Patterns for Agent-Based Information Systems: A Case Study in Transport

Vincent Couturier, Marc-Philippe Huget and David Telisson
LISTIC – Polytech Annecy-Chambéry, Université de Savoie
France

1. Introduction

Designing information systems is a complex task especially when these systems use agents to allow adaptability, cooperation and negotiation, and automatic behaviours. Difficulties arise due to the absence of understandable documentation associated with agent-based methodologies. These methodologies consider concepts defined implicitly and not explicitly requiring from engineers a good understanding of agent theory. This has as consequence an important learning curve for engineers trying to use agents for their information systems. This chapter proposes a collection of agent patterns to reduce time required to develop agent-based information systems.

We propose, in this chapter, to develop software patterns and to reuse them to design complex information systems such as the ones based on agents. According to Alexander (Alexander et al., 1977; Alexander, 1979), a pattern describes a problem, which occurs frequently in an environment as well as a solution that can be adapted for the specific situation. A software pattern (Beck & Cunningham, 1987) follows the same principle and offers a solution to developers when building software in a specific context.

Different categories of software patterns exist as mentioned in Section 2 and here, we present in this chapter, examples of agent patterns for analysis, design and implementation. They are illustrated on our case study in transport: enriched traveller information. These patterns are completed with reuse support patterns that help designing and building such agent-based information systems by guiding them among our collection of patterns.

The chapter is structured as follows. Section 2 presents the concept of pattern. Section 3 describes the categories of patterns dedicated to engineering Agent-based Information Systems (AIS) and the reuse process. Section 4 describes examples of such patterns. Section 5 illustrates these patterns on a transport information system example. Section 6 compares with previous works in literature. Finally, Section 7 concludes the chapter and draws perspectives.

2. The concept of pattern

Alexander introduced the concept of pattern in 1977 for the design and construction of homes and offices (Alexander et al., 1977; Alexander, 1979). This concept was adapted to software engineering and mainly to object-oriented programming by Beck and Cunningham in 1987 (Beck & Cunningham, 1987). These patterns are called *software patterns*.

In Alexander's proposition, a pattern describes a problem, which occurs over and over again in an environment as well as a solution that can be used differently several times. A *software pattern* follows the same principle and can be seen as abstractions used by design or code experts that provide solutions in different phases of software engineering. A pattern can also be considered as a mean to capitalize, preserve and reuse knowledge and know-how.

Patterns can be divided into five categories: **analysis patterns** (Coad, 1996; Fowler, 1997), **architectural patterns** (Buschmann et al., 1996), **design patterns** (Gamma et al., 1995), **idioms**--also known as **implementation patterns**--(Coplien, 1992), and **process patterns** (Ambler, 1998).

Analysis patterns are used to describe solutions related to problems that arise during both the requirement analysis and the conceptual data modeling phases. Among them, we can distinguish generic analysis patterns (Coad, 1992), which represent generic elements that can be reused whatever the application domain is. There exist as well analysis patterns for specific domains (Hay, 1996; Fowler, 1997) called *domain-specific patterns* or *domain patterns*. These patterns (Fowler, 1997) represent conceptual domain structures denoting the model of a business system domain rather than the design of computer programs. Fowler associates to domain patterns *support patterns* that show how domain patterns fit into information system architecture and how conceptual models turn into software. These patterns describe how to use domain patterns and to apply them to a concrete problem.

Architectural and design patterns are both related to the design process. Though, they differ in the level of abstraction where each one is applied. *Architectural patterns* express a fundamental structural organization schema for software systems and can be considered as templates for concrete software architectures (Buschmann et al., 1996). *Design patterns* (Gamma et al., 1995) provide scheme to refine the subsystems or components of a software system and thus are more abstract (and of smaller granularity) than architectural patterns.

Idioms are used at code level and deal with the implementation of particular design issues.

Finally, some patterns, called *process patterns* (Ambler, 1998) describe a collection of general techniques, actions, and/or tasks for developing object-oriented software. Actions or tasks can themselves be software patterns.

We present in next section categories of patterns dedicated to develop Agent-based Information Systems and their reuse process.

3. Categories of patterns dedicated to agent-based information system engineering

3.1 Pattern categories

The first patterns applied for engineering Agent-based Information Systems are *Agent Analysis Patterns*. They define agent structure and design multiagent systems at a high level of abstraction. They can be applied to design agents with or without decision behaviours. Thus, the designer will be able to reuse these patterns to design agents for his/her IS at a high level of abstraction.

Patterns dedicated to architectural representation and design of AIS are *Agent Architectural Patterns* and *Agent Design Patterns*.

The former has to be applied at the beginning of the design process and help defining the IS structural organization. They represent the different architectural styles for agent-based information systems which are means of capturing families of architectures and can be seen as a set of design rules that guide and constrain the development of IS architecture (levels, internal elements, collaborations between elements, etc.). Architectural styles depend on which architecture we choose: Market-based one, Subcontract-based one or Peer-to-Peer-based one.

Agent Design Patterns describe technical elements required to develop agent-based Information Systems. Analysis and conceptual models obtained by applying Agent Analysis Patterns are refined with behaviour, collaboration and software entities. Thus, the IS design model is obtained by adapting software elements specified in the design patterns solutions.

Finally, we have specified two kinds of support patterns: *Model Transformation Patterns* and *Reuse Support Patterns*.

Model Transformation Patterns help developers to build applications from design patterns and can be applied at the end of the design phase. They specify transformation rules to map design models to models specific to agent development frameworks such as JADE (Bellifemine et al., 2007) or Madkit (Gutknecht & Ferber, 2000).

Reuse Support Patterns (RSP) are process patterns, which help developers navigating into a collection of patterns and reusing them. They describe, by using activity diagrams, a sequence of patterns to apply to resolve a problem. There exists RSP for every category of patterns (analysis, architectural, design and model transformation).

The different patterns described here regarding the development cycle of an agent-based information system are shown on Figure 1.

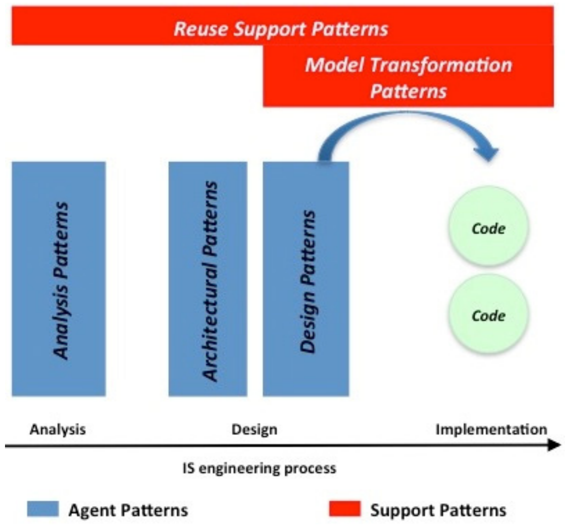


Fig. 1. The use of the different proposed patterns in the development cycle of an agent-based IS.

The description of our software patterns is composed of four parts:

- The *Interface* part contains the following fields: *Name* and *Classification* (used to categorize the pattern: Analysis pattern, Design pattern, etc.), *Context* (defines the conditions under which the pattern could be applied), *Rationale* (gives which problems this pattern addresses) and *Applicability* (gives the scope of this pattern: Information Systems in our case).
- The *Solution* part when proposed as a model-based solution is composed of the following fields: *Model* (an agent pattern presents a solution as a UML class diagram and/or a UML sequence diagram), *Participants* (explanation of the different elements defined on the diagram) and *Consequences* (advantages and drawbacks of this pattern to help developers deciding whether this pattern is the correct one). When the *Solution* part is proposed as a process-based solution (for instance for Reuse Support patterns), the *Solution* part is composed of a unique field entitled *Process* defined as a UML activity diagram.
- The *Example* part describes one or more illustrations on how to use this pattern.
- The *Relationship* part is composed of the following fields: *Uses* (describes the relationship: "the pattern X is using the pattern Y in its solution"), *Requires* ("the pattern X requires the pattern Y to be applied before"), *Refines* ("the pattern X refines the pattern Y if and only if the pattern Y solves the same issues than the pattern X") and *Looks like* ("the pattern X is a variant of the pattern Y").

