A Knowledge Level Software Engineering Methodology for Agent Oriented Programming

Paper ID: 39

keywords: agent-based software engineering, design methodologies

ABSTRACT

Our goal in this paper is to introduce and motivate a methodology, called *Tropos*, for building agent oriented software systems. Tropos is based on two key ideas. First, the notion of agent and all the related mentalistic notions (for instance: beliefs, goals, actions and plans) are used in all phases of software development, from the early analysis down to the actual implementation. Second, Tropos covers also the very early phases of requirements analysis, thus allowing for a deeper understanding of the environment where the software must operate, and of the kind of interactions that should occur between software and human agents. The methodology is illustrated with the help of a case study.

1. INTRODUCTION

Agent oriented programming (AOP, from now on) is most often motivated by the need of open architectures that continuously change and evolve to accommodate new components and meet new requirements. More and more, software must operate on different platforms, without recompilation, and with minimal assumptions about its operating environment and its users. It must be robust, autonomous and proactive. Examples of applications where AOP seems most suited and which are most quoted in the literature are electronic commerce, enterprise resource planning, air-traffic control systems, personal digital assistants, or book travel arrangements, and so on (see for instance [13]).

To qualify as an agent, a software or hardware system is often required to have properties such as autonomy, social ability, reactivity, proactivity. Other attributes which are sometimes requested are mobility, veracity, rationality, and so on. The key feature which makes it possible to implement systems with the above properties, is that, in this paradigm, programming is done at a very abstract level, more precisely, following Newell, at the *knowledge level* [10]. Thus, in AOP, we talk of mental states, of beliefs instead of machine states, of plans and actions instead of programs, of communication, negotiation and social ability instead of interaction and I/O functionalities, of goals, desires, and so on. Mental notions provide, at least in part, the software with the extra flexibility needed in order to deal with the complexity intrinsic in the applications mentioned in the first paragraph. The explicit representation and manipulation of goals and plans allows, for instance, for a run-time "adjustment" of the system behavior needed in order to cope with unforeseen circumstances, or for a more meaningful interaction with other human and software agents.[1]

We are defining a software development methodology, called *Tropos*, which should allow us to exploit all the flexibility provided by AOP. In a nutshell, the two key and novel features of Tropos are the following:

1. The notion of agent and all the related mentalistic notions are used in all phases of software development, from the first phases of early analysis down to the actual implementation. In particular our target implementation agent language and system is JACK [2].

2. A crucial role is given to the earlier analysis of requirements that precedes prescriptive requirements specification. We consider therefore much earlier phases than the phases supported in, for instance, OOP software engineering methodologies. One such example are methodologies based on UML [1] where use case analysis is proposed as an early activity, followed by architectural design. As described in detail below, this move is crucial in order to achieve our objectives.

Our goal in this paper is to introduce and motivate the Tropos methodology, in all its phases. The presentation is carried out with the help of a running example. The example considered is a fragment of a substantial software system (which, in its full implementation, is requiring various man years of work) developed for the government of Trentino (Provincia Autonoma di Trento, or PAT). The system (which we will call throughout the *eCulture system*) is a web-based broker of cultural information and services for the province of Trentino, including information obtained from museums, exhibitions, and other cultural organizations and events. It is the government's intention that the system be usable by a variety of users, including Trentinos and tourists

[1]AOP is often introduced as a specialization or as a "natural development" of Object Oriented Programming (OOP), see for instance [12, 8, 13]. In our opinion, the step from OOP to AOP is more a paradigm shift than a simple specialization. Also those features of AOP which can be found in OOP languages, for instance, mobility and inheritance, take in this context a different and more abstract meaning.

looking for things to do, or scholars and students looking for material relevant to their studies.

The paper is structured as follows. Section 2 introduces the five basic steps of the Tropos methodology, namely, early requirement analysis, late requirements analysis, architectural design, detailed design, and implementation, and discusses its main novel points respect to other relevant works in AOP methodologies. The five Tropos phases are then described, as applied in the context of the *eCulture system* example, in Sections 3, 4, 5, 6 and 7. The conclusions are presented in section 8.

This paper follows on two previous papers, [9] and [3], which provide some motivations behind the Tropos project, and an early glimpse of how the methodology works. With respect to these earlier papers much more emphasis has been put on the issue of developing knowledge level specifications.

