Agent-oriented Requirements Modeling

Liang Xiao
Supervisor: Dave Robertson
School of Informatics, University of Edinburgh
s0231274@sms.ed.ac.uk
September 6, 2003

Abstract

A standard way of describing requirements for system design is according to function. One describes high level functions of the operation of the system and decomposes these into lower level functions which perhaps overlap. For complex systems the number of functions can be large and they may interact, making it important to have a methodical way of describing the structure of functions and their interactions.

Agent-Oriented Software Engineering methodology addresses complex systems by three strategies: decomposition, abstraction, and organization. This is a useful and effective way to solve problems because it gives one a clear way to decompose the problem into agents. In addition, an agent-based system is endowed with some degree of automation and self-adaptation in that agents are autonomous entities and they are "intelligent" in some aspects. This makes the system more flexible and robust.

In this paper, we demonstrate our belief that, agent technology, with its virtue of domain knowledge capture capability and high-level abstraction for interactions, can be applied to Requirements Engineering.

In Section1 of the paper we demonstrate how we are inspired to adopt extended UML and XML for representing and encoding requirements knowledge. We compare agent-oriented requirements modeling with traditional descriptive functional-based requirements representation and argue that the new modeling approach has some fundamental advantages. In Section2 we introduce related work on agent-oriented Software Engineering, Requirements Engineering and AUML, an integration of these techniques

brings us distinctive features and benefits. In Section3, we give readers an impression of what the new requirements look like and the steps to build agent-oriented UML diagrams including identification of agents and their interactions. Later on we discuss the detailed notation systems and apply the agent-oriented approach to represent a portion from an existing Rail Track requirements document in Section4. Then we introduce in Section5 our specially designed agent-oriented UML CASE tool and illustrate how to do requirements transformation in detail with this tool, in this section we also introduce the tool's integrated functionalities of automatic framework code generation and architecture consistency validation. Finally we evaluate our approach, make a conclusion and discuss open issues and possible further work.

1 Introduction

The aim of this paper is to investigate a way to reconstruct a functional description in an agent-oriented style, where the main components of the description will be agent (not function) definitions and the main interactions between definitions will be via message passing (these messages being transfer of "conceptual" information as well as the more normal style of agent messaging). We will apply this transformation to a part of an existing document describing the Production Function of a national rail operator. This document is large (over 250 pages) and adopts a uniform system of function description throughout.

The original document is a collection of function statements and other static knowledge, they are co-related and overlapped. This certainly does not give us advantages of easy visual inspection, which means we may lack clear recognition of the whole system structure. In addition, when part of the document changes, it is not easy to guarantee that the whole system remains consistent because it is very likely that changes to other parts of the document are omitted.

The increasing use, high availability and familiarity of object-oriented techniques and tools inspire our adoption of the graphically expressed modeling standard UML (Unified Modeling Language) as a basic notation with which to describe agents and their interactions.

As UML is an object-oriented modeling technique, and does not account for high level functionalities belong to agents, it is not adequate if we simply use the original UML notation. We propose to extend UML so that the extended version can accommodate agent specific features.

We add a role element to the extended UML diagram, roles are played by agents, and they are behaviors of agents. With the use of role, we extract the functions from original requirements document, assign them to agents and reflect in the diagram relationships between agents and roles, also collaborations between roles.

We also add an agent interaction message element to the extended UML diagram. There may be a sample message given in practice in the diagram. It gives definition of message content format. Both agent/role definition and message content definition are represented in XML.

We take the de facto standard document format XML for the representation of communication messages between agents. We format these described messages in the original document to the specified interaction content in XML. We define the format structure which means we set rules to which agents must conform when they communicate. Only messages that are expressed according to this semantics are supposed to appear in transmission between certain agents.

We will give details of how to do the XML translation in Section4 of the paper.

Document Type Definition (DTD) and XML Schema are XML document validation techniques, we may use them to guarantee that entities are formatted to desired XML structure and they are in correct interrelated graph. (This may be aided by the use of XLink/ XPointer)

The idea of this diagram-based requirements document construction is to let agents to be represented as UML elements, which carry domain knowledge. They interact with each other using messages, passed on from one agent to another to exchange knowledge. Business behaviors are organized to roles that are played by agents. Roles are also represented as UML elements in the diagram. The system is in operation when messages are delivered among agents and roles are played. Descriptive requirements are captured and constituted to the definition and interrelationship of agents, roles and messages.

Much of the important and crucial domain functional knowledge can be captured in this way of requirements representation. Moreover, it helps visual inspection and adds to higher accessibility to related entities in the diagram which are not shown so explicitly in the declarative document, and it may also bring the benefits of more easily derivation of additional knowledge and logical reasoning.

With this requirements description transformation, we argue the development process can go more smoothly. On completion of the agent-oriented UML diagram, the transition from requirements analysis phase to design phase is fairly easy. The UML diagram can be extended to an integrated diagram of both agents and classes when we start to design. This is sensible because an agent-oriented Software System can be developed from the agent-oriented requirements modeling (more discussion following), this Software System in turn can be implemented with object-oriented technology. Agents are higher level elements while classes are lower level blocks. Agents may be subsystems or an organization of classes; their interactions are main business flows in the system. We identify agents and their interactions during the requirements capture phase, drawing diagrams to reflect the relationships so that the design phase is only to figure out detailed classes and other aspect of refining design. Combining agents and classes together gives an integrated modeling approach using the UML a major advantage.

Here is such an agent/class combined diagram from [1]:

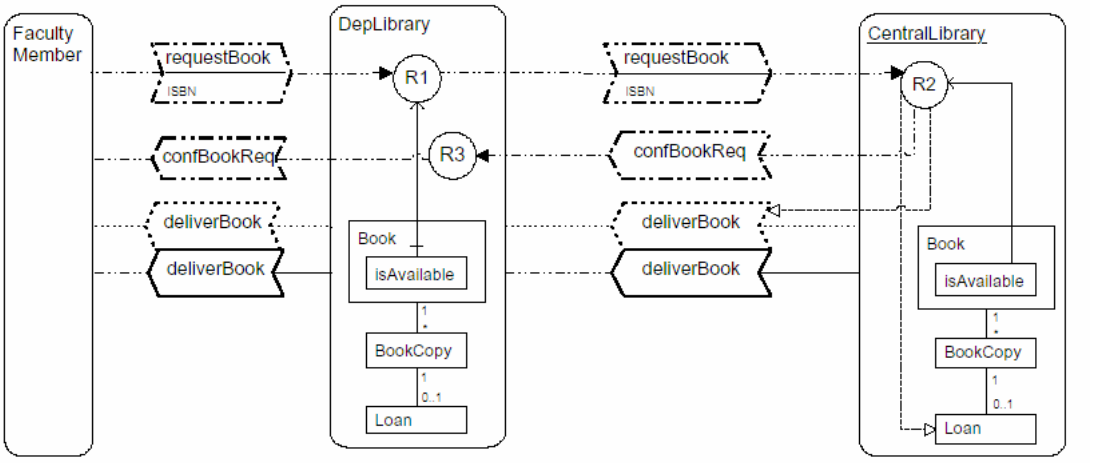


Figure 1. An interaction pattern diagram describing the process type where a faculty member requests a book from a department library such that the request is forwarded to the central library because the requested book is not available at the department library

In the diagram, FacultyMember, DepLibrary and CentralLibrary are agents while Book and BookCopy are classes. The consideration of what classes are essential in agents and their relationships can be put in later design phase. However this kind of diagram is certainly built on top of the agent-oriented requirements diagram.

2 Background - Related Work

2.1 Agent & Agent Architecture

As stated in [2], the following definition of agent is useful:

An agent is an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives

The author N. R. Jennings also points out that, agents are: *(i) clearly identifiable problem solving entities with well-defined boundaries and interfaces; (ii) situated (embedded) in a particular environment—they receive inputs related to the state of their environment through sensors and they act on the environment through effectors; (iii) designed to fulfil a specific purpose—they have particular objectives (goals) to achieve; (iv) autonomous—they have control both over their internal state and over their own behaviour; (v) capable of exhibiting flexible problem solving behaviour in pursuit of their design objectives—they need to be both reactive (able to respond in a timely fashion to changes that occur in their environment) and proactive (able to opportunistically adopt new goals).* Moreover, the agents will need to interact with one another to achieve their individual objectives. The interactions are conducted at the high-level (knowledge-level) agent communication language [3]: in terms of which goals should be followed, at what time, and by whom. Agents need to make context-dependent decisions and initiate or respond to interactions that were not foreseen at design time.

As according to Mike Wooldridge [4], agents are intelligent and autonomous. They perform communication acts in the furtherance of their intentions and desires; negotiate each other to reach agreements just like human beings do. Probably one of the best-known and most-implemented agent architectures in the literature, the belief-desire-intention architecture, is intended to enable an agent to make good decisions without any help. In this architecture [5], "*decision making is viewed as a process of practical reasoning of the kind that we humans do every day: in order to decide what to do, an agent starts by looking at the world and updating its beliefs about how the world is, and on the basis of this, deciding what options are available to it. Having determined a set of options, an agent must then fix upon some subset of these possibilities, and commit to achieving them. Once an agent has committed to an option, it becomes an intention, which focuses the agent's*

future actions: it must typically develop some appropriate recipe or plan for achieving the intention, and start executing this plan. But an agent cannot blindly execute a plan, without ever stopping to consider how the world is - from time to time, it must reconsider its intentions, by deliberating over them, in order to determine whether a change of focus is necessary."

2.2 Agent-Oriented Software Engineering

Agent-Oriented Software Engineering is being described as a new paradigm [6] for the research field of *Software Engineering.*

Agent-oriented programming (AOP) can be seen as an extension of object-oriented programming (OOP). In OOP the main entity is the object. An object is a logical combination of data structures and their corresponding methods (functions). Objects are successfully being used as abstractions for passive entities in the real-world, and agents are regarded as a possible successor of objects since they can improve the abstractions of active entities [7].

Other advantages of agents over objects are: agents support structures for representing mental components, i.e. beliefs and commitments; they support high-level interaction (using agent-communication languages like FIPA ACL and KQML) between agents as opposed to ad-hoc messages frequently used between objects [6]. In addition, objects are controlled from the outside (white box control), as opposed to agents that have autonomous behaviour which can't be directly controllable from the outside (black box control). In other words, agents have the right to say "no" [8].

GAIA, one of the first methodologies that have been specifically developed for agent-based systems [9], is presented by Wooldridge, Jennings and Kinny [10, 11] for agent-oriented analysis and design. Gaia is a general methodology that supports both the micro-level (agent structure) and macro-level (agent society and organization structure) of agent development. Using Gaia, software designers can systematically develop an implementation-ready design based on system requirements. The first step in the Gaia *analysis* process is to find the *roles* in the system, and the second is to model *interactions* between the roles found. In the Gaia *design* process, the first step is to map roles into *agent types*, and then to create the right number of *agent instances* of each type. The second step is to determine the *services model* needed to fulfill a role in one or several agents, and the final step is to create the *acquaintance model* for the representation of communication between the agents.

Wood and DeLoach [12, 13] suggest the Multiagent Systems Engineering Methodology (MaSE). MaSE is similar to Gaia with respect to generality and the application domain supported, but in addition MaSE goes further regarding support for automatic code creation through the MaSE tool. The goal of MaSE is to lead the designer from the initial system specification to the implemented agent system. The MaSE methodologies are divided into seven sections (phases) in a logical pipeline: *capturing goals, applying Use Cases, refining roles, creating agent classes, constructing conversations, assembling agent classes, system design.*

Following the steps of either methodology we can build an *agent-based system*, the key abstraction used in which is that of an *agent.* Such a system potentially consists of multiple agents. They encapsulate both state and behaviour, and communicate via message passing. An *agent* itself is a *rational decision making system*, which enjoys the following properties: *autonomy*, *reactivity*, *pro-activeness*, and *social ability* [14].
Although a wide range of well-known Software Engineering paradigms have already been devised to deal with complexity in software including: object-orientation, component-ware, design patterns and software architectures, agent-oriented Software Engineering is emerging as a more efficient and influential technique. Booch [15] identifies three tools to tackle complexity in software: *Decomposition, Abstraction* and *Organisation.* In fact they are available not only in object-oriented approach, but also in agent-oriented approach. As these tools are more intuitively designed to tackle complex problems, they become even more powerful.

In terms of complex software, complexity manifests itself as a large number of sub-systems that have many interactions [16]. Given this state of affairs, the role of Software Engineering is to provide models and techniques that make it easier to handle this complexity [2]. In the other hand, agents are modularised as components in terms of the objectives they achieve. This philosophy to objective-achieving *decompositions* means that the individual components of agents localise and encapsulate their own control. Thus, entities have their own thread of control and they have control over their own choices and actions. They are endowed with the ability to initiate and respond to interactions in a flexible manner at run-time and, as a result, they are able to satisfy the unpredictable necessity according to dynamic situations caused by the system's inherent complexity.

As active and autonomous components, agents intend to fulfil both their individual and collective objectives, they can be viewed as intentional systems whose behaviour can be predicted and explained in terms of *attitudes* such as belief, desire, and intention [17], the behaviour of a complex system is understood via the attribution of attitudes such as

believing and desiring. It is convenient shorthand for talking about complex systems, which allows us to succinctly predict and explain their behaviour without having to understand how they actually work. System complexity can eventually be managed with greater ease by using the intentional stance of agents as an *abstraction tool* [14].