Note: The different patterns presented here are reduced versions. We only describe the most important parts and fields required to understand what a pattern means. As a consequence, we remove the Example part, which is presented in Section 5.

3.2 Pattern reuse

The reuse of patterns dedicated to develop Agent-based IS consists in applying them during analysis, design and implementation phases.

First, developers analyze context and problem and should have to answer questions to decide which patterns have to be applied and in which order. This activity can be favoured by using *Reuse Support Patterns* which represent sequences of patterns that can be applied to develop Agent-based IS (See Table 1 for an example of RSP suited for navigating in our Analysis Pattern collection). They help to navigate into the pattern collection and to reuse them. Thus, developers adapt analysis, architectural or design pattern solution elements (instantiation) to represent the system they want to develop. Finally, the third activity aims at using Model Transformation Patterns to generate skeleton application from pattern instances.

It is worth mentioning that, here, reuse is realised by adaptation. Designers do not directly reuse the patterns but adapt the different solutions (instantiation) to their specific applications by modifying the level of abstraction given by the patterns. Moreover, as briefly depicted in the "Service Integration" RSP below, designers should have to answer questions to decide which patterns have to be applied. Another example is given in Table 1 where the "Restrict access to resources" pattern is used if and only if some policies are in use on resources.

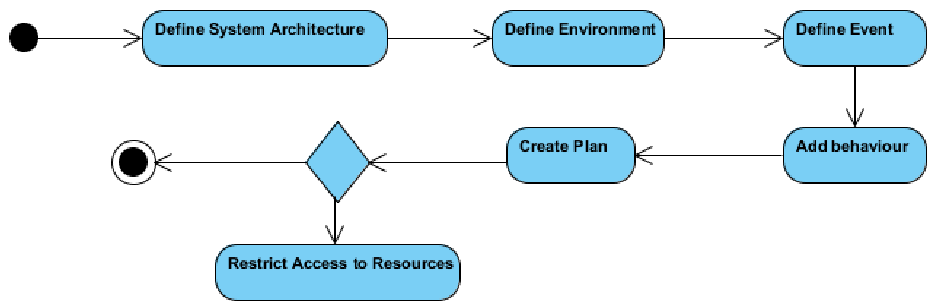
Interface
Name
Base Agent Design
Classification
Reuse Support Pattern
Rationale
This pattern presents Agent Analysis Patterns that can be applied to develop a base agent. Here, a base agent is an agent that plays roles within organisations, lives in an environment and reacts to events in the environment, and optionally acts on resources (perception and action) if it has the associated permission.
Applicability
Agent Analysis Pattern Collection ^ Base Agent
Solution
Process
 Define System Architecture Define Environment Define Event Add behaviour Create Plan Restrict Access to Resources Designers first have to apply the pattern "Define system architecture", then the patterns "Define environment", "Define event", "Add behaviour" and "Create plan". After applying the "Create plan" pattern, it is possible either to terminate the process or to continue with the "Restrict access to resources" pattern depending on the necessity to have policies on resource access (a resource is for instance digital documents such as contracts, proposals, enterprise database, etc.). This decision is fuelled by considering the place of agents in the environment: do all agents access resources? Do some resources need to be kept private? Based on the answers, designers may decide to apply the "Restrict access to resources" pattern. *Note : only the "Define System Architecture" Analysis Pattern in this RSP is presented in Section 4.*
Relationships
Uses
"Define System Architecture", "Define Environment", "Define Event", "Add Behaviour", "Create Plan", "Restrict Access to Resources" Agent Analysis Patterns.

Table 1. Reuse Support Pattern "Base Agent Design"

Several other reuse support patterns (RSP) are proposed in our approach to address specific needs. Amongst them, we can quote the "Service Integration" RSP. The "Service Integration" RSP helps designers integrating the notion of services and service-oriented architectures within the information system. In this particular RSP, the process is not limited to a set of patterns to apply in a given order but obliges designers to think about the overall enterprise Information Systems:

- Do we need to agentify the services from the Information System?
- Do we consider agents as a wrapper of services?
- Do we need to present agent behaviours as services to Information Systems?
- Do we need to provide access to external Information Systems and partners then requiring interoperability and the definition of ontologies?

Based on designer's answers, a specific process will appear from the complete activity diagram in the "Service Integration" RSP.

Moreover, we have developed a toolkit, which is based on our software patterns. It takes as input a Reuse Support Pattern, guides the developer--by asking questions--through the different patterns to be used, and finally generates code skeleton. The process is then not fully automated due to interactions with developer. Thus, s/he can complete and refine the generated code and run his/her agents on a target platform.

We present, in next section, Agent Patterns we designed to develop Agent-based IS.

4. Patterns for engineering agent-based information systems

4.1 Patterns for the analysis phase

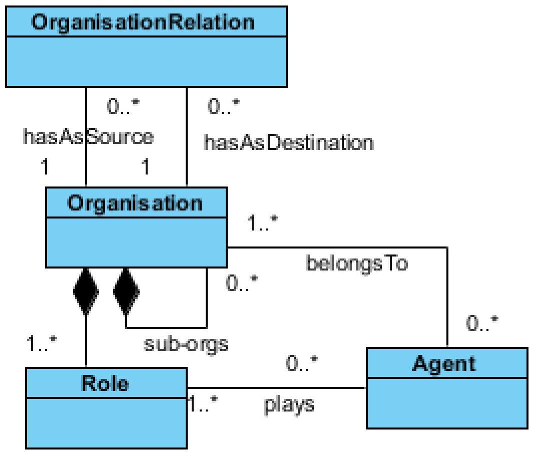
In following sections, we present patterns for the analysis phase of information systems engineering, which are *Agent Analysis Patterns*.

4.1.1 Agent analysis patterns

The analysis patterns described below are generic ones used for building agent-based information systems at a high level of abstraction. Due to space restriction, we only present two among twelve analysis patterns for building agents used in Information Systems.

4.1.1.1 Agent analysis pattern "define system architecture"

Interface
Name
Define System Architecture
Classification
Agent Analysis Pattern
Rationale
The aim of this pattern is to define the organisation and sub-organisations, their relations, and the roles played by agents in these organisations.
Applicability
Designing agents ^ Information systems
Solution
Model



Participants

This pattern describes the overall structure of the multiagent system underlying the IS. A multiagent system is here an *Organisation* possibly composed of sub-organisations. Each (sub-) organisation is related to other (sub-) organisations by some *OrganisationRelation*. *Agent*s play *Role* in these organisations.

The *Agent* concept corresponds to the notion of agent defined in the agent theory (Wooldridge, 2002). An agent is an autonomous and active entity, which asynchronously interacts with other agents and cooperates with others so as to solve a global problem. An agent is seen as an aggregation of *Role*. An agent is uniquely identified within the system.

The *Role* concept describes a role that the agent will play. It defines a catalogue of behaviours played within the system.

An association entitled *plays* links the *Agent* concept to the *Role* concept. This association has the following cardinalities: an *Agent* may have 1 or more roles, and a *Role* may be played by an agent.

The *Organisation* concept defines the organisational structure used in the system. There could be a flat organisation or an organisation composed of sub-organisations.