2. THE TROPOS METHODOLOGY: AN OVERVIEW

Tropos is intended to support five phases of software development:

- *Early requirements*, concerned with the understanding of a problem by studying an existing organizational setting; the output of this phase is an organizational model which includes relevant actors and their respective dependencies. Actors, in the organizational setting, are characterized by having goals that, in isolation, they would be unable to achieve; the goals are achievable in virtue of reciprocal means-end knowledge and dependencies [17].
- *Late requirements*, where the system-to-be is described within its operational environment, along with relevant functions and qualities; this description models the system as a (small) number of actors, which have a number of social dependencies with other actors in their environment;
- *Architectural design*, where the system's global architecture is defined in terms of subsystems, interconnected through data and control flows; in our framework, subsystems are represented as actors while data and control interconnections correspond to actor dependencies; in this step we specify actor capabilities and agents types (where agents are special kinds of actors, see below); this phase ends up with the specification of the system's agents;
- *Detailed design*, where each agent of the system architecture is defined in further detail in terms of internal and external events, plans and beliefs and agent's communication protocols;
- *Implementation*, where the actual implementation of the system is carried out, consistently with the detailed design. We use JACK, an agent programming platform, based on the BDI (Beliefs-Desires-Intentions) agent architecture for this phase.

The idea of paying attention to the activities that precede the specification of the initial, prescriptive requirements, such as understanding how the intended system would meet the organizational goals, is not new. It was first proposed in the requirements engineering literature (see for instance [5, 16]). In particular we adapt ideas from Eric Yu's model for requirements engineering, called ***i****, which offers actors, goals and actor dependencies as primitive concepts [16].[2] The main motivation underlying this earlier work was to develop a richer conceptual framework for modeling processes which involve multiple participants (both humans and computers). The goal was to have a more systematic reengineering of processes. One of the main advantages is that, by doing this kind of analysis, one can also capture not only the *what* or the *how* but also the *why* a piece of software is developed. This in turn allows for a more refined analysis of the system dependencies and, in particular, for a much better and uniform treatment not only of the system's functional requirements but also of the non-functional requirements (the latter being usually very hard to deal with).

Neither Yu's work, nor, as far as we know, any of the previous work in requirements analysis was developed with AOP in mind. The application of these ideas to AOP, and the decision to use mentalistic notions in all the phases of analysis has important consequences. When writing agent oriented specifications and programs one uses the same notions and abstractions used to describe the behavior of the human agents, and the processes involving them. The conceptual gap from *what* the system must do, and *why*, and *what* the users interacting with it must do, and *why*, is reduced to a minimum, thus providing (part of) the extra flexibility needed to cope with the complexity intrinsic in the applications mentioned in the introduction.

Indeed, the software engineering methodologies and specification languages developed in order to support OOP essentially support only the phases from the architectural design downwards. At that moment, any connection between the intentions of the different system components (human and software) cannot be explicitly specified. By using UML, for instance, the software engineer can start with the use case analysis (possibly refined by developing some activity diagrams) and then moves to the architectural design. Here he can do static analysis using class diagrams, or dynamic analysis using, for instance, sequence or interaction diagrams. The target is to get to the detail of the level of abstraction allowed by the actual classes, methods and attributes used to implement the system. However, applying this approach and the related diagrams to AOP misses most of the advantages coming for the fact that in AOP one writes programs at the knowledge level. It forces the programmer to translate goals and the other mentalistic notions into software level notions, for instance the classes, attributes and methods of class diagrams. The consequent negative effect is that the former notions must be reintroduced in the programming phase, for instance by writing JACK code: the programmer must program goals, beliefs, and plans, having lost the connection with the original mentalistic notions used in the early and late requirements. The work on AUML [11], though relevant in that it provides a first mapping from OOP to AOP specifications, is an example of work suffering from this kind of problem.

In the following sections we will present the five Tropos phases as applied in the context of the *eCulture system* example.

[2] ***i**** has been applied in various application areas, including requirements engineering [15], business process reengineering [19], and software modeling processes [18].

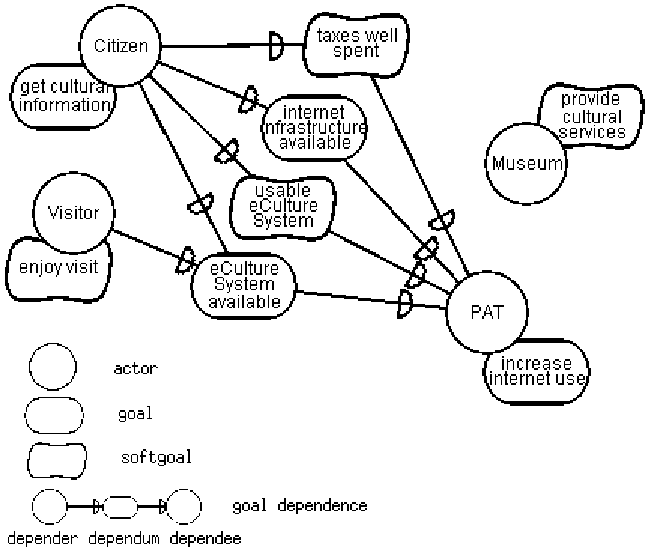

Figure 1: An actor diagram specifying the stakeholders of the eCultural project and their main goal dependencies.