In addition, individual agents or *organisational* groupings can be developed in relative isolation and then added into the system in an incremental manner. This ensures there is a smooth growth in functionality.

Agent-oriented approaches can significantly enhance our ability to model, design and build complex (distributed) software systems [2].

Moreover, existing AI and Knowledge techniques can also help to build agents, as they are intelligent and rational systems. Wamberto Vasconcelos, David Robertson [18], etc. use formal methods and introduce a lifecycle for models of large multi-agent systems.

2.3 Requirements Engineering

"*Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families.*" [19]

According to Bashar Nuseibeh [20], *Requirements Engineering* (RE) is the process of discovering the purpose which a *Software System* was intended for, by identifying stakeholders and their needs, and documenting these in a form that is amenable to analysis, communication, and subsequent implementation. This process is inherent difficult because stakeholders (including paying customers, users and developers) may be numerous and distributed. Their goals may not be not explicit and difficult to articulate, they may also be vary and conflict depending on their perspectives of the environment in which they work and the tasks they wish to accomplish. [21]

Through an understanding of beliefs of stakeholders, RE specify a problem to be solved, identify system boundaries, analyse properties of the environment, and characterise the behaviours of the resulting software.

Core RE activities include:

· *Eliciting* requirements

Elicitation is to find out what problem needs to be solved, and hence identify system *boundaries*. Identifying and agreeing a system's boundaries affects all subsequent elicitation efforts. The identification of stakeholders and user classes, of goals and tasks, and of scenarios and use cases all depend on how the boundaries are chosen.

· *Modelling* and *analysing* requirements

Modelling, the construction of abstract descriptions that are amenable to interpretation is a fundamental activity in RE. There are several general categories of RE modelling approaches:

Enterprise modelling is often used to capture the purpose of a system, by describing the behaviour of the organisation in which that system will operate.

Data modelling provides the opportunity to address issues like what information is to be understood, manipulated and managed in RE, and how to represent them.

Behavioural Modelling: Modelling requirements often involves modelling the dynamic or functional behaviour of stakeholders and systems, both existing and required.

Domain Modelling: A significant proportion of the RE process is about developing *domain descriptions* [22]. A model of the domain provides an abstract description of the world in which an envisioned system will operate.

Another important issue in requirements modelling is the modelling of *Non-Functional Requirements (NFRs)*; they tend to be properties of a system as a whole, such as safety, security, reliability, and usability. They are generally more difficult to express in a measurable way, but there is a growing research interest in tackling them.

Modelling requirements provides the opportunity for analysing them.

· *Communicating* requirements

RE facilitates effective communication of requirements among different stakeholders. The way in which requirements are documented plays an important role in ensuring that they can be read, analysed, (re-)written, and

validated.

The focus of requirements documentation research is often on specification languages and notations, with a variety of formal, semi-formal and informal languages suggested. Different languages have been shown to have different expressive and reasoning capabilities.

Natural language

· Extremely expressive and flexible
· Very poor at capturing the semantics of the model
· Better used for elicitation, and to annotate models for communication

Semi-formal notation

· Captures structure and some semantics
· Can perform (some) reasoning, consistency checking, animation, etc.
E.g.: diagrams, tables, structured English, etc.

Formal notation

· Very precise semantics, extensive reasoning possible
· Long way removed from the application domain
Requirements formalisms are geared towards cognitive considerations, hence differ from most computer science formalisms.

· *Agreeing* requirements

As requirements are elicited and modelled, maintaining agreement with all stakeholders can be a problem, especially where stakeholders have divergent goals. Explicitly describing the requirements is a necessary precondition not only for validating requirements, but also for resolving conflicts between stakeholders.

Requirements negotiation attempts to resolve conflicts between stakeholders without necessarily weakening satisfaction of each stakeholder's goals. Early approaches to requirements negotiation focused on modelling each stakeholder's contribution separately rather than trying to fit their contributions into a single consistent model [23] and on the importance of establishing common ground [24]. Boehm introduced the win-win approach [25] in which the win conditions for each stakeholder are identified, and the software process is managed and measured to ensure that all the win conditions are satisfied, through negotiation among the stakeholders. There are other such models to promote agreement with similar process: compare functional requirements with one another, identify the most important goals

of each participant, and ensure these goals are met.

There are contextual issues, including contractual and procurement issues in agreeing requirements. Requirements engineers are supposed to investigate organisational and social context in which the new system will operate and, interact with a variety of stakeholders, including potentially non-technical customers, systems designers and developers before an agreement is reached.

· E*volving* requirements

Successful software systems always evolve as the environment in which these systems operate changes and stakeholder requirements change. In Software Engineering, it has been demonstrated that focusing change on program code leads to a loss of structure and maintainability [26]. It is also noted that requirements errors, such as misunderstood or omitted requirements, are more expensive to fix later in the lifetime of project lifecycles [27; 28]. Therefore *managing change* is a fundamental activity in RE [29]. Changes to requirements documentation need to be managed through configuration management, version control [30], and traceable links in the documentation so that impact of changes and risk of the project can be monitored and controlled.

RE is a multi-disciplinary activity, deploying a variety of techniques and tools at different stages of development and for different kinds of application domains. A variety of approaches have been suggested to manage and integrate different RE activities and products. Jacskon uses problem frames to structure different kinds of elementary and composite problems [31]. An alternative approach to organising, selecting and tailoring multiple methods is through the use of multiple perspectives or views of requirements [32; 33]. A viewpoint can be treated as an encapsulation of an individual technique, with a defined notation, a set of actions that can be performed on that notation, and a set of rules for consistency relationships with other viewpoints.

RE is increasingly recognised as a critically important activity in Software Engineering processes. Many delivered systems do not meet their customers' requirements due, at least partly, to ineffective RE. Effective RE approaches will play a crucial role in the management of change in software development, in making assessment of feasibility and associated risks of projects that to be undertaken, in determining the success or failure of projects, and in determining the quality of systems that are delivered.

2.4 Use Agents for Requirements Engineering

Agent-oriented approaches are becoming popular for requirements modelling; they are introduced in Requirements Engineering (RE) mainly to characterize active elements in the environment. Several such RE frameworks are briefely reviewed in [34]: *Composite Systems Design and KAOS*; *Albert II*; *The F3 framework*; *The i* modelling framework.* In *i**, the term *agent* is used to refer to the concrete, implementable variety, and therefore whose identity is determined by physical and implementability criteria. The author of the above paper argues that for an RE framework to be truly agent-oriented, the identity and existence of an agent needs to be determined within the RE level, based on RE criteria, not on implementation level criteria. If agent identity and existence are pre-determined, the RE process may not be benefitting much from having an agent construct. A distinctive RE agent can serve as a powerful abstraction mechanism and a concept of agent for RE that is ontologically distinct from those in design and implementation is needed. This is the viewpoint we agree with.

Albert II supports the modelling of functional requirements in terms of a collection of agents interacting in order to provide services necessary for an organization, notion of agent is seen as a way of organizing the specification so that behaviour pertaining to each agent is collected together. Similarly to this approach, we start with the functional requirements specification, identify logically separate agent components, and assign lists of actions they can perform to them. Although the agent/role during requirements modelling are simply turned into class/method during implementation (automatic code generation) step, it is supposed that an agent/role refining design step is essential between these two tasks, so that requirements analysis and implementation are bridged by the design.

2.5 AUML

Agent UML specification has recently been defined by FIPA, a non-profit organization aimed at producing standards for the interoperation of heterogeneous software agents. FIPA AUML class diagrams extend UML class diagrams to cover needs for agent design. In the context of agents and multi-agent systems, FIPA AUML class diagrams describe the agents and their architectures [35]. Their use of AUML introduces new notations for representations and focuses on the process that various agents playing roles hence a sequence of actions happen one after another thus accomplish a certain task. According to the definition, agents are autonomous entities act and react on its own right. Communication protocols between agents as well

as several kind of agent diagrams including Sequence Diagram, Interaction Overview Diagram, Communication Diagram and Timing Diagram [36] are defined to describe the intra-agent or inter-agent activities; *levelling* [37] is used to express detailed interaction process. Their use of agents emphasizes design issues and hence aids the development later on.

Agent-based UML in this project, although similarly designed in principal to embrace agents with the traditional UML, is used to capture requirements knowledge. An easy to understand notation system is used just like a Collaboration Diagram from class diagram. The diagram focuses on agents, roles and relationships where message passing is also a central component. In the diagram, the architecture of internal structure is reflected and it is a straightforward UML extension to support additional functionalities. We only adopt the very basic concepts of: Agent, Role, Interaction and Message from [38]. We also only concentrate on interactions between pairs of agents, therefore the model is simplified and how data flows among components to accomplish a goal is not taken into account. Fundamental system structure will emerge after such a diagram is completed, thorough design will be needed afterwards.

Relatively complicated notation systems are often used by FIPA to accommodate the interactions between intelligent agents and give details of how to accomplish the complex tasks while designing. More concepts, notations and diagram views are designed to give exhaustive solutions for agent-based system development. This project is not for that goal but primarily for two main tasks, the requirements document transformation and framework code generation. The comparatively simple concepts and easy-to-understand notation system are intended to be intuitive; a specially designed and user-friendly graphical CASE tool is implemented; stipulated guidelines according to which agent-oriented UML diagrams are to be constructed from original requirements are given, these are sufficient for our specific purpose. There is a design gap between the two tasks described above, which must be bridged by the human interventions: some additional components and relationships are supposed to be added to the transformed diagram manually through designers' understanding, and then go on to generate the right code. The combination of transforming requirements to diagrams and transforming drawn diagram to framework code is a unique feature. This whole process facilitates the agent-based development from the beginning to the end.

3 Agent-oriented Requirements Modeling Diagram

The agent-oriented UML diagrams are used to document the architecture of the system through a high level abstraction; it captures requirements knowledge and organizes them to be accommodated by agents. Interrelated agents interact with each other to exchange knowledge.

The following is a sample diagram discussed in this paper. It's drawn for a Train Running part of the Rail Track requirements document. We will give details of it in Section4.

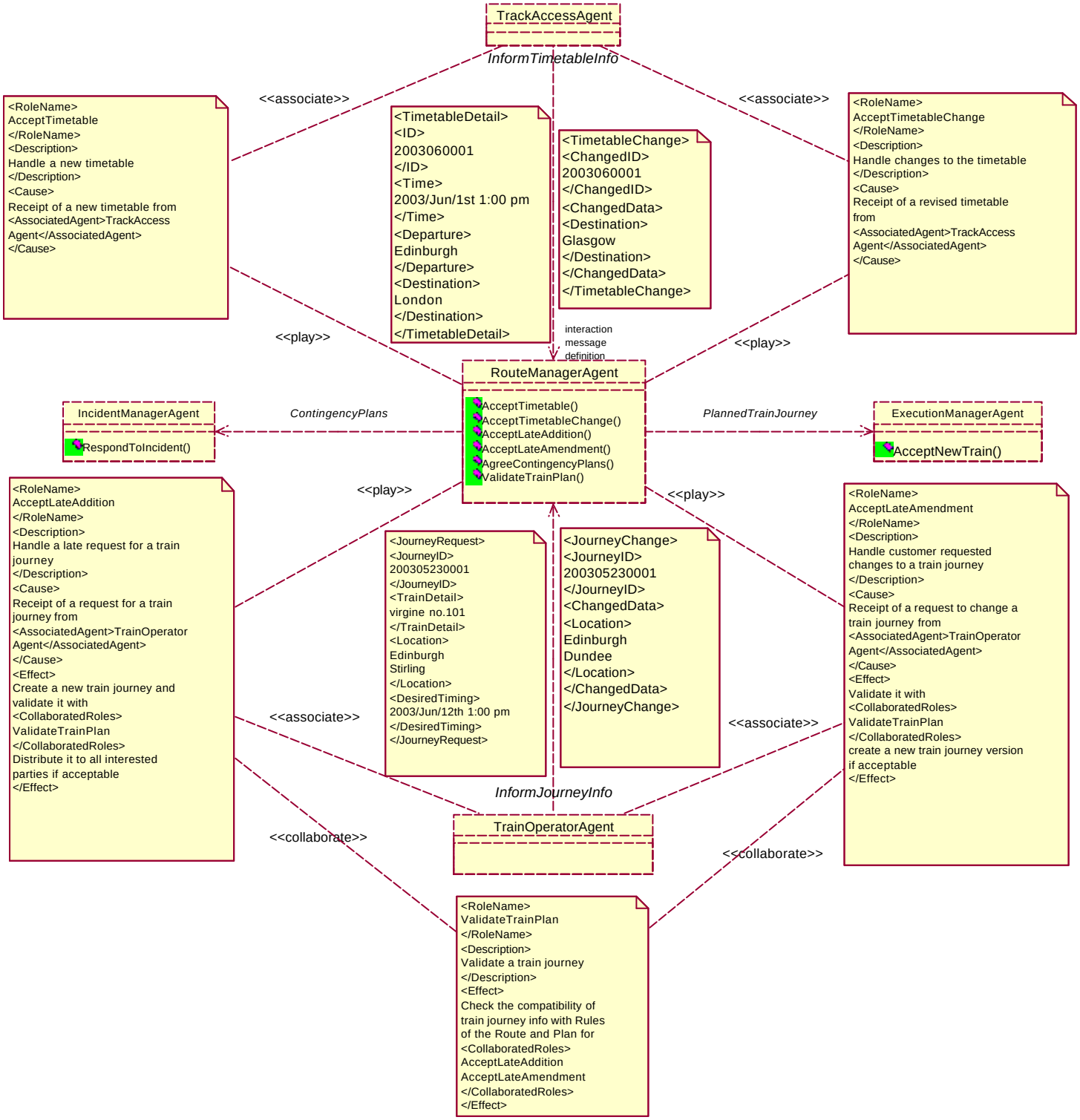


Figure 2. A sample agent-oriented requirements diagram, for a portion of Train Running part of the Rail Track requirements document

The diagram is somewhat similar to a class diagram and its design is really stimulated by that. However it is higher level so we do not care about what kind of classes we need in an agent, not to mention attributes. Nevertheless agents (like classes); their relationships (like associations) and roles played by agents (like methods of classes) are central part of the diagram.