Table 2. Agent Analysis Pattern "Define System Architecture"

4.1.1.2 Agent analysis pattern "define protocol"

Interface
Name
Define Protocol
Classification
Agent Analysis Pattern
Context

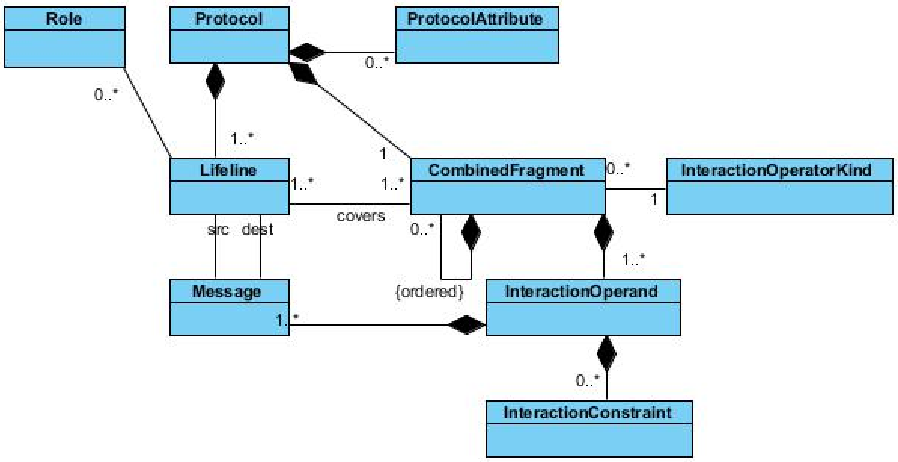
This pattern requires applying the "Define Communication between Roles" pattern before.
Rationale
This pattern defines the protocol with the messages between roles.
Applicability
Designing agents ^ Information systems
Solution
Model
 Role Protocol ProtocolAttribute 0..* 0..* 1..* 1 Lifeline 1..* 1..* covers CombinedFragment 0..* 1 InteractionOperatorKind src dest 0..* {ordered} 1..* Message 1..* InteractionOperand 0..* InteractionConstraint
Participants
The different roles present in the *Protocol* are denoted by *Lifeline*. *Lifeline* specifies when a *Role* enters the conversation and when it leaves it. *Message* are exchanged between *Lifeline* and are gathered within *InteractionOperand*. These *InteractionOperand* correspond to sequence of messages. Some *InteractionConstraint* may alter how *InteractionOperand* can be used. Finally, *InteractionOperand* are gathered within *CombinedFragment* and the semantics of these fragments is given by *InteractionOperatorKind*. These *InteractionOperatorKind* are *alt* (one *InteractionOperand* is selected based on *InteractionConstraint), opt* (an *InteractionOperand* is applied if the corresponding *InteractionConstraint* are satisfied else nothing is done) and *loop* (a *CombinedFragment* is applied over and over again as long as the *InteractionConstraint* are satisfied). Some *ProtocolAttribute* may be defined for the *Protocol*, they correspond to parameters for the protocol.
Relationship
Requires
Agent Analysis Pattern "Define Communication between Roles".

Table 3. Agent analysis pattern "Define Protocol"

4.2 Patterns for the design phase

In this section, we present Agent Architectural and Design Patterns for the architectural and detailed design of AIS. These patterns have to be applied after analysis patterns described above.

4.2.1 Agent architectural patterns

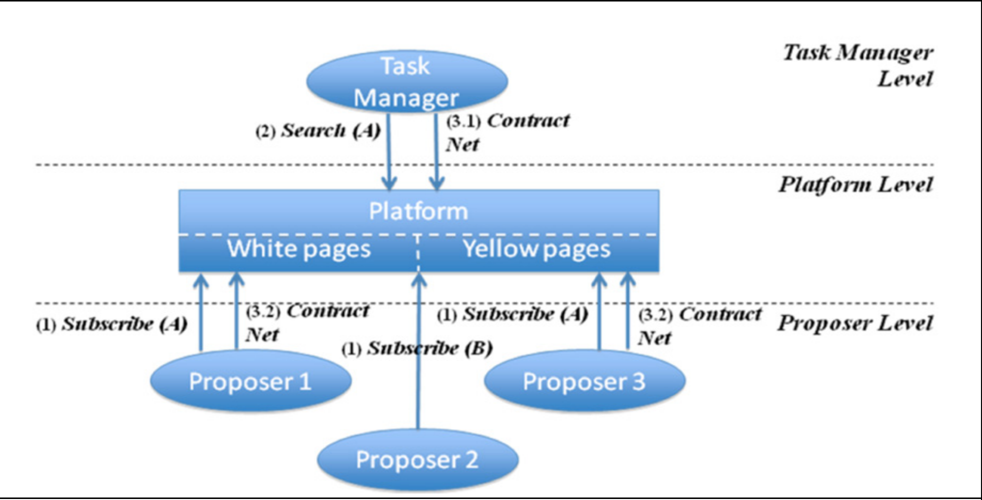
We develop three architectural patterns related to the different architectures an AIS could have:

- Pattern "Market-based AIS": a marketplace is defined with this pattern. A marketplace is composed of several proposers and several task managers. Task managers try to find the best proposal for a service. Two approaches are possible to retrieve this best proposal: (1) A descending price auction or (2) A call for proposals.
- Pattern "Subcontract-based AIS": An AIS with subcontracts is a restricted version of the previous pattern "Market-based AIS". In this particular case, there is only one task manager and several proposers. The best proposal is found after a call for proposals.
- Pattern "Peer-to-Peer-based AIS": previous patterns impose to use a central server so as to store the address of the different task managers and proposers. This approach does not resist to the scalability problem and the bottleneck is located on querying the central server to retrieve the different task managers and proposers. In this pattern here, there is no central server and the different task managers and proposers know each other via social networks. This kind of architecture copes with the scalability problem.

Below, we only present the pattern "Subcontract-based Agent-based Information System".

Note: The different design and model transformation patterns described below are those required for building a Subcontract-based AIS.

Interface
Name
Subcontract-based Agent-based Information System
Classification
Agent Architectural Pattern
Rationale
This pattern gives the structure of a subcontract-based information system with a unique *Task Manager* and several *Proposers*.
Applicability
Designing agents ^ Information systems
Solution
Model



Participants

This kind of AIS architecture considers three layers: the *Task Manager* layer, the *Platform* layer and the *Proposer* layer.

The Task Manager layer contains one unique *Task Manager* playing the role of task manager in AIS. It is the one that requests services from Proposer.

The Proposer layer contains one or more *Proposer* playing the role of proposers who provide services.

The *Platform* layer contains two services proposed to the different task manager and proposers, that is the white pages and the yellow pages. White pages give the address of the different entities within the system and yellow pages return the service proposed by proposers.

Collaborations and communications within the architecture:

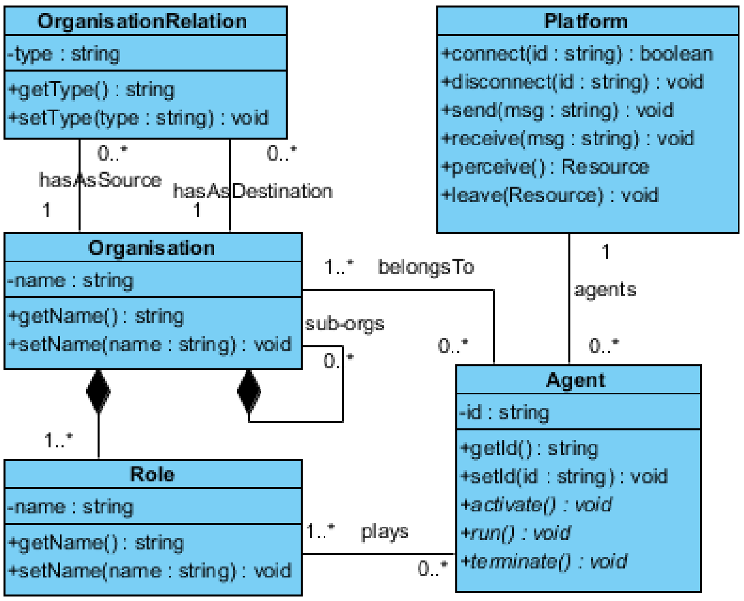
1. Proposers register their services within the yellow pages with the performative *subscribe*.
2. A task manager looks for proposers providing a specific service (here the service A) within the yellow pages with the performative *search*. It then retrieves their address within the white pages so as to contact them.
3. A Contract Net protocol (Davis & Smith, 1983) is then used between proposers and task manager so as to find the best proposal for a specific requested service (here the service A).