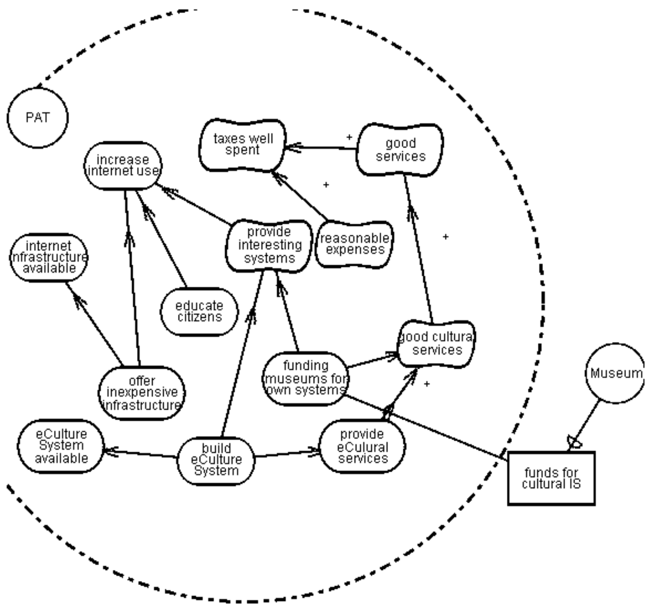

Figure 2: A rational diagram for `PAT`. The rectangular box added to a dependency, models a resource dependency.

3. EARLY REQUIREMENTS

During early requirements analysis, the requirements engineer models and analyzes the intentions of stakeholders. Following ***i****, in Tropos, the stakeholders' intentions are modeled as goals which, through some form of a goal-oriented analysis, eventually lead to the functional and non functional requirements of the system-to-be. Early requirements are assumed to involve social actors who depend on each other for goals to be achieved, tasks to be performed, and resources to be furnished. Tropos includes *actor diagrams* for describing the network of social dependency relationships among actors, as well as *rationale diagrams* for analyzing and trying to fulfill goals through a means-ends analysis.[3] These primitives are formalized using intentional concepts from AI, such as goal, belief, ability, and commitment.

An actor diagram is a graph, where each node represents an actor, and a link between two actors indicates that one actor depends, for some reason, on the other in order to attain some goal. We call the depending actor the depender and the actor who is depended upon the dependee. The object around which the dependency centers is called the dependum (see, e.g., Figure 1). By depending on another actor for a dependum, an actor is able to achieve goals that it would otherwise be unable to achieve on its own, or not as easily, or not as well. At the same time, the depender becomes vulnerable. If the dependee fails to deliver the dependum, the depender would be adversely affected in its ability to achieve its goals.

In our eCulture example we can start by informally listing (some of) the stakeholders:

- *Provincia Autonoma di Trento (PAT)*, that is the government agency funding the project; their objectives include improving public information services, increase tourism through new information services, also encouraging Internet use within the province;
- *Museums*, that are major cultural information providers for their respective collections; museums want government funds to build/improve their cultural information services, and are willing to interface their systems with the eCulture system;
- *Visitors*, who will want to access cultural information before or during their visit to Trentino to make their visit interesting and/or pleasant;
- *(Trentino) Citizens*, who want easily accessible information, of any sort.

These stakehorders are made correspond to some actors in an actor diagram. More precisely, notice that citizens and visitors correspond to (human) agents while this is not the case for the other two stakeholders. Museums and PAT correspond, rather, to roles. An actor is an *agent*, a *role* or a *position*, according to the fact that the actor is a well identified (human or software) entity (agent), it is a function (role) that can be played by an agent, or collects a set of roles that are usually played by a single agent (position).

Figure 1 shows the actors involved in the *eCulture project* and their respective goals. In particular, `PAT` is associated with a single relevant goal: **`increase internet use`**, while `Visitor` and `Museum` have associated softgoals, **`enjoy visit`** and **`provide cultural services`** respectively. Softgoals are distinguished from goals because they don't have a formal definition, and are amenable to a different (more qualitative) kind of analysis (see [4] for a detailed description of softgoals). `Citizen` wants to get cultural information and depends on `PAT` to fulfill the softgoal **`taxes well spent`**, a

[3]In ***i**** actor diagrams are called *strategic dependency models*, while rationale diagrams are called *strategic rationale models*.