We give below the process to build an agent-oriented UML diagram from information derived from the original requirements:

1. Identify subsystems and delegate an agent for each. An agent is a relatively independent unit and it should play a variety of related roles. Consider consolidating agents if roles that they play are related. Consider to split agents if one agent plays completely different roles.
2. Connect related agents, one agent may deliver requests and trigger another agent to play a certain role. The interaction message between them should be given in an XML content format definition so that the process of producing messages by one agent, transmitting messages from this one agent to another, and parsing the messages by the latter agent could conform to a certain manner.
3. Identify roles each agent play, give agent and role definitions in XML. Add <<play>> connections between roles and the agent which plays them. There is surely a reference of each role an agent plays in the agent box. Roles are identified through the function description from original requirements document. XML definitions are given in Section3.
4. Parse <cause> tag in role definition XML, give <<associate>> connections between roles and agents which cause they to be played.
5. Parse <effect> tag in role definition XML, give <<collaborate>> connections between interrelated pairs of roles that one role is aided by another to accomplish a certain goal.

After step 5 is finalized, it is optional to continue:

6. Validate the model: Eliminate whatever elements that are redundant which may be caused by the duplicated information from original document; Find chances to give connections that are not explicit in original document but that are logically sensible, also try to consolidate related information.

Additional Suggestion:

Do not incorporate detailed objects to agents now; this is to be addressed at the later design phase along with issues at object level like reuse optimization and inheritance.

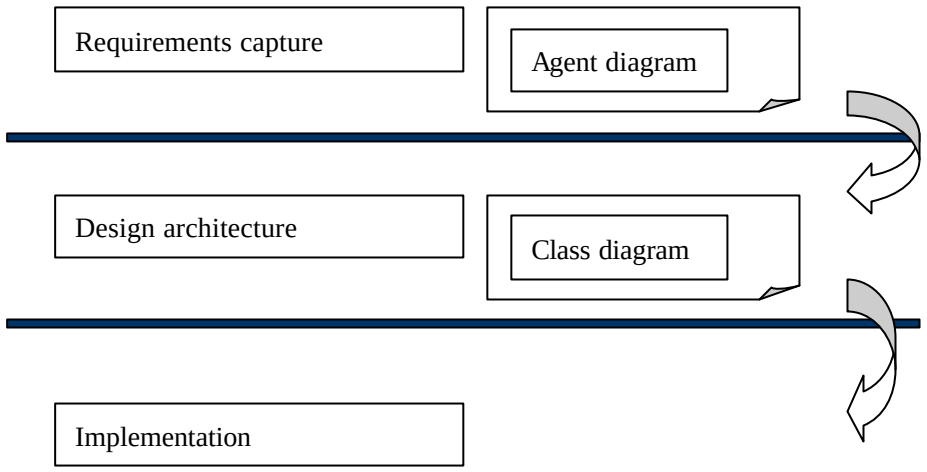


Figure 3. Position of the agent diagram in Software Engineering process layers

4 Sample transformations

The original requirements document is more than 250 pages, expressed in natural English language. The document defines the background and scope of the problem domain. It describes what the system will do in three areas: Train Running and Performance; Infrastructure Management; Performance and Common Communication. Throughout this part of description, a standard format is applied to depict system operations as Production Function Tables like the one shown in Figure7. One table is used to define one function. To one single function, the Identifier and Description of the function, the Cause that invokes the function, the Assumptions that have to be satisfied before this function is invoked, Information Used while invoke this function, Output of this function after the invocation along with Required Effect and other properties like Safety are given in its corresponding Function Table. These account for a significant part of the whole document. The requirements also provide Data Model and Dataflow in and out among Functions.

We take a small part of the Rail Track requirements document and show the transformation process in this section according to the steps listed in Section3. Note some steps can be operated in consolidation or separation and it is not necessary to strictly conform to the 5/6-step-transformation.

1. There are three main areas under concern in the original Rail Track Production Function requirements document. We focus on the first area: Train Running and Performance. Each of three areas is sub-divided into Business, Incident and Execution domains. So in the first step of transformation we delegate three agents: RouteManagerAgent, IncidentManagerAgent and ExecutionManagerAgent. We make most efforts to illustrate RouteManagerAgent in this paper.

2. Establish ***agent*** elements structure in interconnection in a diagram

An agent represents a scope of knowledge in the view of requirements capture; it can be a subsystem or a group of objects in the view of design and an autonomous entity in the view of implementation.

Functions categorized in the original documents by domains are organized to roles assigned to agents. We pick those primary functions of RouteManagerAgent and list them as roles that it plays in the following very basic agent diagram:

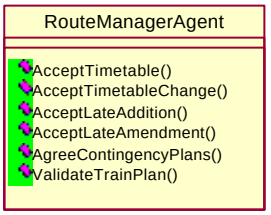


Figure 4. A single agent diagram

XML content of this agent:

<AgentName>RouteManagerAgent</AgentName>
<RolesGroup>
<Role>AcceptTimetable</Role>
<Role>AcceptTimetableChange</Role>
<Role>AcceptLateAddition</Role>
<Role>AcceptLateAmendment</Role>
<Role>AgreeContingencyPlans</Role>
<Role>ValidateTrainPlan</Role>
</RolesGroup>
<Collaborators>
IncidentManagerAgent
ExecutionManagerAgent
</Collaborators>

A good point of using XML can be seen here, we can easily extend the XML

structure to accommodate more knowledge the agent is aware of by simply adding additional tags. In the above case, it is very convenient to redefine agents by adding more <Role> tags to expand functionalities of agents to adapt them to the mutable requirements.

As stated in the description of function RespondToIncident, the IncidentManagerAgent handles perturbations to train journeys by playing this role. In the "Information Used" block, it says contingency plans will be used to make amended train journeys. Since contingency plans are produced by RouteManagerAgent playing the role of AgreeContingencyPlans, we have a clue that there is a connection between IncidentManagerAgent and RouteManagerAgent so that contingency plans can be provided by the latter agent to the former one.

Figure 5. Two agents with a connection between them

As it is hard to imagine how these plans are structured, we choose another diagram to illustrate the XML format of interaction message content.

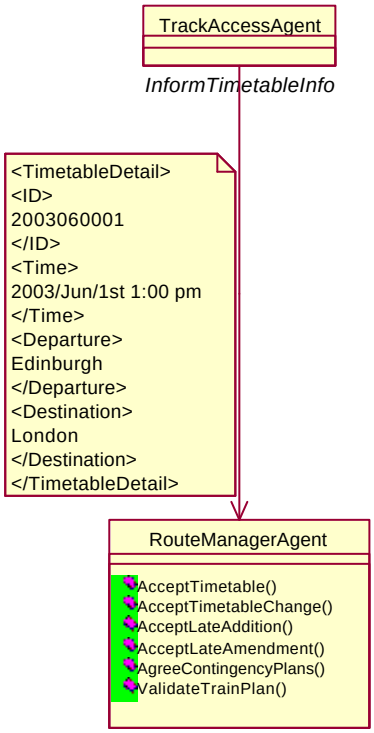


Figure 6. Two agents with a message passing between them

RouteManagerAgent accepts timetables from TrackAccessAgent; timetable structure may be organized in the way shown in the above diagram. It is clear such message semantics is set at the beginning and should be conformed since then. Corrupted messages can be found by XML parser during validation check.

3. Establish ***role*** elements in the diagram and connect them to agents that play these roles:

A role is played by an agent; it represents a capability of the agent. A role captures a function description in the view of requirements capture; it can be a method in the view of design.

These identified roles belong to RouteManagerAgent:
AcceptTimetable (),
AcceptTimetableChange (),
AcceptLateAddition (),
AcceptLateAmendment (),
AgreeContingencyPlans (),
ValidateTrainPlan ()

We establish graphical elements for these roles in the diagram and connect each role with the agent which plays it.

Identify agents and roles they play are the core work for this agent-oriented requirements representation transformation. We format role descriptions to basically four main XML segments embedded by four key tags in the following way.

We illustrate this with an example of role AcceptLateAddition. Before that we give its original function description document.

Accept Late Addition

Description	To handle a late request for a train journey.
Cause	Receipt of a request for a train journey directly from a ***Train operator*** or from the driver entering the production function's area. ***The request is provided in the form of a combination of relevant train details, locations and desired timings***.
Assumption	The crew is competent for the route requested.
Information Used	Relevant locations.
Outputs	A new train journey, to Train Operator and others.
Required Effect	A new train journey is created from the request, and validated (***PF.TR.B-ValidateTrainPlan***). If the train journey is acceptable then it is distributed to all interested parties; otherwise the request is rejected or renegotiated. Having been accepted, the new train journey is known to the Production Function. The train journey is made known outside the Production Function (PF.CC.B-ProvideTrainPlan).
Frequency	Currently 20-100 per day.
Timeliness	-
Impact of Unavailability	The unavailability of this function would render the Production Function unable to respond at short notice to customer requests for additional train paths.
Safety	-
Comment	-
Identifier	PF.TR.B-AcceptLateAddition

Figure 7. Original function description document

Four XML tags for role document definition:

1). <RoleName> Tag: the name of the role, just taken from original document. The same naming convention should be adopted in all role documents and the same name should be applied throughout requirements capture, design and implementation.

<RoleName>AcceptLateAddition</RoleName>

2). <Description> Tag: a short description for the role, it should capture its main functionality, maybe a shorter and more concise version of the original one.

<Description>Handle a late request for a train journey</Description>

3). <Cause> Tag: describes what causes the role to function. In most cases, there is an <AssociatedAgent> tag inside this one, showing which external agent requests this agent to play the role. These important pieces of information may be given directly in the "Cause" block of the original document description but it is also very likely to need human understanding and reasoning.

<Cause>
Receipt of a request for a train journey from
<AssociatedAgent>TrainOperatorAgent</AssociatedAgent>
</Cause>

4). <Effect> Tag: describes what the effect is after the role is played. In some cases, there are also <CollaboratedRoles> tags inside this one, showing there are other roles that this role seeks help to accomplish a certain task. By contrast, when there is no such tag, this role is capable to fulfill the goal by itself. Some information in the "Required Effect" block of the original document description is useful in the late design phase as it describes the necessary processing details, which may be best viewed and understood through object-level analysis.

<Effect>
Create a new train journey and validate it with
<CollaboratedRoles>ValidateTrainPlan</CollaboratedRoles>
Distribute it to all interested parties if acceptable
</Effect>

The complete XML document for role AcceptLateAddition:

<RoleName>AcceptLateAddition</RoleName>
<Description>Handle a late request for a train journey</Description>
<Cause>
Receipt of a request for a train journey from
<AssociatedAgent>TrainOperatorAgent</AssociatedAgent>
</Cause>
<Effect>
Create a new train journey and validate it with
<CollaboratedRoles>ValidateTrainPlan</CollaboratedRoles>
Distribute it to all interested parties if acceptable
</Effect>

The relationship diagram for role AcceptLateAddition and the agent RouteManagerAgent which plays it:

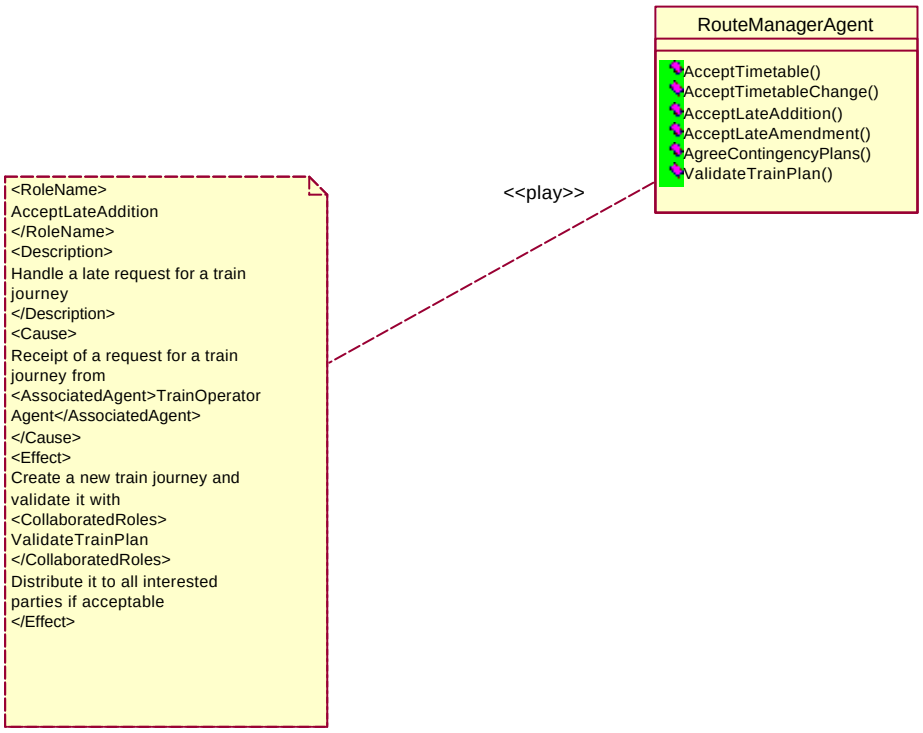


Figure 8. A role is played by an agent

4/5. Establish ***association/collaboration*** relationships

As we can see clearly from the above XML format definition for role AcceptLateAddition, TrainOperatorAgent embedded in the AssociatedAgent tag is the agent that causes the role to function; and ValidateTrainPlan in the CollaboratedRoles tag is the role it seeks help to accomplish its task. We add a dotted line <<associate>> between role AcceptLateAddition and agent TrainOperatorAgent to illustrate it is TrainOperatorAgent that causes role AcceptLateAddition to play. We also add a dotted line <<collaborate>> between role AcceptLateAddition and role ValidateTrainPlan to illustrate that it is ValidateTrainPlan from which role AcceptLateAddition seeks help.