Table 4. Agent Architectural Pattern "Subcontract-based Agent-based Information System".

4.2.2 Agent design patterns

The following patterns describe the different concepts needed for designing an Agent-based IS. We only present here two examples of such patterns.

4.2.2.1 Agent design pattern "platform-based system architecture"

Interface
Name
Platform-based System Architecture
Classification
Agent Design Pattern
Rationale
This pattern describes the overall structure of the system taking count of the platform.
Applicability
Designing agents ^ Information systems
Solution
Model
 OrganisationRelation -type : string +getType() : string +setType(type : string) : void Platform +connect(id : string) : boolean +disconnect(id : string) : void +send(msg : string) : void +receive(msg : string) : void +perceive() : Resource +leave(Resource) : void 0..* 0..* hasAsSource hasAsDestination 1 1 Organisation -name : string +getName() : string +setName(name : string) : void 1..* belongsTo 1 agents sub-orgs 0..* 0..* 0..* Agent -id : string +getId() : string +setId(id : string) : void +activate() : void +run() : void +terminate() : void 1..* Role -name : string +getName() : string +setName(name : string) : void 1..* plays 0..*
Participants
This pattern describes the overall structure of the multiagent system underlying the IS from the design point of view. A multiagent system is here an *Organisation* possibly composed of *sub-organisations*. Each (sub-) organization is related to other (sub-) organizations by some *OrganisationRelation*. Agents play *Role* in these organizations. The *Agent* concept corresponds to the notion of agent defined in the agent theory (Wooldridge, 2002). An agent is an autonomous and active entity, which asynchronously interacts with other agents and cooperates with others so as to solve a global problem. An agent is seen as an aggregation of *Role*. An *Agent* is defined as an abstract class from object

theory since three operations mentioned below are abstract. An agent is uniquely identified in the system via the attribute *id*. Getter and setter operations are defined for the attribute *id*. Three other operations are defined abstract and have to be instantiated in the instance of this *Agent*. *Activate()* contains behaviours for initialising the agent. *Run()* is executed every time it is the turn of the agent to be executed. Finally, *terminate()* describes behaviours executed when ending the agent execution. The *Role* concept describes a role that the agent will play. It defines a catalogue of behaviours played within the system. The *Role* concept defines an attribute *name* and its corresponding getter and setter operations. An association entitled *plays* links the *Agent* concept to the *Role* concept. This association has the following cardinalities: an *Agent* may have 1 or more roles, and a *Role* may be played by an agent. The *Organisation* concept defines the organizational structure used in the system. There could be a flat organization or an organization composed of sub-organisations. An attribute *name* and its corresponding getter and setter operations are associated to the *Organisation* concept. An association *belongsTo* links the *Organisation* concept to the *Agent* concept. It expresses the fact that an agent may belong to several organizations and an organization has zero or more agents whatever their roles are. The *OrganisationRelation* concept describes the relation between two organizations. Finally, the *Platform* concept defines the platform and the different services provided by this one. These services are present by the operations available on the Platform concept: connection to the platform, disconnection from the platform, send a message, receive a message saved on the platform, perceive for sensing traces in the environment, and leave for adding traces in the environment.
Relationships
Requires
Agent Analysis Pattern "Define System Architecture".

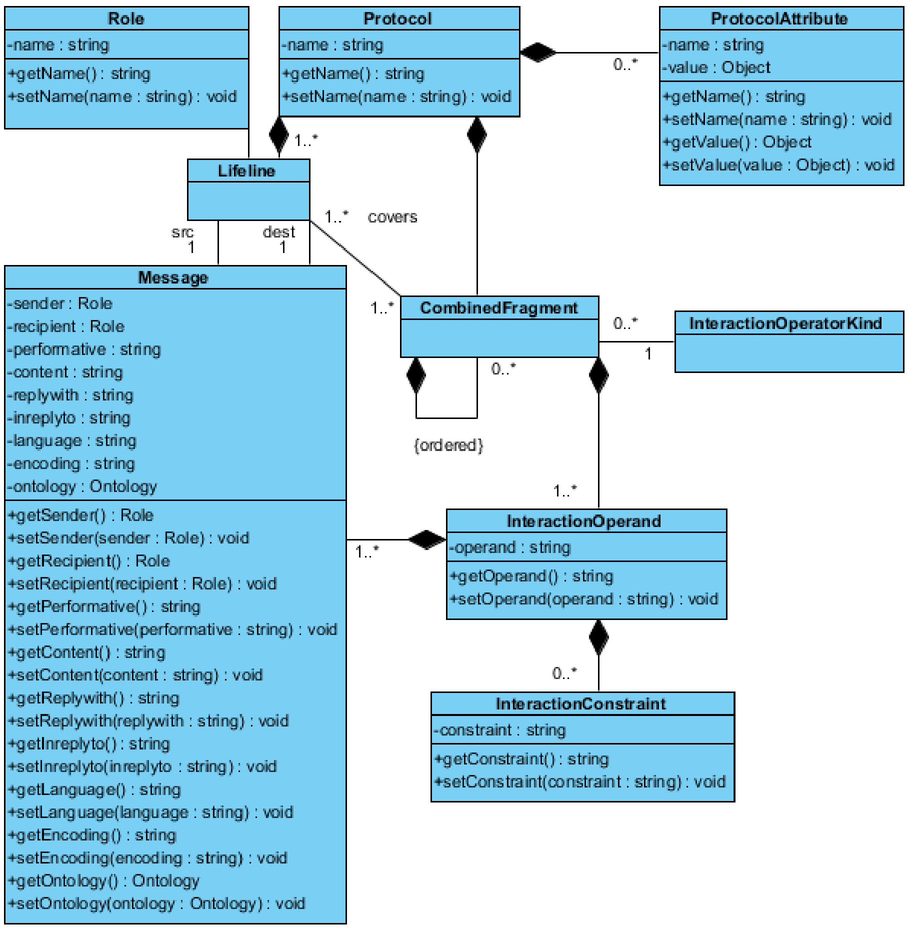
Table 5. Agent Design Pattern "Platform-based System Architecture"

4.2.2.2 Agent design pattern "FIPA-based interaction with protocol"

Interface
Name
FIPA-based Interaction with Protocol
Classification
Agent Design Pattern
Rationale
This design pattern describes the notion of cognitive interaction in terms of protocols within roles. This interaction is FIPA-compliant.
Applicability
Designing agents ^ Information systems

Solution

Model



Participants

Interactions between roles are either based on pheromones left in the environment (we speak about reactive interactions) or based on communicative acts as humans do (we speak then about cognitive interactions). In this design pattern, we consider cognitive interactions through protocols. Protocols help directing the conversations between roles since only messages from the protocol are granted when agents interact with this protocol.