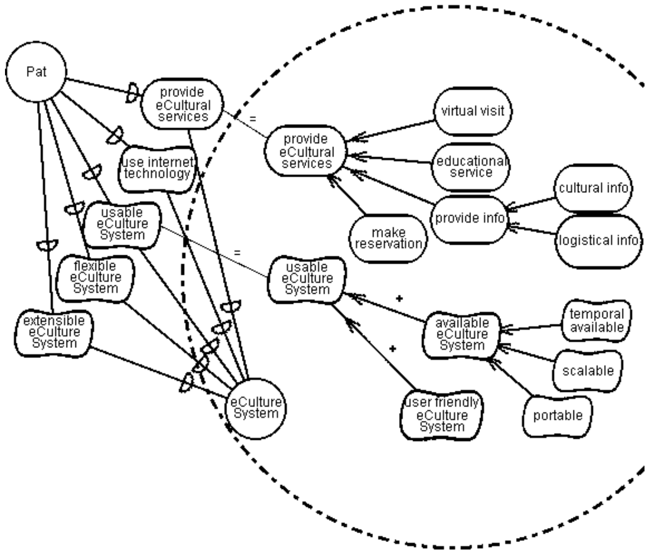

Figure 3: A fragment of the actor diagram including the `PAT` and the `eCulture System` actors; the rationale diagram for the `eCulture System` is detailed within the balloon.

high level goal that motivates more specific `PAT`'s responsibilities, namely to provide an Internet infrastructure, to deliver on the `eCulture system` and make it usable too.

The early requirements analysis goes on extending the actor diagram by incrementally adding more specific actor dependencies which come out from a means-end analysis of each goal. We specify this analysis using rationale diagrams. Figure 2 depicts a fragment of one such diagram, obtained by exploding part of the diagram in Figure 1, where the perspective of `PAT` is modeled. The diagram appears as a balloon within which `PAT`'s goals are analyzed and dependencies with other actors are established. This example is intended to illustrate how means-ends analysis is conducted. Throughout, the idea is that goals are decomposed into subgoals and positive/negative contributions of subgoals to goals are specified. Thus, in Figure 2, the goals `increase internet use` and `eCulture system available` are both well served by the goal `build eCulture System`. The (high level) softgoal `taxes well spent` gets two positive contributions, which can be thought as justifications for the selection of particular dependencies. The final result of this phase is a set of strategic dependencies among actors, built incrementally by performing means-end analysis on each goal, until all goals have been analyzed. The later it is added, the more specific a goal is. For instance, in the example in Figure 2 `PAT`'s goal `build eCulture system` is introduced last and, therefore, has no subgoals and it is motivated by the higher level goals it fulfills.[4]

4. LATE REQUIREMENTS

During late requirement analysis the system-to-be (the *eCulture System* in our example) is described within its operating environment, along with relevant functions and qualities. The system is represented as one or more actors which have a number of dependencies with the actors in their environment. These dependencies define all functional and non-functional requirements for the system-to-be.

Figure 3 illustrates the late requirements actor diagram where the `eCulture System` actor has been introduced. The `PAT` depends on it to `provide eCultural services`, one of the `PAT`'s subgoals discovered during the means-end analysis depicted in Figure 2. The softgoal `usable eCulture system`, for which `Citizen` depends on `PAT` (see Figure 1), has been delegated by `PAT` to the `eCulture system`. Moreover, the `eCulture System` is expected to fulfill other `PAT` softgoals such as `extensible eCulture system`, `flexible eCulture system`, and `use internet technology`. The balloon in Figure 3 shows how two of the `PAT`'s dependums can be further analyzed from the point of view of the `eCulture System`. The goal `provide eCultural services` is decomposed (AND decomposition) into four subgoals: `make reservation`, `provide info`, `educational services` and `virtual visit` that can be further specified along a subgoal hierarchy. For instance, the types of information that the system has to provide are both logistical (timetables and visiting instructions for museums), and cultural (for instance, cultural content of museums and special cultural events).

The rationale diagram includes also a softgoal analysis. The `usable eCulture system` softgoal has two positive (+) contributions from `user friendly eCulture system` and `available eCulture system`. This latter softgoal in turns specifies the following three basic requirements: system portability, scalability, and availability over time.

Starting from this analysis the system-to-be actor can be decomposed into sub-actors that take on the responsibility of fulfilling one or more goals of the system. Figure 4 shows the resulting `eCulture System` actor diagram: the

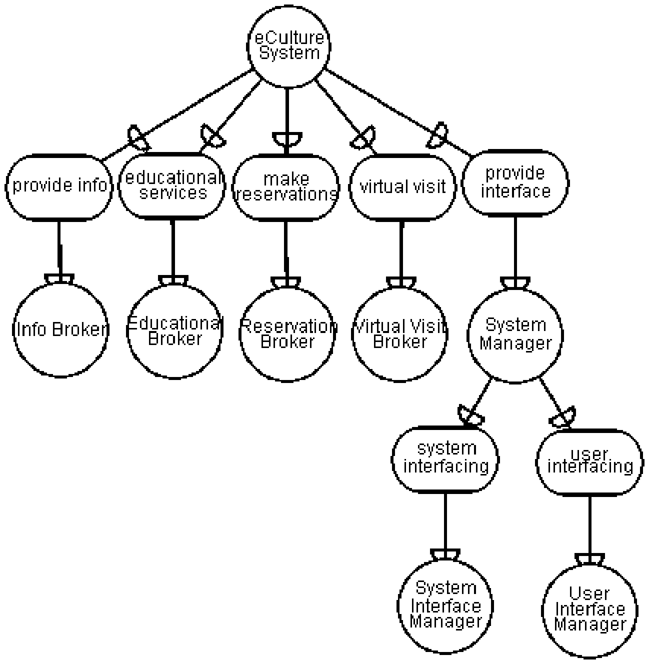

Figure 4: The system actor diagram. Sub-actors decomposition for the *eCulture System*.