The following is a partial diagram to demonstrate the relationship between these entities: (There ought to be a play connection between RouteManagerAgent and ValidateTrainPlan which we simply ignore)

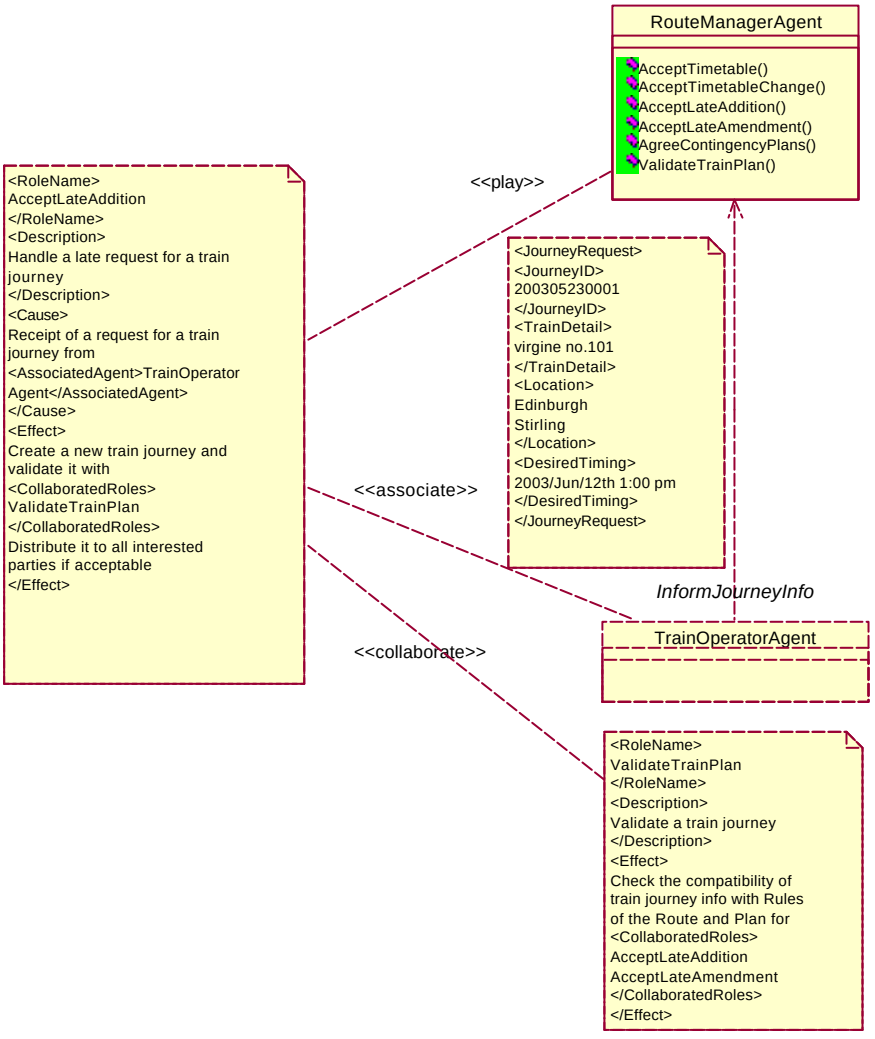


Figure 9. A more complex diagram with two agents, two roles and a message involved

When TrainOperatorAgent sends a certain format of JourneyRequest to RouteManagerAgent, it causes the later agent to play its role AcceptLateAddition, with the help of this role's collaboration partner ValidateTrainPlan it will finally accomplish certain goals.

6. Define ***interaction messages*** between agents

We can define semantic format for agent interaction messages; in this case, RouteManagerAgent can only understand certain format of JourneyRequest from TrainOperatorAgent. If the message it receives is not satisfactory, it deduces that the message is corrupted and simply abandons it or requests a replica from the source. The message is in XML so that it can parse and extract desired pieces of information with ease.

Through the description in the "Cause" block from the original document, we know the message transmitted from TrainOperatorAgent to RouteManagerAgent is to be like this:

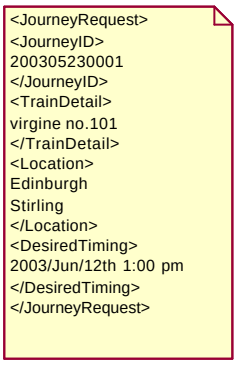


Figure 10. A message element

The XML format message definition gives a specification for the structure of potential objects transmitted between agents during design phase when agent-level infrastructure diagram is detailed to object-level infrastructure diagram.

5 Support Tool Development

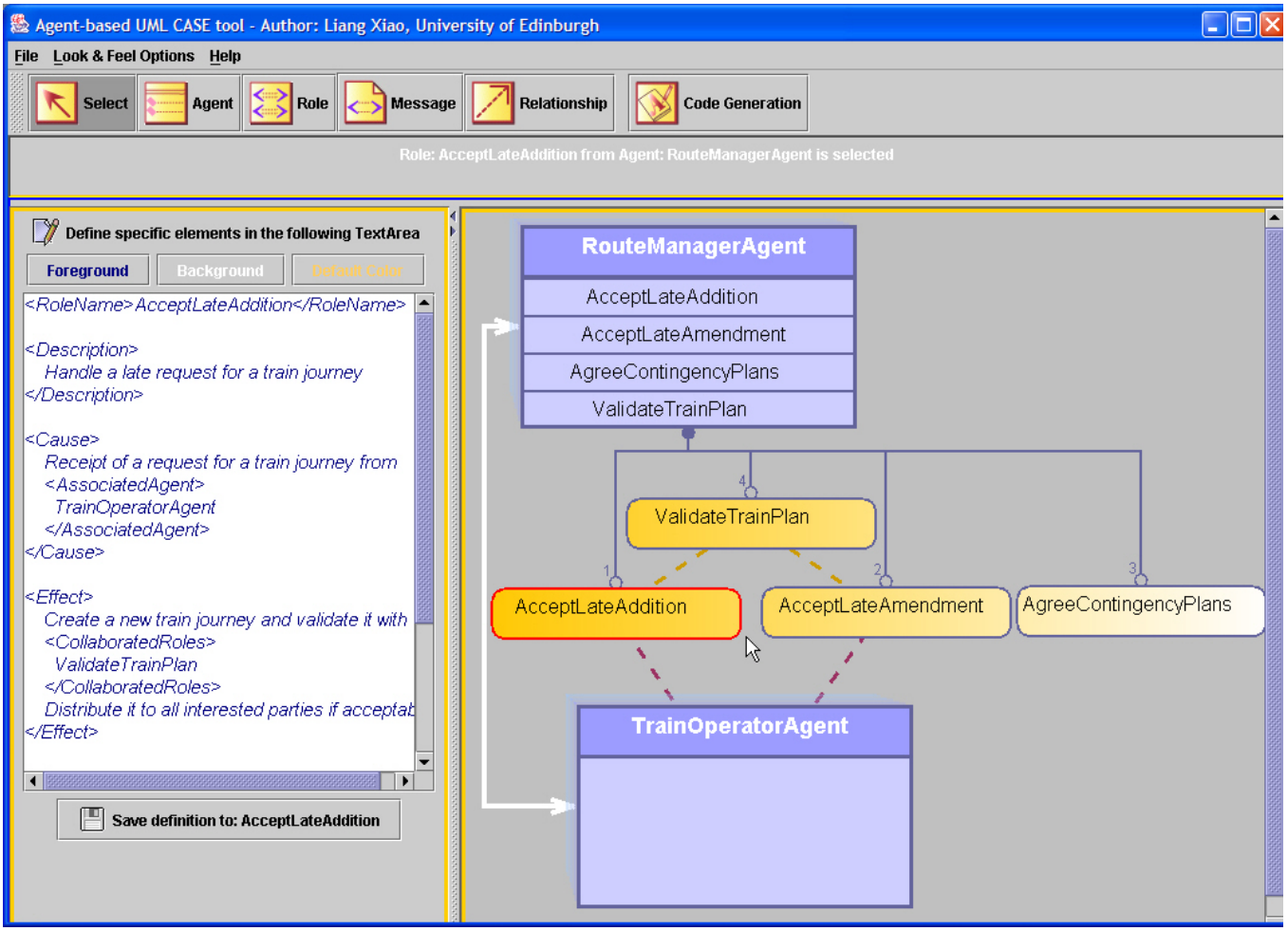
In the previous section, we illustrate our ideas of the requirements transformation with diagrams drawn by Rational Rose. However this tool has no idea of "association" or "collaboration" relationships between agents and roles, not to mention the element of interaction messages. However this UML CASE tool does have the capability to capture the architecture of a component system and show it in a desired style, so we are inspired to develop a similar tool and add agent/role/message concepts to it to represent requirements. The advantage of designing our own tool is that it will suit our specially designated task of requirements representation. Furthermore, we could enhance our tool to make it able to generate framework code from drawn diagrams and possibly validate implemented system by checking known relationships between components. The Software Engineering process could be made much easier if we adopt the following proposed procedure:

1. Understand original requirements document.
   (Described in Section4)
2. Figure out what agents and roles are needed; draw diagrams to represent requirements in our tool; give their definitions in XML according to their original descriptions. (Section4 & Section5.1)
3. Generate source code by the internal functionality of this tool.
   (Section5.1)
4. Implement the system from the automatically generated framework.
   (Section5.2)
5. Validate the complete implementation with the tool. (Section5.3)

5.1 A CASE Tool for Generating Source Code

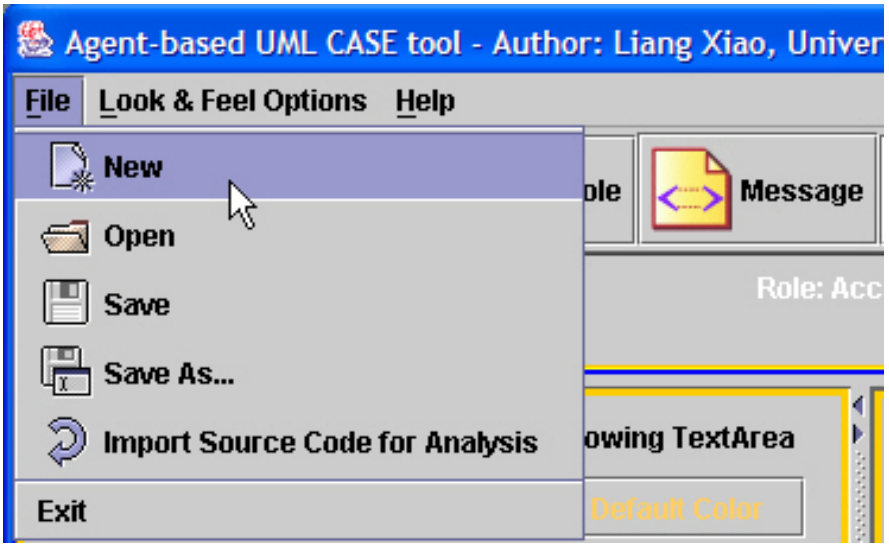
In this section, we will apply our method to a tiny part of the Rail Track document and give details of how to do this with our tool step by step. The suggested steps conform to the agent-oriented requirements diagram building process described in Page16. The diagram we finally get is the counterpart of Figure9 drawn by Rational Rose.