This design pattern is FIPA-compliant (FIPA, 2002) and is based on the UML 2.x sequence diagram specifications. We just remove some classes that are nonsense for agents.

The following concepts are present in this design pattern:

The *Role* concept describes a role that the agent will play. It defines a catalogue of behaviours played within the system. The *Role* concept defines an attribute *name* and its corresponding getter and setter operations.

The *Protocol* concept defines a protocol. It contains all the sequences of messages allowed for this protocol. The *Protocol* concept defines an attribute *name* and its corresponding getter and setter operations.

A protocol may contain some *ProtocolAttribute.* These attributes correspond to local attributes required during the execution of the protocol. It could be for instance the set of recipients of a specific message. The *ProtocolAttribute* concept defines an attribute as a *name* and a *value*. The corresponding getter and setter operations are defined too.

The different roles are denoted by the *Lifeline* concept in the protocol.

Since this protocol definition is based on UML 2.x sequence diagram specification, a protocol is decomposed into *CombinedFragment*. Each *CombinedFragment* has an associated *InteractionOperatorKind* from the following list: *alt*, *opt* and *loop*. *Alt* denotes an alternative between several *InteractionOperand.* One and only one alternative will be chosen. *Opt* denotes an option on an *InteractionOperand*. This *InteractionOperand* is executed if and only if the conditions-represented by *InteractionConstraint*- are satisfied. Finally, *loop* denotes the execution of a set of messages as long as the conditions are satisfied.

The *Message* concept is the concept following the FIPA definition. It contains a set of attributes and their getter and setter operations. *Sender, recipient, performative* and *content* denote from whom the message is sent to whom. A message is composed of two parts: a *performative* depicting the verb of the communicative act (*inform*, *request*, etc.) and a *content* on which this *performative* is applied. The other attributes are for administrative duties: *replywith* and *inreplyto* correspond to identifier respectively for the sender and the recipient. *Language* denotes the language in which the content parameter is expressed. *Ontology* defines which ontology is used for this message. Finally, *encoding* denotes the specific encoding of the content language expression.

Relationships
Requires
Agent Analysis Pattern “Define Protocol”

Table 6. Agent Design Pattern “FIPA-based Interaction with Protocol”.

4.3 Patterns for the implementation phase: Model transformation patterns

We define several *Model Transformation Patterns* for developing AIS for different architectures (subcontract-based architectures, market-based ones and peer-to-peer-based ones) and for different execution platforms (JADE and Madkit). We only present here in Table 7, a short version---without method transformations---of a Model Transformation Pattern for Madkit implementation of a subcontract-based AIS.

Interface
Name
Madkit Implementation of a subcontract-based Agent-based Information System
Classification
Model Transformation Pattern

Rationale

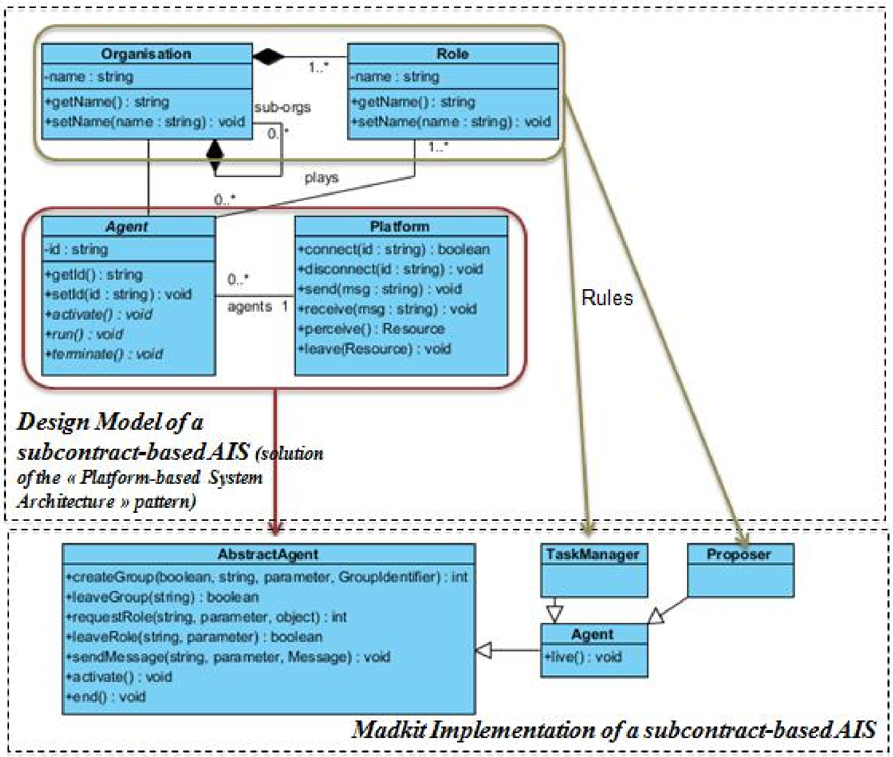
This pattern performs the model transformation from a design model of a subcontract-based Agent-based Information System to the Madkit platform.

Applicability

Implementing agents ^ Subcontract-based Information Systems

Solution

Model


Organisation
-name : string
+getName() : string
+setName(name : string) : void
Role
-name : string
+getName() : string
+setName(name : string) : void
1..*
sub-orgs
0..*
1..*
plays
0..*
Agent
-id : string
+getId() : string
+setId(id : string) : void
+activate() : void
+run() : void
+terminate() : void
Platform
+connect(id : string) : boolean
+disconnect(id : string) : void
+send(msg : string) : void
+receive(msg : string) : void
+perceive() : Resource
+leave(Resource) : void
0..*
agents 1
Rules
Design Model of a subcontract-based AIS (solution of the « Platform-based System Architecture » pattern)
AbstractAgent
+createGroup(boolean, string, parameter, GroupIdentifier) : int
+leaveGroup(string) : boolean
+requestRole(string, parameter, object) : int
+leaveRole(string, parameter) : boolean
+sendMessage(string, parameter, Message) : void
+activate() : void
+end() : void
TaskManager
Proposer
Agent
+live() : void
Madkit Implementation of a subcontract-based AIS

Rule 1: If the instance of Agent is linked to Organisation org1 then insert in activate() code of the instance the following line: *int g = createGroup(true, "community", "org1", null, null);*
Rule 2: If the instance of Agent is linked to Role role1 then insert in activate() code of the instance the following line: *int r = requestRole("community", "org1", "role1",null);*

Note: In this pattern and due to space restriction, we do not consider the OrganisationRelation concept since it is not mandatory for a subcontract-based AIS.

Participants

This pattern ensures the transformation from a design model of an AIS to a set of classes for the Madkit platform. Agents on the Madkit platform are defined as a specialization of the *AbstractAgent* class provided by the Madkit platform. The *AbstractAgent* class from the Madkit platform provides the different methods required for the Agent lifecycle (creation, invocation, execution and deletion). These methods correspond to the ones proposed in the *Agent* concept. The set of attributes and methods from the *Role* concept is added to the

Task manager and *Proposer* classes. The *Task manager* and *Proposer* are the two unique roles in a subcontract-based AIS according to the "Subcontract-based Agent-based Information System" architectural pattern (see Table 4). Two rules are added for model transformation. *Rule 1* expresses that organizations are created within agents in the *activate()* operation. Agents are responsible to create the organizations. *Rule 2* specifies that roles agents have, are taken within the *activate()* operation of the corresponding agent.
Relationships
Uses
Agent Design Pattern "Platform-based System Architecture".