[4]In rationale diagrams one can also introduce tasks and resources and connect them to the fulfillment of goals.

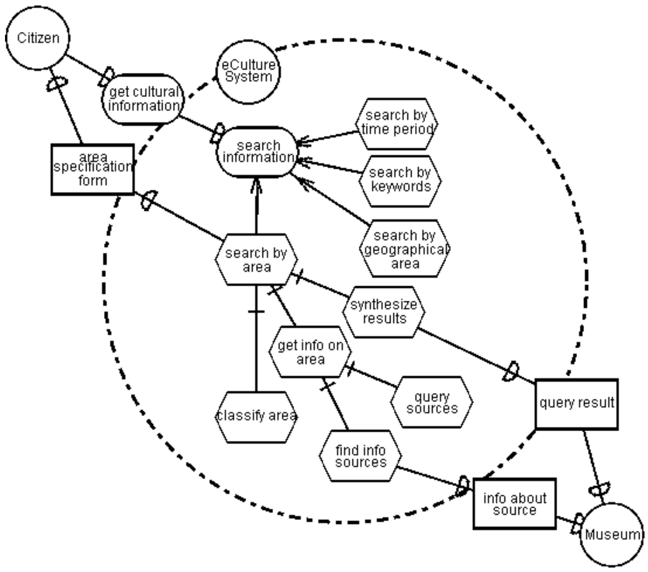

Figure 5: Rationale diagram for the goal `get cultural information`. Hexagonal shapes model tasks. Task decomposition links model task-subtask relationships. Goal-task links are a type of means-ends links.

`eCulture System` depends on the `Info Broker` to `provide info`, on the `Educational Broker` to provide `educational services`, on the `Reservation Broker` to `make reservation`, on the `Virtual Visit Broker` to `provide virtual visit`, and on the `System Manager` to `provide interface`. Furthermore each sub-actor can be further decomposed in sub-actors responsible for the fulfillment of one or more sub-goals.

At this point of the analysis we can look into the actor diagram for a direct dependency between the `Citizen`, which plays the role of system user, and the `eCulture System`. In other words we can now see how the former `Citizen`'s goal `get cultural information` can be fulfilled by the current `eCulture System`. The rational diagram of this goal dependency, see Figure 5, provides a sort of use-case analysis [7].

5. ARCHITECTURAL DESIGN

The architectural design phase consists of three steps:

1. refining the system actor diagram
2. identifying capabilities and
3. assigning them to agents.

In the first step the system actor diagram is extended according to design patterns [6] that provide solutions to heterogeneous agents communication and to non-functional requirements.[5] Figure 6 shows the extended actor diagram with respect to the *Info Broker*.[6] The `User Interface Manager` and the `Sources Interface Manager` are responsible for interfacing the system to the external actors `Citizen` and `Museum` respectively.

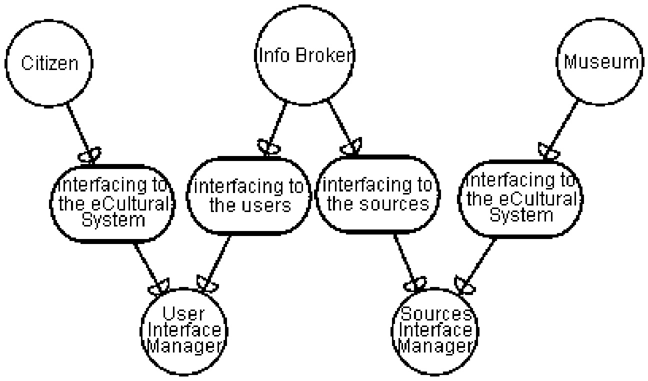

Figure 6: Extended actor diagram, `Info Broker`.

The second step consists in capturing actor capabilities from the analysis of the tasks that actors and sub-actors will carry on in order to fulfill functional requirements (hard goals). A capability is the set of *events*, *plans* and *beliefs* necessary for the fulfillment of actor goals and tasks. Figure 7 shows an example for the `Info Broker` actor analysis, with respect to the goal of searching information by topic area. The `Info Broker` is decomposed into three sub-actors: the `Area Classifier`, the `Results Synthesizer`, and the `Info Searcher`. The `Area Classifier` is responsible for the classification of the information provided by the user. It depends on the `User Interface Manager` for interfacing to the users. The `Info Searcher` depends on the `Area Classifier` to have information about the thematic area that the user is interested in and on the `Sources Interface Manager` for interfacing to the sources (museums). The `Results Synthesizer` depends on the `Info Searcher` for the information concerning the pending query and on the `Museum` to have the query results.