Overview of the tool and the achieved diagram:

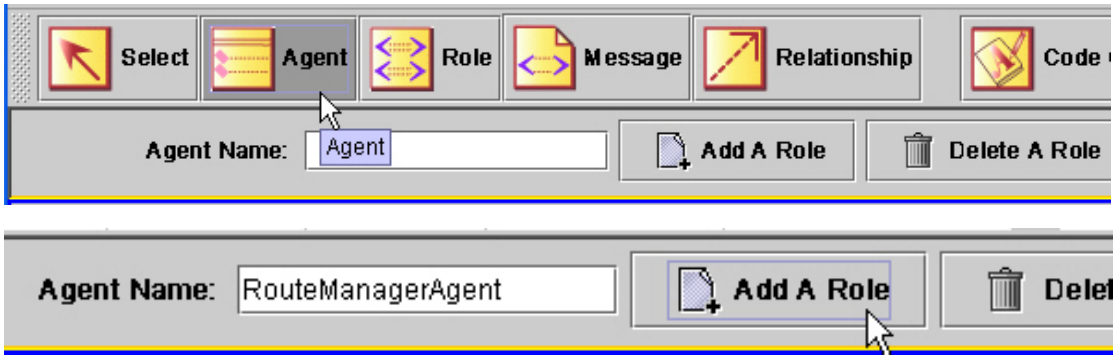


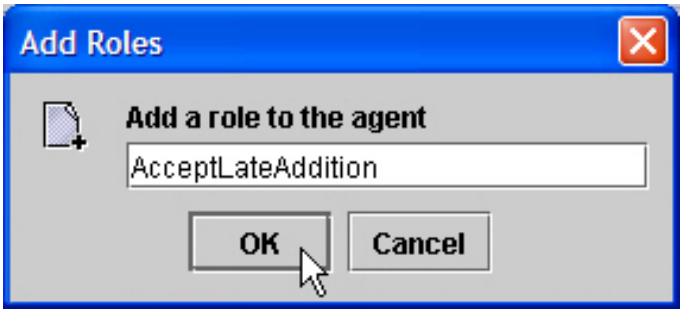
Step1: Define agents and the roles played by them.

1.1 Begin to create a new diagram:

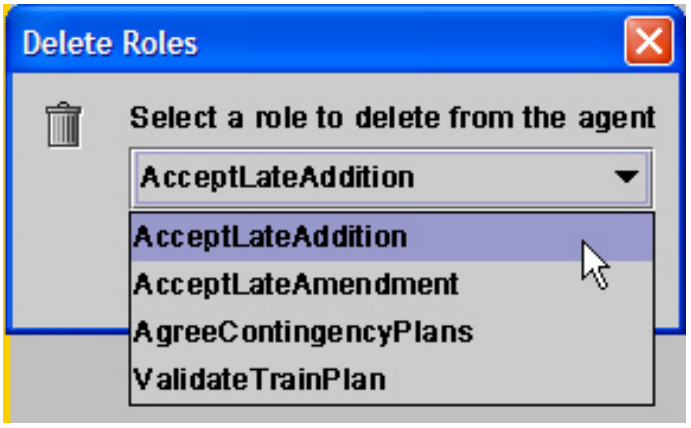


1.2 Input an agent name ‘RouteManagerAgent’ and add roles ‘AcceptLateAddition’, ‘AcceptLateAmendment’ etc. that belong to it one by one:

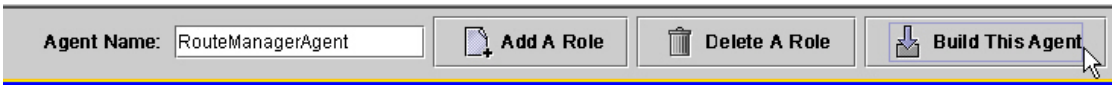




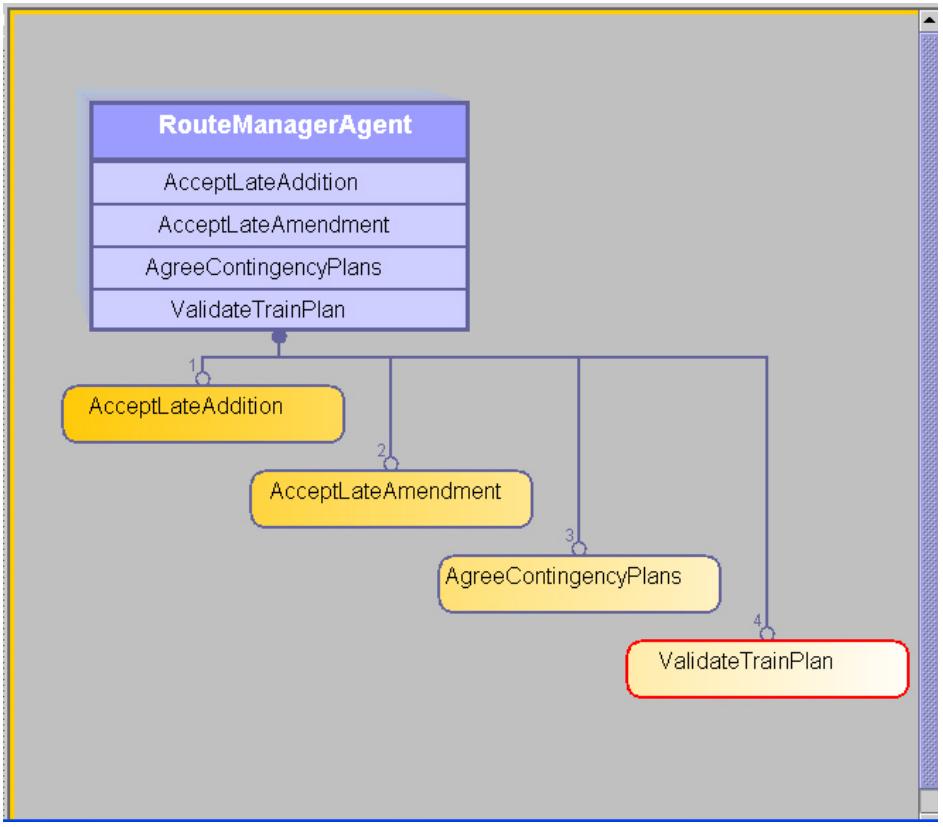
1.3 Delete undesired roles from the added roles list:



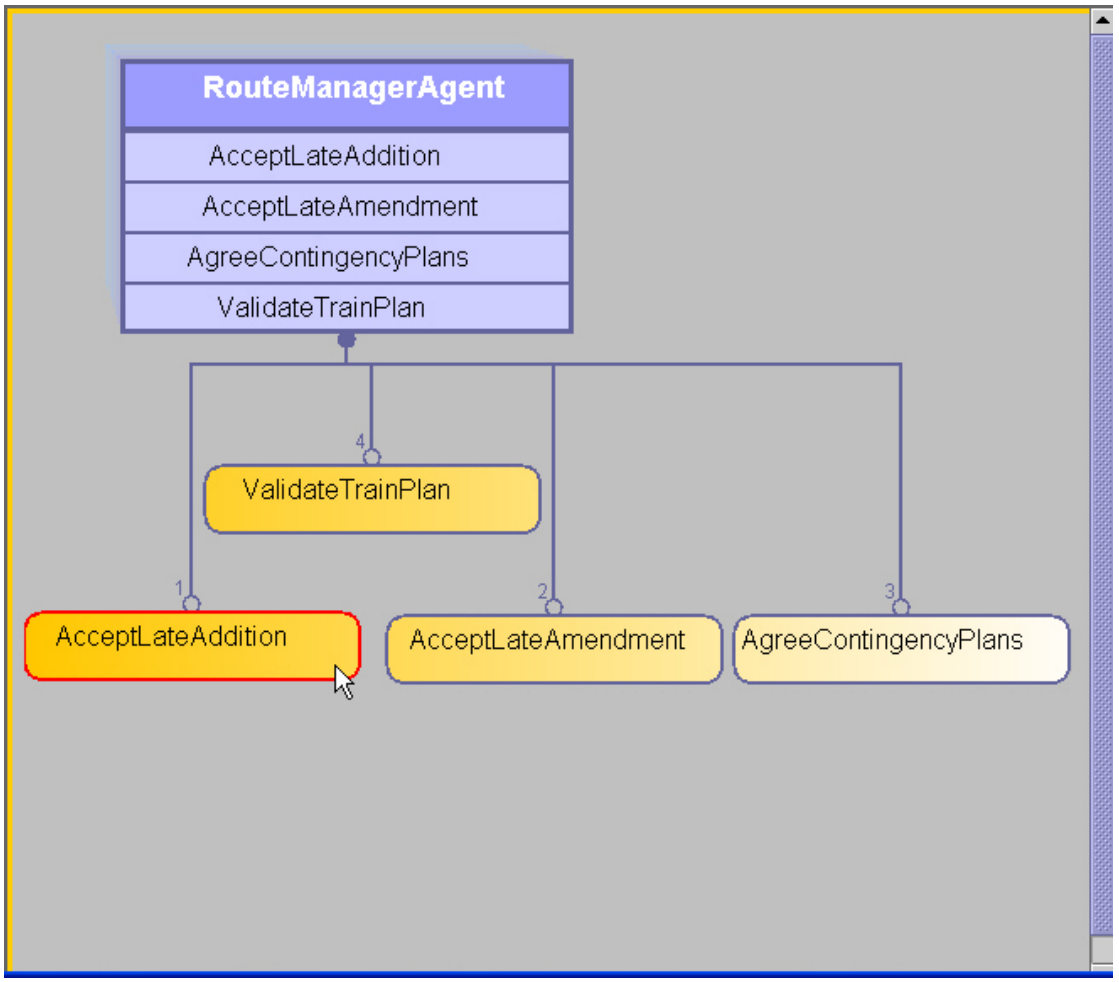
1.4 Build this agent:



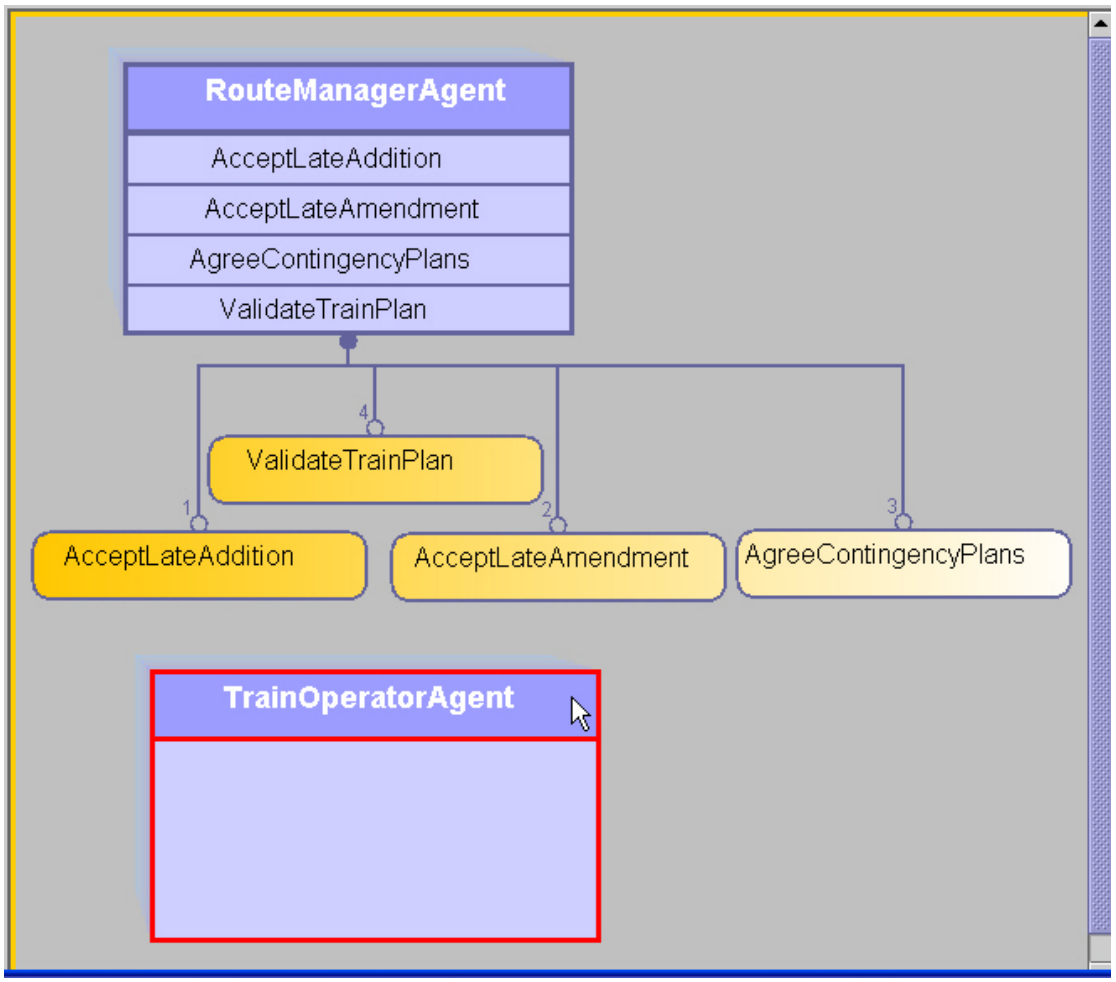
1.5 The agent ‘RouteManagerAgent’ and its four roles are created in the drawing panel, with a shadowed rectangle representing the agent, round-corner-rectangles representing roles played by this agent. There is a connection between the agent and each role it plays, with a filled circle at the agent end and an unfilled circle at the role end:



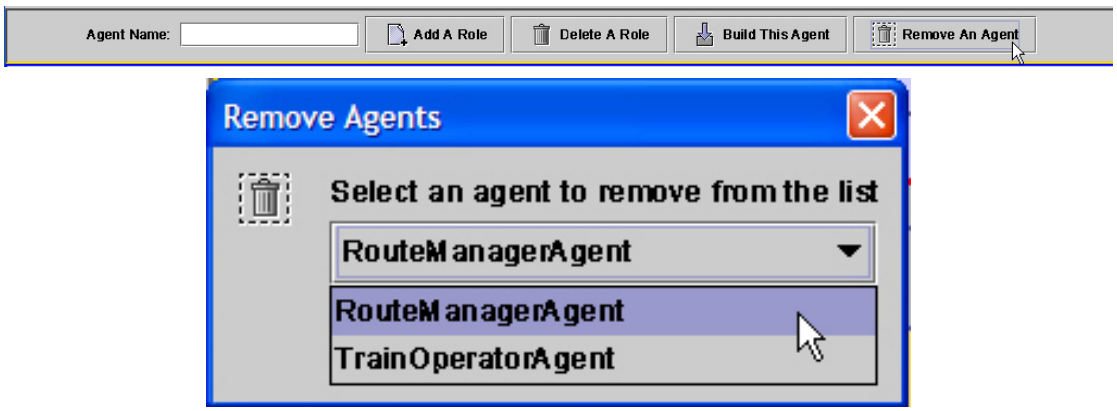
1.6 Adjust agents and roles to the desired locations by dragging and dropping. In this process, the selected component of role ‘AcceptLateAddition’ is highlighted in red:



1.7 Keep running the above processes until every agent and their roles are added to the diagram. We simply add the agent ‘TrainOperatorAgent’ without roles in this step (we are not concerned about roles of this agent):

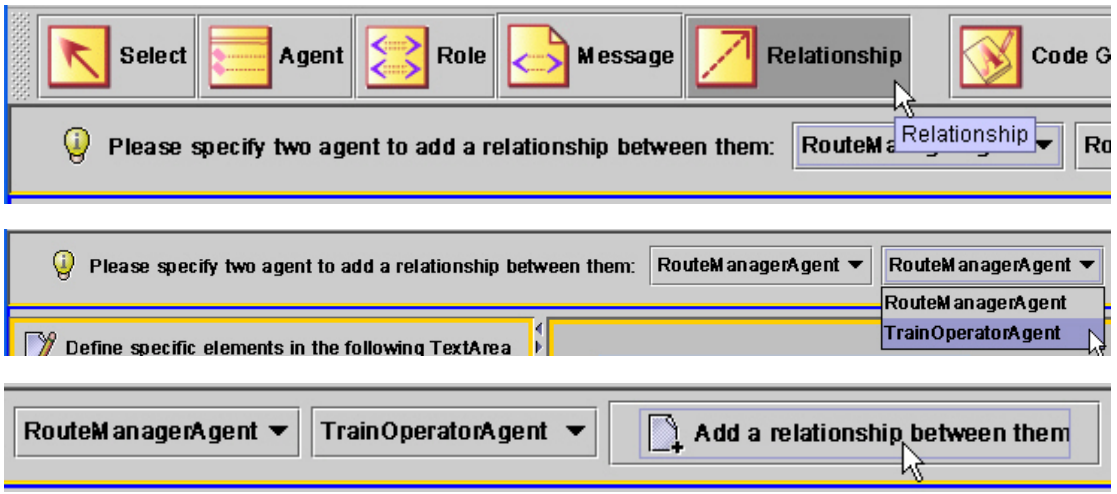


1.8 Remove agents if unnecessary:

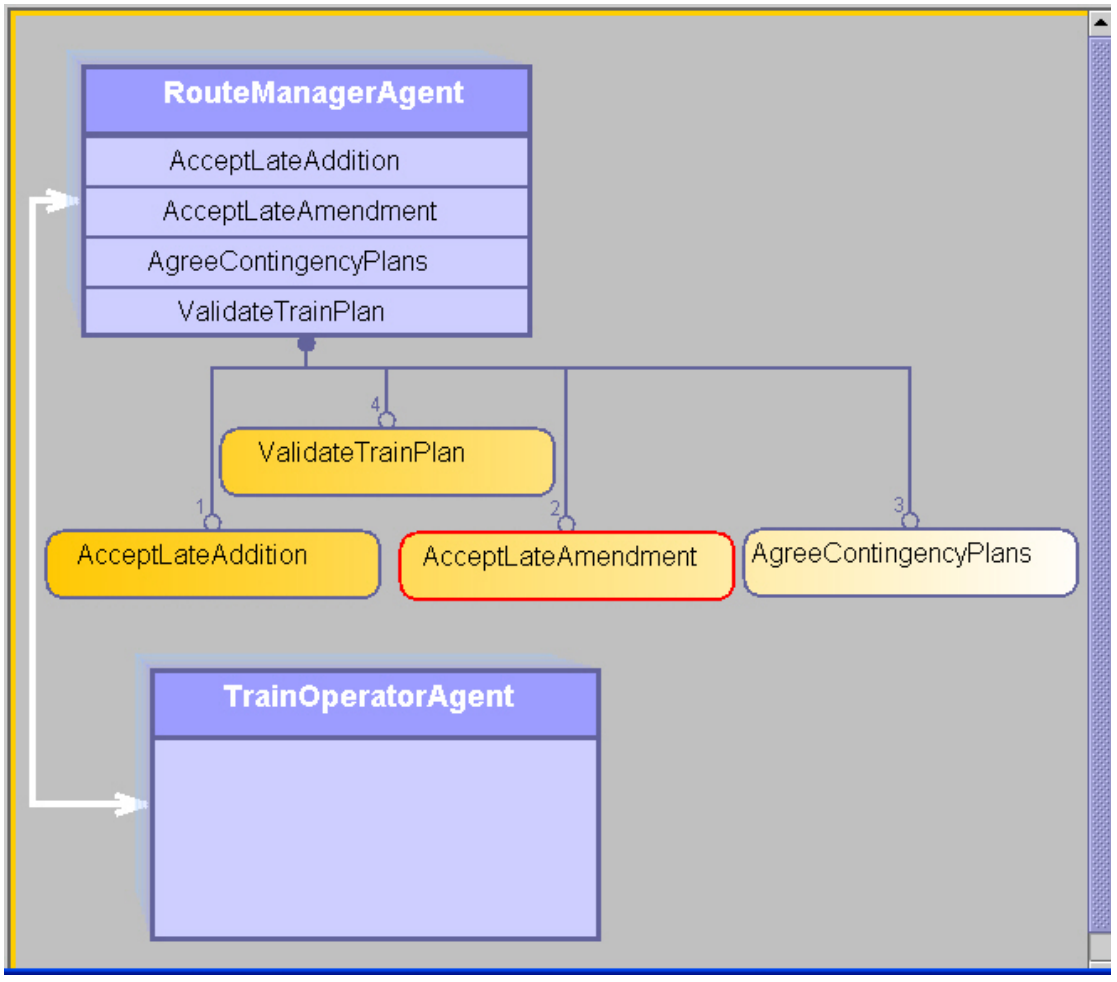


Step2: Establish relationships between related agents, and then give the definition of interaction messages between them.

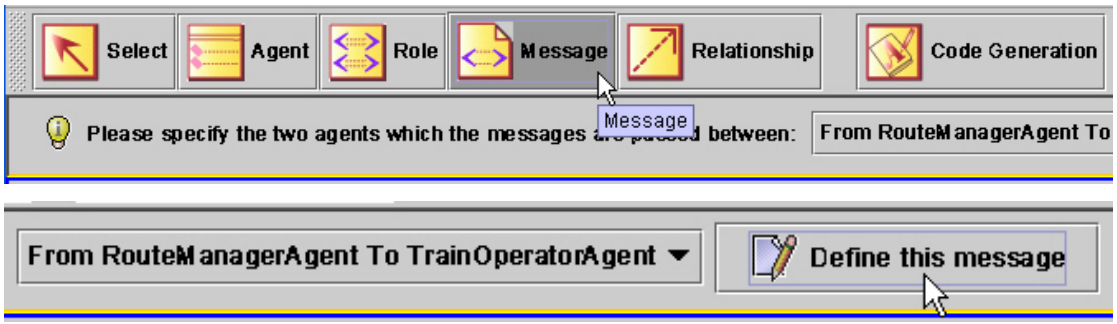
2.1 Connect related agents. 'RouteManagerAgent' and 'TrainOperatorAgent' are related agents so we add a relationship between them:



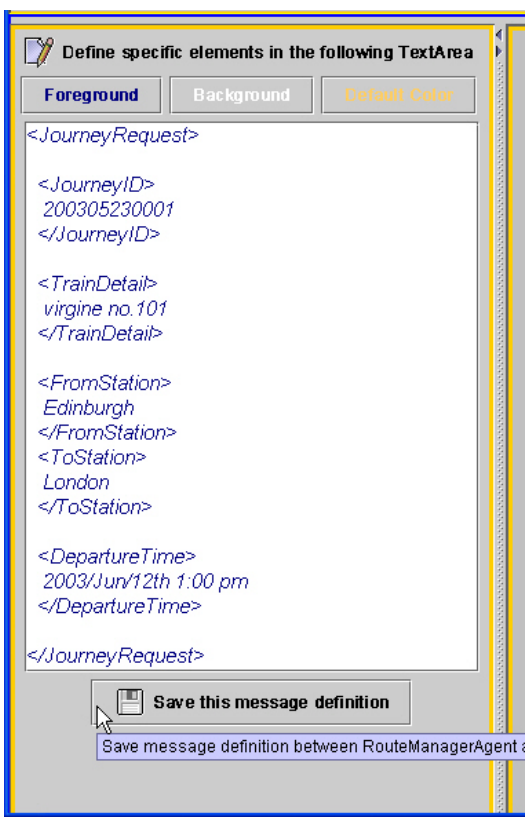
2.2 A white thick line connecting related agents is drawn, with an arrow at each end:

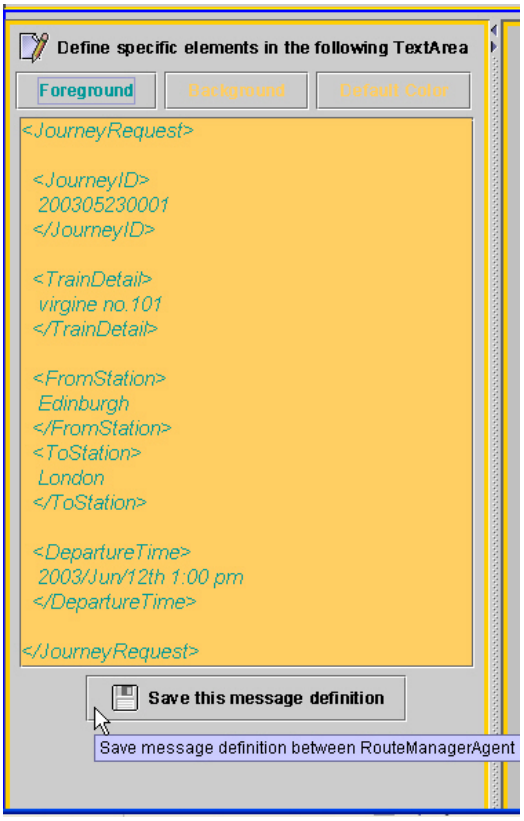


2.3 Specify the message to be defined (from which agent to which other, only between related agents) and give the definition. The message passing between the related agents of ‘RouteManagerAgent’ and ‘TrainOperatorAgent’ is to be defined in this case. The communicative act is supposed to take place on the directed line between these two agents. For sake of conciseness and generality, the message content is not included in the diagram, but can be (re-)defined through “Define this message” button.



2.4 Give the definition in the text-edit area on left, save the message definition. The XML definition of the message passing between ‘RouteManagerAgent’ and ‘TrainOperatorAgent’ is given below:



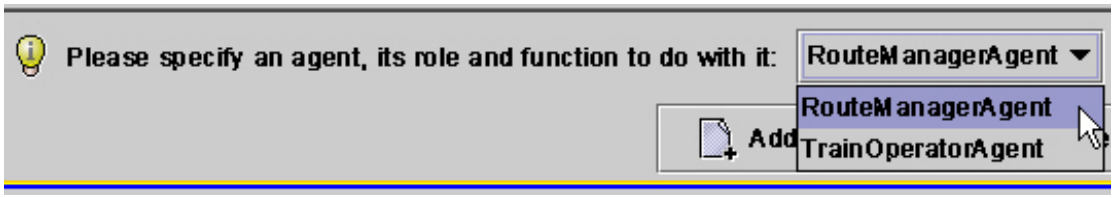


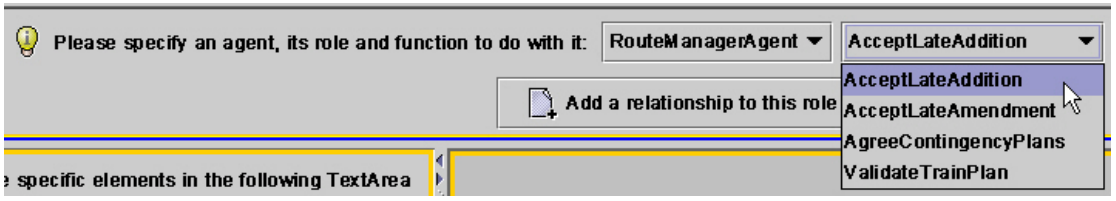
2.5 It is possible to choose comfortable foreground color for text and background color for the text-edit area as above:

Step3: Establish association relationships between roles and agents which cause them to be played, and collaboration relationships between interrelated roles that one role is aided by others.