Table 7. Model Transformation Pattern "Madkit Implementation of a Subcontract-based Agent-based Information System"

5. A case study

The objectives of our case study are to provide enriched traveller information. This enriched traveller information is in fact the collaboration of two different tools: (1) A route planner considering usual travel means such as buses and undergrounds but also taxis, personal vehicles, rent bicycles and walking, and (2) An adorned travel with points of interests related to traveller preferences (cultural interests, food preferences, etc.). The process of proposing a route to traveller is as followed. After entering origin and destination, the information system composed of all the different operators (bus, underground, taxi, and rent bicycle) cooperate to find the best route proposals based on the preferences (cost, duration, number of connections, etc.) and requirements (no stairs, disabled access, ease of use, etc.) of the traveller. Then, the system prunes all the proposed routes based on traveller requirements. Finally, points of interest providers adorn the routes with contextual information such as restaurants matching the traveller's food preferences if the route is during meal hours, shops or monuments, etc.

This information system exhibits some specific features that are compatible with an agent-based system. First of all, route planning is not realised according a client/server approach. Every operator is responsible of its data and is the only one to know how to deal with scheduled and/or unexpected events (delays, traffic jam, disruptions, etc.). As mentioned above, operators collaborate to find routes from origin to destination.

A second reason is the openness of the system. The list of operators (especially taxis) and points of interest providers is subject to evolve, especially during execution. The system should be able to take account of appearing and disappearing providers.

Finally, a third reason is the necessity for the information system to present some adaptability mechanisms. A route may change due to unexpected events or after traveller requests. The system should be able to modify the proposals during execution.

For all these reasons, an agent-based system is well-adapted since adaptability, openness, and context-aware are part of the intrinsic features of agents. We invite the reader to consult (Wooldridge, 2002) for details on agent-based systems and their characteristics.

We focus in this chapter on how designing and building the transport information system responsible to provide enriched traveller information.

Figures 2 and 3 give the instantiation for our case study of the two analysis patterns presented in Section 4.1.

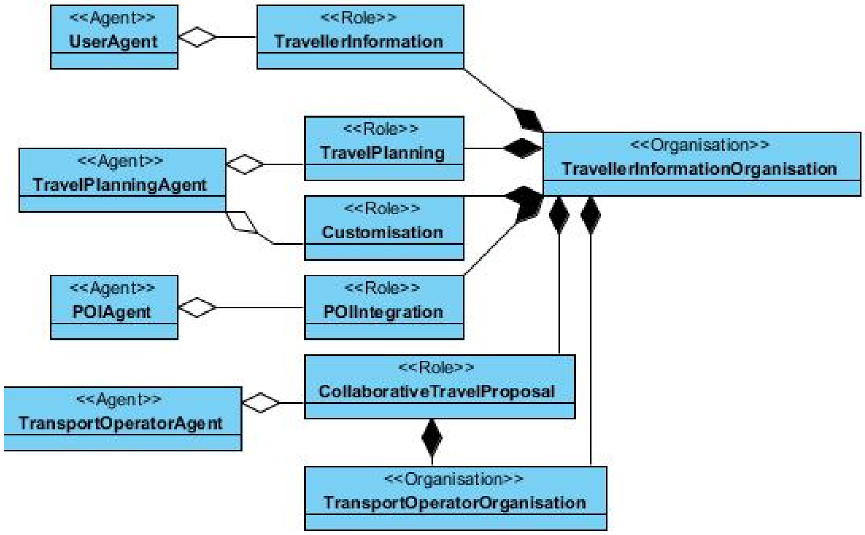


Fig. 2. Instantiation of the “Define System Architecture” Agent Analysis Pattern

Figure 2 describes the complete system architecture with one organisation *Traveller Information Organisation*, one sub-organisation *Transport Operator Organisation*, five roles *Traveller Information, Travel Planning, Customisation, POI Integration* and *Collaborative Travel Proposal*, and four agents *User Agent, Travel Planning Agent, POI Agent,* and *Transport Operator Agent*.

Each agent *Transport Operator Agent* represents a means to travel inside a city: underground if available, bus, taxi, rent bicycle, personal vehicle or by foot. These agents play the role *Collaborative Travel Proposal* since they try to collaborate so as to complete the travel from origin to destination. All these agents are part of the *Transport Operator Organisation.*

The *Transport Operator Organisation* is part of the *Traveller Information Organisation, which* carries information to travellers.

User Agent represents the traveller requesting the system. *Travel Planning Agent* is responsible to ask for a list of journeys to *Transport Operator Agent. Travel Planning Agent* has two roles: (1) *Travel Planning* to request journeys and (2) *Customisation* to prune the journeys based on user preferences and requirements. This role sends journeys back to the *User Agent.*

POI Agent represents point of interests within the city. These agents intervene when a journey is completed and add some points of interest based on user preferences. Points of interest might be restaurants, monuments, shops to name a few.

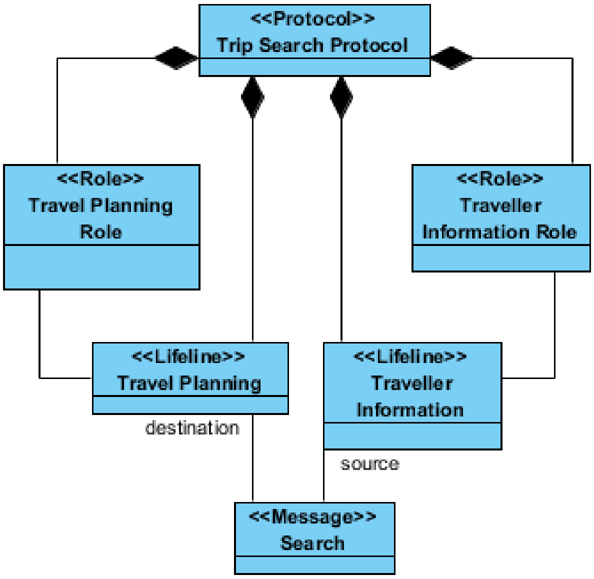


Fig. 3. Instantiation of the "Define Protocol" Agent Analysis Pattern

Figure 3 presents the protocol (instantiation of the "Define Protocol" analysis pattern) that initiates the search for a trip between the *Traveller Information* role acting for the user and the *Travel Planning*. The message sent is the *Search* message.

Figures 4 and 5 give the instantiation for our case study of the two design patterns presented in Section 4.2. The developer of a transport information system has to apply the agent analysis and architectural--not presented here--patterns before.

Figure 4 corresponds to Figure 2 after refining analysis model, i.e. inserting some attributes and operations. All the operations (except for the Platform concept) are getter and setter operations. We define below the different attributes for the concepts on this pattern:

UserAgent has attributes corresponding to the travel: there are an origin, a destination, a maximum amount s/he would like to pay and a maximum duration for the travel. *Preferences* and *Requirements* contain a *description* attribute describing the preferences or the requirements the user has.

TravelPlanningAgent has three attributes: *queries* containing the different user queries the enriched travel planning system has to satisfy, *plannedTravels* containing the raw travel planning answering user queries and finally *enrichedTravels* contain the list of enriched travels with points of interest to send to users.

POIAgent has a unique attribute *description* describing the point of interest (position, description, etc.) for inclusion in travel plans.

TransportOperatorAgent has an attribute *TransportOperatorDB* corresponding to the database of all the details about the journeys proposed by the operator. When the operator is a rent bicycle one, the database contains the different locations of rent and where bicycles are.