Capabilities can be easily identified by analyzing the diagram in Figure 7. In particular each dependency relationship can give place to one or more capabilities triggered by external events. Table 1 lists the capabilities associated to the extended actor diagram of Figure 7. They are listed with respect to the system-to-be actors, and then numbered in order to eliminate possible copies whereas.

The last step of the architectural design consists in defining a set of agent types and in assigning to each agent one or more different capabilities (agent assignment). Table 2 reports the agents assignment with respect to the capabilities listed in Table 1. The capabilities concern exclusively the task search by area assigned to the *Info Broker*. Of course, many other capabilities and agent types are needed in case we consider all the goals and tasks associated to the complete extended actor diagram.

In general, the agents assignment is not unique and depends on the designer. The number of agents and the capabilities assigned to each of them are choices driven by the analysis of the extend actor diagram and by the way in which the designer thinks the system in term of agents. Some of the activities done in architectural design can be compared to what Wooldridge et al. propose to do within

[5]In this step design patterns for agent systems are mapped to actor diagrams.

[6]For the sake of readability we do not show all the actors needed to take into account other non-functional requirements, e.g., system extensibility and user friendliness.

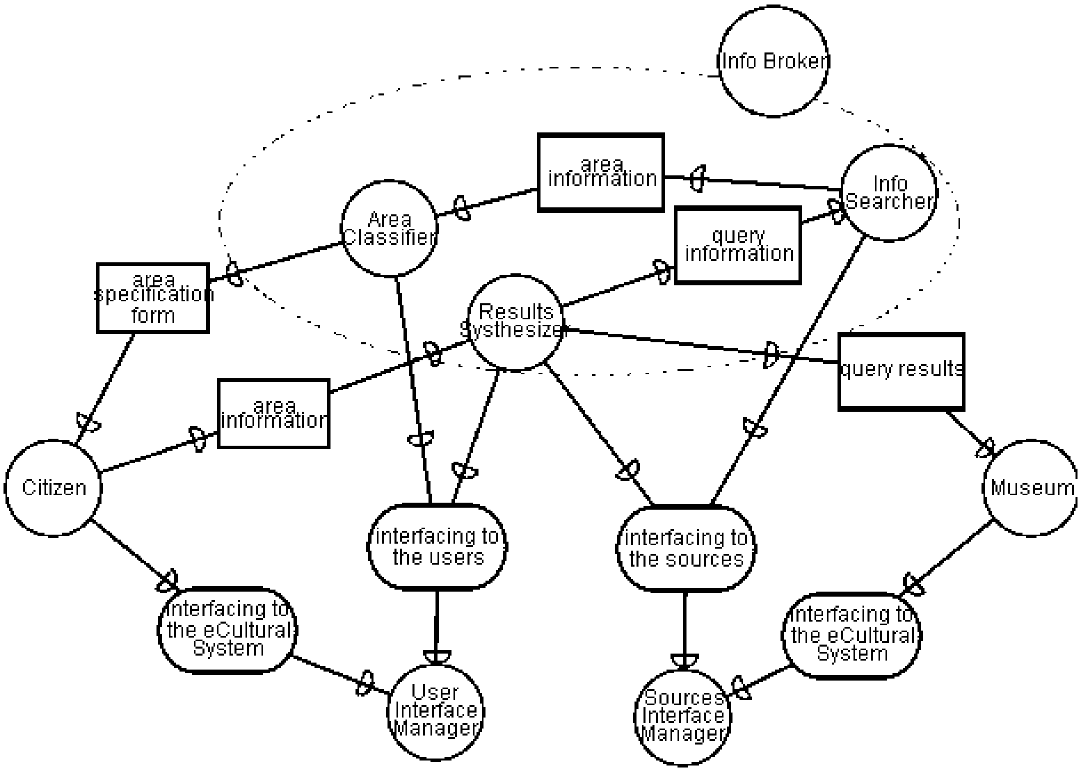

Figure 7: Actor diagram for capability analysis, `Info Broker`.

the Gaia methodology [14]. For instance what we do in actor diagram refinement can be compared to "role modeling" in Gaia. We instead consider also non-functional requirements. Similarly, capability analysis can be compared to "protocols modeling", even if in Gaia only external events are considered.