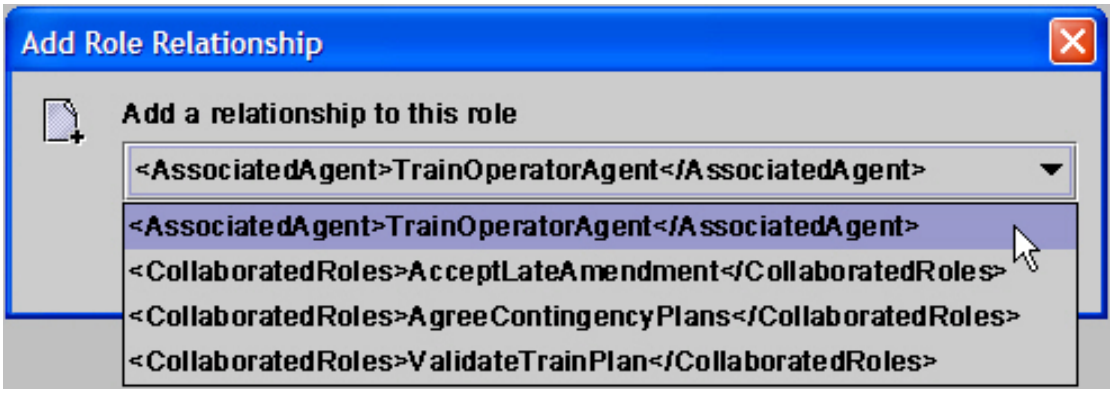
3.1 Specify an agent and a role of it to add a relationship. In this case there are:
A. Two AssociatedAgent relationships:
A.1 Role ‘AcceptLateAddition’ from agent ‘RouteManagerAgent’ with agent ‘TrainOperatorAgent’
A.2 Role ‘AcceptLateAmendment’ from agent ‘RouteManagerAgent’ with agent ‘TrainOperatorAgent’
B. Two CollaboratedRoles relationships:
B.1 Role ‘AcceptLateAddition’ with role ‘ValidateTrainPlan’
B.2 Role ‘AcceptLateAmendment’ with role ‘ValidateTrainPlan’

To add relationship A.1 we choose agent ‘RouteManagerAgent’, then choose role ‘AcceptLateAddition’ and finally click “Add a relationship to this role”:

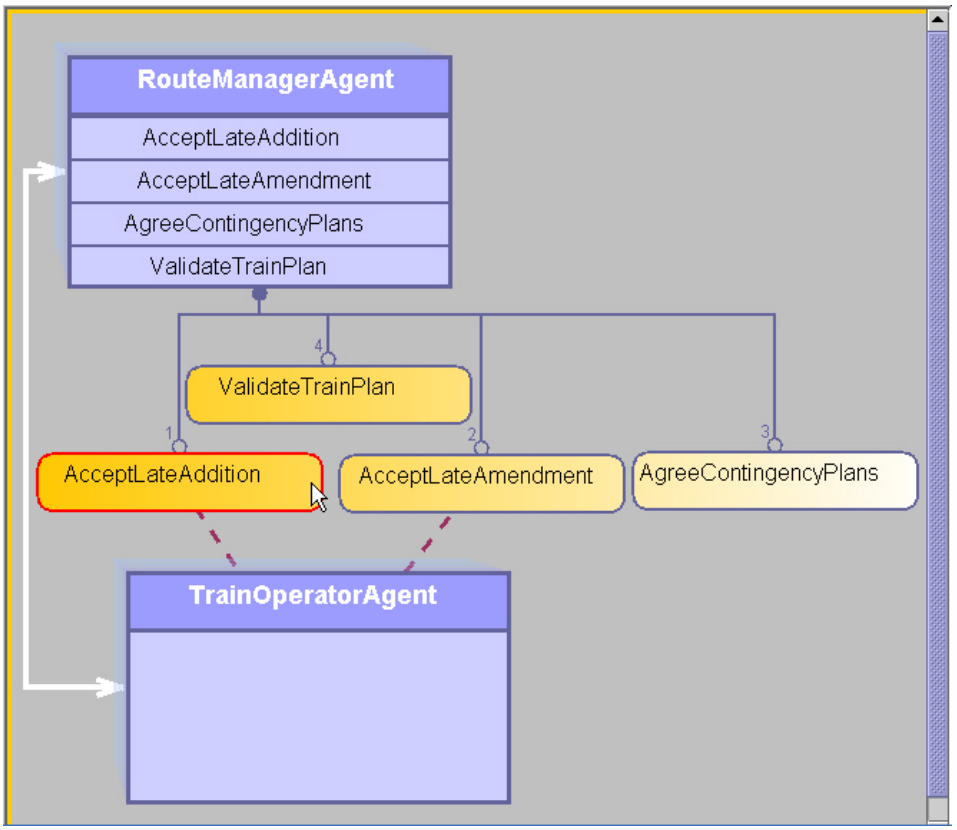




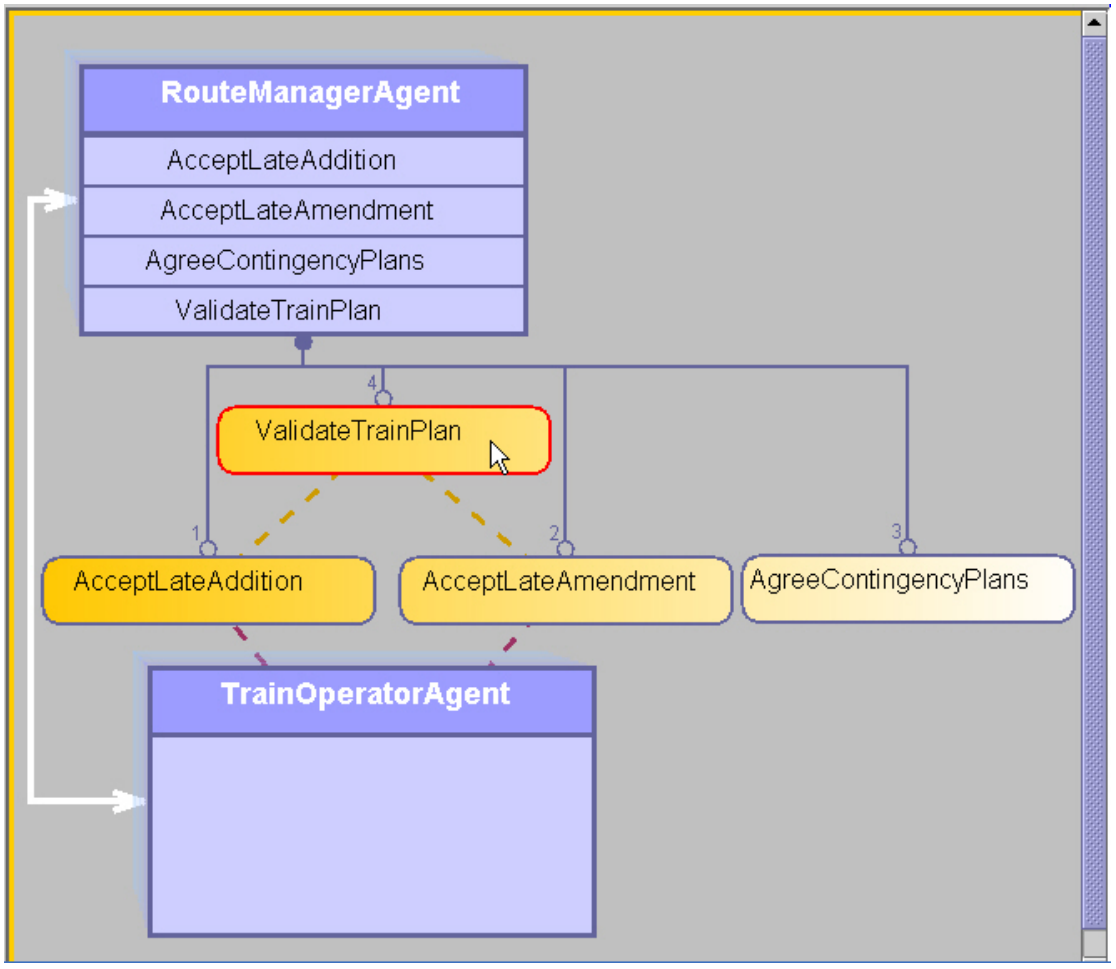
3.2 Once a role from an agent is selected, specify the relationship to be added to this role. There are two categories of possible relationships: AssociatedAgent and CollaboratedRoles. Each agent which relates to this role's host agent will be added to the first category list. Each role which belongs to the same agent as this role does will be added to the second category list. To establish A.1, we choose "<AssociatedAgent>TrainOperatorAgent</AssociatedAgent>" from the possible relationship list to make up an AssociatedAgent relationship:



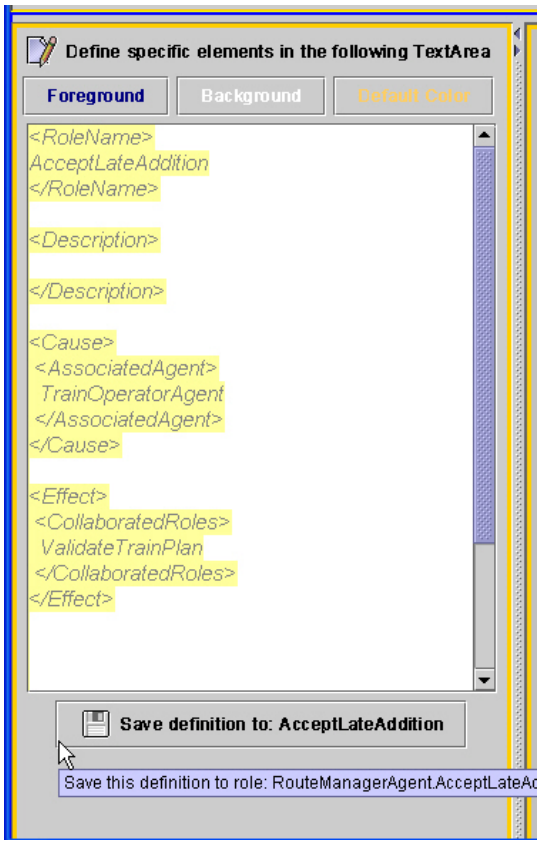
3.3 Add each relationship that belongs to AssociatedAgent category and a dashed purple line will be created connecting the role and its associated agent. A.2 will be constructed in the same fashion in this step:

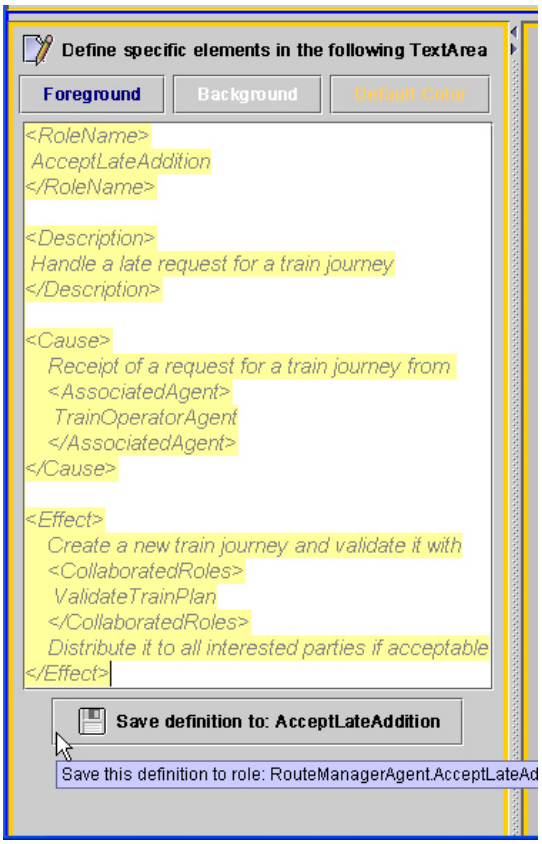


3.4 Add each relationship that belongs to CollaboratedRoles category and a dashed golden line will be created connecting the role and its collaborated agent. Similarly B.1 and B.2 can be made up:



3.5 An initial definition of each role has already been generated by the tool, with the role name, its associated agents and collaborated roles given. In our scenario, role ‘AcceptLateAddition’ from agent ‘RouteManagerAgent’ is associated with the agent ‘TrainOperatorAgent’ and collaborated with the role ‘ValidateTrainPlan’ correctly:





3.6 Complete this role definition and supplement the rest important description from the original requirements and save it.

Once the diagram is finished, the requirements document transformation is completed. We could generate the framework source code now or preferably do so after a refining design step. It is for sure that the former transformation and the later development, which starting from the framework code will be successful only when we have understood the original document and it is also true that we have to make efforts to the later implementation – not everything could be generated automatically.