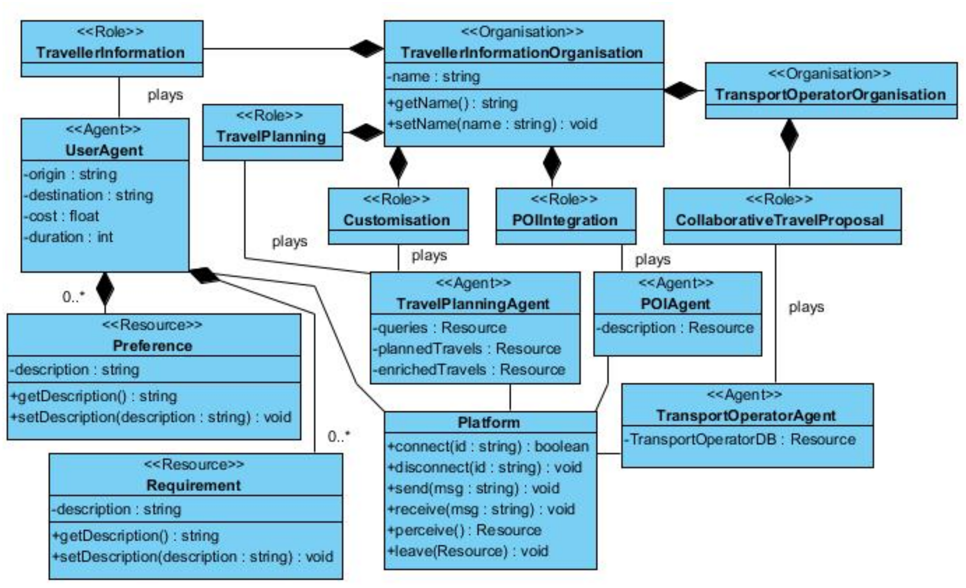


Fig. 4. Instantiation of the "Platform-based System Architecture" Agent Design Pattern

Figure 5 presents the sequence diagram corresponding to the situation where a user asks for a travel from an origin to a destination. This figure is the instantiation of the "FIPA-based Interaction with Protocol" Agent Design Pattern. His/her *UserAgent* leaves the query in the environment. This insertion generates an event, a *TravelPlanningAgent* can perceive. If this *UserAgent* is a newcomer in the system, the *TravelPlanningAgent* asks the *UserAgent* about its user's preferences and requirements.

TravelPlanningAgent then leaves in the environment a travel from origin to destination but without schedule. This empty travel is perceived by *TransportOperatorAgent* that tries to complete it. Every *TransportOperatorAgent* tries to update this travel or to propose an alternative. When this travel was considered by all *TransportOperatorAgent*, it turns into a planned travel. The *TravelPlanningAgent* perceives it and turns it into an enriched travel to let *POIAgent* to perceive it.

POIAgent tries to update it with points of interest and leaves the enriched travel plans in the environment. Finally, *Customisation* prunes the different proposals based on user's preferences and requirements and informs *UserAgent* of the best proposals.

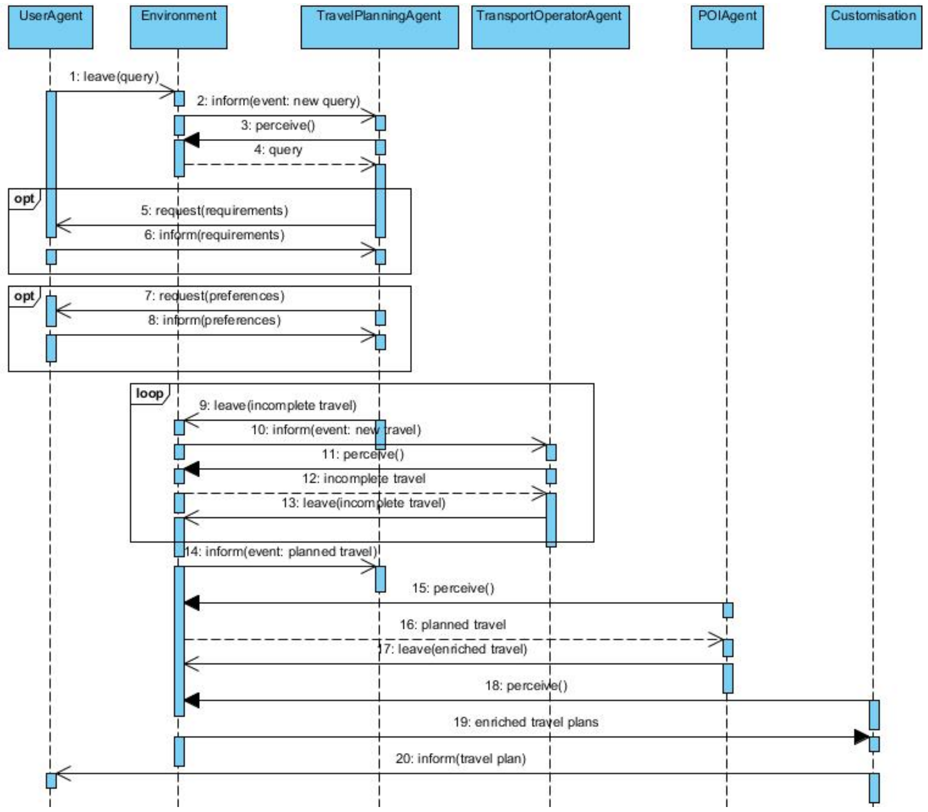


Fig. 5. Instantiation of the “FIPA-based Interaction with Protocol” Agent Design Pattern

6. Related work

We can find two kinds of patterns related to Information Systems and agent-based systems engineering in literature:

- Agent-based patterns
- Patterns for Information Systems engineering

Patterns for Information Systems engineering (for instance, patterns for cooperative IS (Couturier, 2004) (Saidane, 2005), e-bidding applications (Jureta et al., 2005; Couturier et al., 2010), distributed IS (Renzel & Keller, 1997), enterprise application architecture (Fowler, 2002), etc.) are generally domain-dependent and/or do not deal with advanced information systems requiring adaptability, cooperation or negotiation such as agent-based ones.

On the other hand, the concepts of agent technology, which include, among others, autonomy, proactivity, reactivity, social behaviours, adaptability and agents, differ from those of traditional software development paradigms. The various concepts and the relationships among them generate different agent-oriented software engineering problems for which agent-oriented patterns have been written.

According to Oluyomi et al. (Oluyomi et al., 2007), numerous efforts have been made by agent software practitioners to document agent-oriented software engineering experiences as patterns and they establish a listing of ninety-seven agent-oriented patterns gathered from literature.

Oluyomi et al. classify agent-oriented patterns based on the definition of the software tasks/concepts of agent technology and the stages of development.

According to this first point of view, numerous works such as (Kendall et al., 1998) (Aridor & Lange, 1998) feature agent-oriented concepts as object-oriented ones and adapt existing object-oriented patterns to their needs. This is not suited for agent-based system engineering due to the differences between agent technology concepts and object ones and their implementation languages.

According to the second point of view, existing agent patterns are not designed to capture all the different phases and processes of agent-oriented software engineering. Indeed, proposals ((Hung and Pasquale, 1999), (Tahara et al., 1999), (Sauvage, 2003), (Schelfthout et al., 2002) to name a few) focus either only on the implementation phase of development or only on some aspects of the design phase but scarcely to analysis. Other works are based on implementation of only a particular application of agent technology, for example, mobile agents (Aridor and Lange, 1998), or reactive or cognitive ones (Meira et al., 2001) for instance. It is worth mentioning that it is difficult to reuse these proposals to realise a complete agent-based information system: either the proposals only deal with a specific agent type, or the collection of patterns is partial and not homogeneous enough.

We add a third classification: proposals specifying patterns with or without providing tools or a methodology to help reusing these patterns. Some patterns underlie methodologies such as Tropos (Do et al., 2003) or PASSI (Cossentino et al., 2004). These methodologies aim at guiding developers when using patterns to develop agent-based systems. However, Tropos only proposes patterns for detailed design. These patterns focus on social and intentional aspects frequently present in agent-based systems. Patterns in PASSI methodology deal with detailed design and implementation. One hurdle in PASSI is this is not trivial selecting the appropriate patterns especially for new agent developers. Most of works do not propose a methodology or a guide to reuse patterns.