6. DETAILED DESIGN

The detailed design phase aims at specifying agent capabilities and interactions. The specification of capabilities amounts to modeling external and internal events that trigger plans and the beliefs involved in agent reasoning. Practical approaches to this step are often used.[7] In the paper we adapt a subset of the AUML diagrams proposed in [11]. In particular:

1. *capability diagrams.* The AUML activity diagram allows to model a capability (or a set of correlated capabilities), from the point of view of a specific actor. External events set up the starting state of a capability diagram, activity nodes model plans, transition arcs model events, beliefs are modeled as objects. For instance Figure 8 depicts the capability diagram of the `query results capability` of the *User Interface Agent.*
2. *plan diagrams.* Each plan node of a capability diagram can be further specified by AUML action diagrams.
3. *agent interaction diagrams.* Here AUML sequence diagrams can be exploited. In AUML sequence diagrams, agents corresponds to objects, whose life-line is independent from the specific interaction to be modeled (in UML an object can be created or destroyed during the interaction); communication acts between agents correspond to asynchronous message arcs. It can be shown that sequence diagrams modeling Agent Interaction Protocols, proposed by [11], can be straightforwardly applied to our example.

7. IMPLEMENTATION USING A BDI ARCHITECTURE

The BDI platform chosen for the implementation is JACK Intelligent Agents, an agent-oriented development environment built on top and fully integrated with Java. Agents in JACK are autonomous software components that have explicit goals (desires) to achieve or events to handle. Agents are programmed with a set of plans in order to make them capable of achieving goals.

The implementation activity follows step by step, in a natural way, the detailed design specification described in section 6. In fact, the notions introduced in that section have a direct correspondence with the following JACK's constructs, as explained below:

- Agent. A JACK's agent construct is used to define the behavior of an intelligent software agent. This includes the capabilities an agent has, the types of messages and

[7]For instance the *Data-Event-Plan diagram* used by JACK developer. Ralph Rönnquist, personal communication.

Actor Name	N	Capability
Area Classifier	1	Get area specification form
	2	Classify area
	3	Provide area information
	4	Provide service description
Info Searcher	5	Get area information
	6	Find information source
	7	Compose query
	8	Query source
	9	Provide query information
		Provide service description
Results Synthesizer	10	Get query information
	11	Get query results
	12	Provide query results
	13	Synthesize area query results
		Provide service description
Sources Interface Manager	14	Wrap information source
		Provide service description
User Interface Manager	15	Get user specification
	16	Provide user specification
	17	Get query results
	18	Present query results to the user
		Provide service description

Table 1: Actors capabilities

Agent	Capabilities
Query Handler	1, 3, 4, 5, 7, 8, 9, 10, 11, 12
Classifier	2, 4
Searcher	6, 4
Synthesizer	13, 4
Wrapper	14, 4
User Interface Agent	15, 16, 17, 18, 4

Table 2: Agent types and their capabilities

events it responds to and the plans it uses to achieve its goals.

- Capability. A JACK's capability construct can include plans, events, beliefs and other capabilities. An agent can be assigned a number of capabilities. Furthermore, a given capability can be assigned to different agents. JACK's capability provides a way of applying reuse concepts.

- Belief. Currently, in Tropos, this concept is used only in the implementation phase, but we are considering to move it up to earlier phases. The JACK's database construct provides a generic relational database. A database describes a set of beliefs that the agent can have.

- Event. Internal and external events specified in the detailed design map to the JACK's event construct. In JACK an event describes a triggering condition for agents actions.

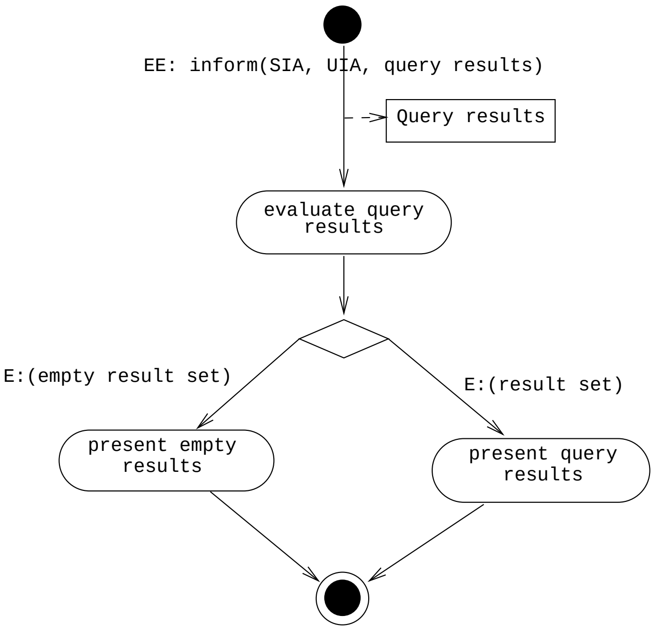

Figure 8: Capability diagram using AUML activity diagram. Ovals represent plans, arcs internal and external events.

- Plan. The plans contained into the capability specification resulting from the detailed design level map to the JACK's plan construct. In JACK a plan is a sequence of instructions the agent follows to try to achieve goals and handle designed events.

