1995] RUSSELL, S. AND NORVIG, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

[Russell and Wefald, 1991] RUSSELL, S. J. AND WEFALD, E. H. (1991). *Do the Right Thing : Studies in Limited Rationality*. MIT Press.

[Schank and Abelson, 1977] SCHANK, R. AND ABELSON, R. (1977). *Scripts, Plans, Goals, and Understanding.* Hillsdale:Erlbaum.

[Schmidt et al., 2000] SCHMIDT, D., STAL, M., ROHNERT, H., AND BUSCHMANN, F. (2000). *Pattern-oriented Software Architecture Vol 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons.

[Shaw, 1995] SHAW, M. (1995). Patterns for Software Architectures. In COPLIEN, J. AND SCHMIDT, D., editors, *Pattern Languages of Program Design*, volume I.

[Smith, 1980] SMITH, R. (1980). The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*.

[Spada, 1990] SPADA, H., editor (1990). *Allgemeine Psychologie*. Verlag Hans Huber, Bern.

[Wooldridge and Jennings, 1998] WOOLDRIDGE, M. J. AND JENNINGS, N. R. (1998). Pitfalls of agent-oriented development. In *Proceedings 2nd International Conference on Autonomous Agents (Agents-98)*, pages 385–391, Minneapolis.

[Yasuyuki Tahara and Honiden, 1999] YASUYUKI TAHARA, A. O. AND HONIDEN, S. (1999). Agent system development method based on agent patterns. In *Proceedings of the 1999 international conference on Software engineering*.

A Pattern Sescription Schemes

In [Gamma et al., 1994] a design pattern is described using the fields shown in Table 2; [Buschmann et al., 1996] uses the fields shown in Table 3.

Pattern Name	A good name is vital, because it will become part of your design vocabulary.
Intent	What does the design pattern do? What is its rationale and intend? What particular design issue or problem does it address?
Also Known As	Other well known names for the pattern, if any.
Motivation	A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you to unerstand the more abstract description of the pattern.
Applicability	What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?
Structure	A graphical representation of the classes in the pattern (OMT).
Participants	The classes and or objects in the design pattern and their responsibilities.
Collaborations	How do the participants collaborate to carry out their responsibilities.
Consequences	How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of the system structure does it let you vary independently?
Implementation	What pitfalls, hints or techniques should you be aware of when implementing the pattern? Are there language specific issues?
Sample Code	Code fragments that illustrate how you might implement the pattern in a particular language (C++)
Known Uses	Example of the pattern found in real systems. Preferably several (at least two) examples from different domains.
Related Patterns	What design patterns are closely related to this one? What are the important differences? With which of the other patterns should this one be used?

Table 2. Description Fields in [Gamma et al., 1994]

Name	The name and a short summary.
Also Known As	Other names, if any.
Example	A real world example that demonstrates the existinence of the problem and the need for the pattern.
Context	The situations in which the pattern may apply.
Problem	The problem the pattern addresses, including a discussion ofthe associated forces.
Solution	The fundamental solution principle underlying the pattern.
Structure	A detailed specification of the structural aspects of the pattern (OMT).
Dynamics	Typical scenarios describing the run-time behavior of the pattern.
Implementation	Guidelines for implementing the pattern.
Example Resolved	Discussion of any important asepcts resolving the example that are not covered elsewhere.
Variants	A brief description of variants or specializations of a pattern.
Known Uses	Examples of the use of the pattern, taken from exsiting systems
Consequences	The benefits the pattern provides, and any potential liabilities.
See Also	References to patterns that solve similar problems and to patterns that help us refine the pattern we are describing.

Table 3. Description Fields in [Buschmann et al., 1996]