Thus, it becomes difficult for a developer to reuse these proposals to design and implement an agent-based information system:

- Proposed patterns are too generic and do not match with information systems issues.
- It is very difficult for non-agent software practitioners to easy understand the different aspects of agent based systems development.
- Users do not have adequate criteria to search for suitable patterns to solve their problems (lack of methodology).
- All stages of software development are not covered and combination of agent-oriented patterns written by different authors, into a well-defined pattern collection is nearly impracticable.

Our proposal, which covers all the phases of agent-based information systems engineering, is suitable for each kind of agent (agents with or without decision behaviours) and addresses information systems issues such as business rules, legacy systems, services and enterprise resources, for instance. We also propose a methodology based on our Reuse Support Patterns.

7. Conclusion

This chapter describes our work about specifying and reusing patterns so as to engineer Agent-based Information Systems. The different patterns presented here represent the building blocks which, after adaptation, can be used to develop analysis and design models for a new IS, define the architecture and ease implementation. The patterns cover all the phases of IS engineering and a methodology, based on our Reuse Support Patterns, is provided to favour their reuse. We have also developed a toolkit so as to ease engineering Information System applications and specifically, intelligent transport systems. This toolkit is based on our software patterns. It takes as input a Reuse Support Pattern, guides the developer through the different patterns to be used, and finally generates code skeleton.

Our approach and the different patterns are experimentally validated on a specific IS for transportation. Reusing the patterns help eliciting the business entities (analysis model), architecting the system (the architecture is a subcontract-based one since there is a unique task manager and several proposers), defining the design model and generating code skeleton for the Madkit platform.

Future work aims at reusing the different patterns presented here so as to develop other Enterprise Information Systems (schedule management for instance).

8. References

Alexander, C.; Shikawa, S.; Silverstein, M.; Jacobson, M.; Fiksdahl-King, I., & Angel, S. (1977). *A pattern language: towns, buildings, construction*, Oxford University Press, ISBN 0195019199, New York

Alexander, C. (1979). *The timeless way of building*, Oxford University Press, ISBN 0195022483, New York

Ambler S.W. (1998). *Process patterns: building large scale systems using object technology*, ISBN 0521645689, Cambridge University Press

Aridor, Y. & Lange, D.B. (1998). Agent Design Patterns: Elements of Agent Application Design, *Proceedings of the second international conference on autonomous agents*, ISBN 0897919831

Beck, K. & Cunningham, W. (1987). *Using pattern languages for object-oriented programs,* technical report CR-87-43, Computer Research Laboratory, Tektronix

Bellifemine, F.L.; Caire, G. & Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE*, Wiley, ISBN 0470057475, New York

Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P. & Stal, M. (1996). *Pattern-oriented software architecture: A system of patterns*, John Wiley & Son, ISBN 0471958697, Chichester, UK

Coad, P. (1992). Object-Oriented Patterns. *Communications of the ACM*, 35(9), 152-159.

Coad, P. (1996). *Object Models: Strategies, Patterns and Applications*, Prentice Hall, ISBN 0138401179

Coplien, J. O. (1992). *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, ISBN 9780201548556

Cossentino, M.; Sabatucci, L. & Chella, A. (2004). Patterns Reuse in the PASSI Methodology. In *Engineering Societies in the Agents World*, Springer, LNCS, Vol. 3071/2004, 520, ISBN 978-3-540-22231-6

Couturier, V. (2004). *L'ingénierie des systèmes d'information coopératifs : une approche à base de patterns*. Unpublished doctoral dissertation, Université Jean-Moulin Lyon 3, France (in French).

Couturier, V.; Huget, M.-P. & Telisson, D. (2010). Engineering agent-based information systems: a case study of automatic contract net systems, *Proceedings of the 12th International Conference on Enterprise Information Systems* (ISAS - ICEIS 2010), Portugal, Volume 3, pp. 242-248

Davis, R. & Smith, R. G. (1983). Negotiation as a Metaphor for Distributed Problem Solving, *Artificial Intelligence*, 20(1), pp. 63-109.

Do, T.T.; Kolp, M.; Hang Hoang, T.T. & Pirotte, A. (2003). A Framework for Design Patterns for Tropos, *Proceedings of the 17th Brazilian Symposium on Software Engineering*, Brazil

FIPA (2002). FIPA ACL Message Structure Specification, 2002.

Fowler, M. (1997). *Analysis Patterns: Reusable Object Models,* Addison-Wesley, ISBN 0201895420

Fowler, M. (2002). *Patterns of enterprise application architecture*, Addison Wesley, ISBN 978-0321127426

Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*, Addison Wesley, ISBN 0201633612

Gutknecht, O. & Ferber, J. (2000). The MADKIT agent platform architecture. In T. Wagner (Ed.), LNCS : vol. 1887. *International Workshop on Infrastructure for Multi-Agent Systems*, London, Springer-Verlag, pp. 48-55

Hay, D. C. (1996). *Data Model Patterns: Conventions of Thought,* Dorset House, ISBN 0932633293, New York

Hung, E. & Pasquale, J. (1999). *Agent Usage Patterns: Bridging the Gap Between Agent-Based Application and Middleware,* technical report CS1999-0638, Department of Computer Science and Engineering, University of California

Jureta, I.; Kolp, M.; Faulkner, S. & Do, T.T. (2005). Patterns for agent oriented e-bidding practices, *Knowledge-Based Intelligent Information and Engineering Systems (KES'05)*, Lecture Notes in Computer Sciences 3682, Springer, pp. 814 – 820

Kendall, E. A.; Murali Krishna, P.V.; Pathak, C.V. & Suresh, C.V. (1998). Patterns of intelligent and mobile agents. In Proceedings of the *Second International Conference on Autonomous Agents*, Minneapolis, Minnesota, USA, pp. 92–99

Meira, N. ; Silva, I.C. & Silva, A. (2001). An Agent Pattern for a More Expressive Approach, *Proceedings of the EuroPLOP'2000*, Germany

Oluyomi, A.; Karunasekera, S. & Sterling, L. (2007). A comprehensive view of agent-oriented patterns, *Autonomous Agents And Multi-agent Systems*, Volume 15, Number 3, pp. 337-377, DOI: 10.1007/s10458-007-9014-9

Renzel, K. & Keller W. (1997). Client/Server Architectures for Business Information Systems - A Pattern Language, *Pattern Languages of Program (PLOP'97)*, September 3-5, Monticello, Illinois, USA

Risi, W.A. & Rossi, G. (2004). An architectural pattern catalogue for mobile web information systems, *International journal of mobile communications*, vol. 2, no3, pp. 235-247, ISSN 1470-949X

Saidane, M. (2005). *Formalisation de Familles d'Architectures Logicielles Coopératives : Démarches, Modèles et Outils*. Unpublished doctoral dissertation, Université Joseph-Fourier - Grenoble I, France (in French).

Sauvage, S. (2003). *Conception de systèmes multi-agents: un thésaurus de motifs orientés agent*. Unpublished doctoral dissertation, Université de Caen, France (in French).

Schelfthout, K.; Coninx, T.; Helleboogh, A.; Steegmans, E. & Weyns, D. (2002). Agent Implementation Patterns. *Proceedings of workshop on Agent-Oriented Methodologies, 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*

Tahara, Y.; Ohsuga, A. & Honiden, S. (1999). Agent system development method based on agent patterns. In Proceedings of the *Fourth International Symposium on Autonomous Decentralized Systems*.

Wooldridge, M. (2002). *An introduction to Multi-Agent Systems*, John Wiley & Sons, ISBN 0470519460