As an example, the definition for the *UserInterface* agent, in JACK code, is as follows:

```
public agent UserInterface extends Agent {
    #has capability GetQueryResults;
    #has capability ProvideUserSpecification;
    #has capability GetUserSpecification;
    #has capability PresentQueryResults;
    #handles event InformQueryResults;
    #handles event ResultsSet;
}
```

The capability *PresentQueryResults*, analyzed in detail in the previous section (see Figure 8) is defined as follows:

```
public capability PresentQueryResults
                              extends Capability {
    #handles external event InformQueryResults;
    #posts event ResultsSet ;
    #posts event EmptyResultsSet ;
    #private database QueryResults ();
    #private database ResultsModel ();
    #uses plan EvaluateQueryResults;
    #uses plan PresentEmptyResults;
    #uses plan PresentResults;
}
```

8. CONCLUSIONS

In this paper we have proposed Tropos, a new software engineering methodology which allows us to exploit the advantages and the extra flexibility (if compared with other programming paradigms, for instance OOP) coming from using AOP. The two main intuitions underlying Tropos are the pervasive use, in all phases, of knowledge level specifications, and the idea that one should start from the very early phase of early requirements specification. This allows us to create a continuum where one starts with a set of mentalistic notions (beliefs, goals, plans, ...), always present in (the *why* of) early requirements, and to progressively convert them into the actual mentalistic notions implemented in an agent oriented software. This direct mapping from the early requirements down to the actual implementation allows us to develop software architectures which are "well tuned" with the problems they solve and have, therefore, the extra flexibility needed in the complex applications mentioned in the introduction.

Several open points still remain. The most important are: we should be able to use concepts such as beliefs and events as early as possible in the Tropos methodology; we should be able to better favor the application of adaptation and reuse concepts during all the activities in the development process, as well as to support an iterative process; we should be able to extend the Tropos process also to other important activities of software engineering, such as testing, deployment and maintenance.

9. ACKNOWLEDGMENTS

The knowledge that Paolo Busetta has of JACK has been invaluable. Without him this paper would have been much harder to write. We'd like to thank also Ralph Rönnquist and Manuel Kolp for their helpful comments on the paper.

10. REFERENCES

[1] G. Booch, J. Rambaugh, and J. Jacobson. *The Unified Modeling Language User Guide*. The Addison-Wesley Object Technology Series. Addison-Wesley, 1999.

[2] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. Jack intelligent agents - components for intelligent agents in java. AOS Technical Report tr9901, Jan. 1999. http://www.jackagents.com/pdf/tr9901.pdf.

[3] J. Castro, M. Kolp, and J. Mylopoulos. Developing agent-oriented information systems for the enterprise. In *Proceedings Third International Conference on Enterprise Information Systems*, Stafford UK, July 2000.

[4] L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.

[5] A. Dardenne, A. van Lamsweerde, and S. Fickas. "goal" directed requirements acquisition. *Science of Computer Programming*, (20), 1993.

[6] S. Hayden, C. Carrick, and Q. Yang. Architectural design patterns for multiagent coordination. In *Proceedings of the International Conference on Agent Systems '99*, Seattle, WA, May 1999.

[7] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: a Use-Case Driven Approach*. Addison Wesley, Readings, MA, 1992.

[8] N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2), 2000.

[9] J. Mylopoulos and J. Castro. *Tropos: A Framework for Requirements-Driven Software Development.* Lecture Notes in Computer Science. Springer-Verlag, 2000.

[10] A. Newell. The knowledge level. *Artificial Intelligence*, 18, 1982.

[11] J. Odell and C. Bock. Suggested UML extensions for agents. Technical report, OMG, Dec. 1999. Submitted to the OMG's Analysis and Design Task Force (ADTF) in response to the Request for Information (RFI) entitled "UML2.0 RFI".

[12] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1), 1993.

[13] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995.

[14] M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.

[15] E. Yu. Modeling organizations for information systems requirements engineering. In *Proceedings First IEEE International Symposium on Requirements Engineering*, pages 34–41, San Jose, Jan. 1993. IEEE.

[16] E. Yu. *Modeling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, Department of Computer Science, University of Toronto, 1995.

[17] E. Yu and J. Mylopoulos. From E-R to "A-R" – modeling strategic actor relationships for business process reengineering. In P. Loucopoulos, editor, *Proceedings of 13th Int. Conf. on the Entity-Relationship Approach (ER'94)*, number 881 in Lecture Notes in Computer Science, pages 548–565, Manchester, U.K., Dec. 1994. Springer-Verlag.

[18] E. Yu and J. Mylopoulos. Understanding 'why' in software process modeling, analysis and design. In *Proceedings Sixteenth International Conference on Software Engineering*, Sorrento, Italy, May 1994.

[19] E. Yu and J. Mylopoulos. Using goals, rules, and methods to support reasoning in business process reengineering. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 1(5), Jan. 1996.