3.7 Use the Code Generation functionality to generate framework source code from the diagram:

Here is the generated Java code for RouteManagerAgent, we focus on the role AcceptLateAddition and ignore other code pieces.

```
// RouteManagerAgent.java

public class RouteManagerAgent
{
    /*
     * Handle a late request for a train journey
     */
    public AcceptLateAddition()
    {

   // please make a call to TrainOperatorAgent according to the following description:

    /*
     Receipt of a request for a train journey from
     <AssociatedAgent>
       TrainOperatorAgent
     </AssociatedAgent>
     */

    /****************/

   // please implement this method to achieve the following described goals:
   // please make sure to call the internal method ValidateTrainPlan to accomplish this

    /*
     Create a new train journey and validate it with
     <CollaboratedRoles>
       ValidateTrainPlan
     </CollaboratedRoles>
     Distribute it to all interested parties if acceptable
     */

    }
```

```
    public AcceptLateAmendment()
    {

    }

    public AgreeContingencyPlans()
    {

    }

    public ValidateTrainPlan()
    {

    }

}
```

The code generation algorithm is not complex, as readers can expect that from the above generated code. First of all, we create a new directory called "GeneratedJavaCode" under the running system directory; then we create a Java file for each agent and name Java files after their corresponding agents, RouteManagerAgent.java and TrainOperatorAgent.java will be created in our case and each class is defined as public in their Java file in the first place; after that, we allocate methods to these Java classes according to roles allocated to agents, each role from an agent is defined as a public method in a class at the beginning, <RoleName> content from the role definition is used as the method name; finally, method comments, which guide developers to implement the methods are given according to the rest of role definition: <Description> content is used as a simple method usage description comment, it is given just above the method declaration; <Cause> content is used as a comment to instruct developers to invoke methods from other classes (<AssociatedAgent>) to obtain some info; <Effect> content is used as a comment to instruct developers what to do in the method, also point out the possibility to call a set of internal methods (<CollaboratedRoles>) from this same class. In this way, the role definition is divided into four main parts and used for different goals in establishing method declaration and comment body. Semantics from early captured requirements are transferred to the implemented bits in this way and hence the whole system framework is constructed ready for further development.

As we can see, with our notation system for requirements representation and supporting tool, it is really easy to map from the encoded requirements

document to the basic architecture of Java implementation. XML Tags from the agent-oriented UML diagram component definition play a crucial role in this translation process as it captures most semantics relevant information in the system, it acts as a bridge from requirements knowledge capture, design to implementation.

5.2 Adapting Generated Source Code

It is very easy to adapt the above code to the more ideal code as the following as indicated by the comments given along with the code:

```
// RouteManagerAgent.java

public class RouteManagerAgent
{
    public TrainOperatorAgent trainOperator;
    public TrainJourney acceptedTrainJourney;

    public RouteManagerAgent()
    {

    }

    /*
    * Handle a late request for a train journey
    */
    public boolean AcceptLateAddition()
    {
        /*
        * <Cause>Receipt of a request for a train journey from
        * <AssociatedAgent>TrainOperatorAgent</AssociatedAgent>
        * </Cause>
        */

       TrainJourney trainJourney = trainOperator.receiveJourneyInfo();

        /*
        * <Effect>Create a new train journey and validate it with
        * <CollaboratedRoles>ValidateTrainPlan</CollaboratedRoles>
        * Distribute it to all interested parties if acceptable
        * </Effect>
        */
```

```
            if(ValidateTrainPlan(trainJourney))
            {
                acceptedTrainJourney = trainJourney;
                return true;
            }

            else
            {
                return false;
            }

        }

        public void AcceptLateAmendment()
        {

        }

        public void AgreeContingencyPlans()
        {

        }

        public boolean ValidateTrainPlan(TrainJourney trainJourney)
        {
            // some checking functions go here.
        }
}
```

As to each message passed between agents like the following:

```
<JourneyRequest>

    <JourneyID>
     200305230001
    </JourneyID>

    <TrainDetail>
     virgine no.101
    </TrainDetail>

    <FromStation>
     Edinburgh
```

```
    </FromStation>

    <ToStation>
     London
    </ToStation>

    <DepartureTime>
     2003/Jun/12th 1:00 pm
    </DepartureTime>

</JourneyRequest>
```

They can be defined as XML streams and validated by XML parser. Alternatively they can be defined as normal Java classes with attributes extracted from XML tags and no methods, as the way we adopt practically:

```
// TrainJourney.java

import java.lang.*;
import java.sql.Timestamp;

public class TrainJourney
{
   int JourneyID;
   String TrainDetail;
   String FromStation;
   String ToStation;
   Timestamp DepartureTime;

   public TrainJourney(int journeyID, String trainDetail, String fromStation, String toStation,
Timestamp departureTime)
   {
        this.JourneyID = journeyID;
        this.TrainDetail = trainDetail;
        this.FromStation = fromStation;
        this.ToStation = toStation;
        this.DepartureTime = departureTime;
   }
}
```

Since the role *AcceptLateAddition* from the agent *RouteManagerAgent* has *TrainOperatorAgent* as its AssociatedAgent, an instance of *TrainJourney* will be passed to *RouteManagerAgent* by a call to *TrainOperatorAgent*:

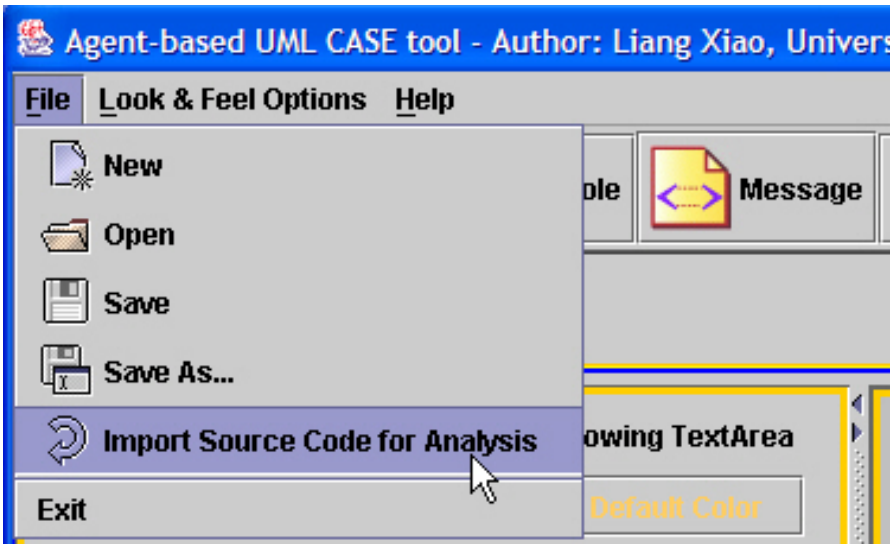
TrainJourney trainJourney = trainOperator.receiveJourneyInfo();

In this way, we relate agent classes and message classes; let objects of later classes be passed to objects of former classes, so that agents are enabled to use messages. Our requirements modeling methods guide designers not only on the design of agent classes but also on message classes.

With our specially designed tool, we can save a drawn diagram and load a previously saved one, make changes to it according to the changes to the requirements and regenerate the code to conform to the new system architecture.

5.3 Validating Adapted Code against the Original Model

In addition, with the aid of this tool, we may be able to check the consistency validity of developers' code by importing and analyzing them. Since relationships between components are given in the diagram, the same relationships are supposed to be reflected as we analyze the code.



We integrate the code analysis functionality with this tool to make sure the correct relationships are established in the finally implemented code. With "Import Source Code for Analysis" under File menu we create report files in the tool running folder. If we generate the code immediately after the diagram is drawn and leave the code untouched like those in Page 38-39, the following is the created report:

****

This report is created on Sat Aug 09 12:12:31 BST 2003 with
Agent-based UML CASE tool - Author: Liang Xiao, University of Edinburgh
****

***** Start analyzing Source Code... *****

Reading RouteManagerAgent.java ...
Reading AcceptLateAddition from RouteManagerAgent.java ...
!!! There is a problem with AcceptLateAddition. It is supposed to be associated with TrainOperatorAgent !!!
!!! There is a problem with AcceptLateAddition. It is supposed to be collaborated with ValidateTrainPlan !!!
Reading AcceptLateAmendment from RouteManagerAgent.java ...
!!! There is a problem with AcceptLateAmendment. It is supposed to be associated with TrainOperatorAgent !!!
!!! There is a problem with AcceptLateAmendment. It is supposed to be collaborated with ValidateTrainPlan !!!
Reading AgreeContingencyPlans from RouteManagerAgent.java ...
--- Relationships in AgreeContingencyPlans from RouteManagerAgent.java are OK! ---
Reading ValidateTrainPlan from RouteManagerAgent.java ...
--- Relationships in ValidateTrainPlan from RouteManagerAgent.java are OK! ---
*** Start logging lost roles for RouteManagerAgent.java ***
--- No lost roles for RouteManagerAgent.java ---
*** Finish logging lost roles for RouteManagerAgent.java ***

Reading TrainOperatorAgent.java ...
*** Start logging lost roles for TrainOperatorAgent.java ***
--- No lost roles for TrainOperatorAgent.java ---
*** Finish logging lost roles for TrainOperatorAgent.java ***

*** Start logging lost agents ***
--- No lost agents ---
*** Finish logging lost agents ***

***** Finish analyzing Source Code... *****

After a full implementation of the method AcceptLateAddition and the code is like those in Page 40-41, we get the report like this:

****

This report is created on Sat Aug 09 12:37:50 BST 2003 with
Agent-based UML CASE tool - Author: Liang Xiao, University of Edinburgh
****

****** Start analyzing Source Code... ******

Reading RouteManagerAgent.java ...
Reading AcceptLateAddition from RouteManagerAgent.java ...
AcceptLateAddition is associated with TrainOperatorAgent correctly
AcceptLateAddition is collaborated with ValidateTrainPlan correctly
--- Relationships in AcceptLateAddition from RouteManagerAgent.java are OK! ---
Reading AcceptLateAmendment from RouteManagerAgent.java ...
!!! There is a problem with AcceptLateAmendment. It is supposed to be associated with TrainOperatorAgent !!!
!!! There is a problem with AcceptLateAmendment. It is supposed to be collaborated with ValidateTrainPlan !!!
Reading AgreeContingencyPlans from RouteManagerAgent.java ...
--- Relationships in AgreeContingencyPlans from RouteManagerAgent.java are OK! ---
Reading ValidateTrainPlan from RouteManagerAgent.java ...
--- Relationships in ValidateTrainPlan from RouteManagerAgent.java are OK! ---
**** Start logging lost roles for RouteManagerAgent.java ****
--- No lost roles for RouteManagerAgent.java ---
**** Finish logging lost roles for RouteManagerAgent.java ****

Reading TrainOperatorAgent.java ...
**** Start logging lost roles for TrainOperatorAgent.java ****
--- No lost roles for TrainOperatorAgent.java ---
**** Finish logging lost roles for TrainOperatorAgent.java ****

**** Start logging lost agents ****
--- No lost agents ---
**** Finish logging lost agents ****

****** Finish analyzing Source Code... ******

Our supporting tool validates code and makes reports in the following way: Firstly the tool creates a new report file under the running system directory with its name ending with a timestamp indicating the code analysis time. Secondly, the tool reads each Java source file under a specified directory and analyzes each class to see whether there exist in each of their method expected references corresponding to role relationships established in the modeled diagram. Any association or collaboration miss, that is, a reference of a role/method's associated agent/class instance or collaborated role/method does not appear in its implemented body, will be reported as a problem of the method. For example, assumed a role is associated with an agent in the modeling diagram, an instance of that agent's implemented

class is supposed to appear in the definition of this role's implemented method. If both of these relationships are correct, a proper relationship establishment announcement will be made. As each method analysis is finished, it goes to the next one, until every method of one class is analyzed, then it goes to the next class. Those methods which have no counterpart roles in the modeling diagram are ignored as they may be assisting functions in classes. Those classes which have no counterpart agents in the modeling diagram are also ignored as they may be assisting components in the system and designed in system design phase. However, extra agent/role without their corresponding class/method would cause fatal errors; this is what the tool validates in the final step. Finally, each role of agents which is not implemented as a method of classes will be reported and each agent which is not implemented as a class will be reported, as the system will be incomplete and does not work with the deficiency of them.

The relationships in AcceptLateAddition are analyzed as correct in the above report once we associate this role with the agent TrainOperatorAgent and collaborate it with the role ValidateTrainPlan. Therefore manually implemented system architecture could be validated of consistency in the final step to remind developers of any incompletely implemented portion according to the requirements specification.

6 Evaluation of the Requirements Translation

Agent-oriented modelling can help design substantially, especially when the designed Software System is complicated and distributed. Although the requirements translation is not essential in case the system is not complex enough, it can still act as a bridge to a design diagram. The translated requirements do not include some domain knowledge. It can capture Functional Requirements which set out services the system should provide; but Non-Functional Requirements, which constrain the system being developed or the development process are hard to be modelled. Apart from this, most of the domain knowledge can be reflected and fit well in the translated requirements; a function described in the table shown in Figure7 is matched perfectly well with a role element which is correlated with other elements in the agent-oriented diagram, reflecting its function definition semantics. The requirements translation brings us benefits in several Software Engineering aspects. Currently there is no quantitative experimental data that shows, on a standard set of software, the superiority of this agent-oriented modelling approach over other alternative techniques. In fact, such data does not exist even to generic agent-oriented Software Engineering approaches. Hence arguments are qualitative in nature.

Original requirements is admittedly essential in any case, it is inevitably elicited through such techniques like interview, questionnaire, prototyping, ethnographic technique, etc. in the first place and act as a fundamental documentation for the agent-oriented modelling. However this natural language specification is not enough. As we have introduced in Section2.3, the graphics-based diagram representation is capable to captures structure and some semantics. It is another important viewpoint along with other potential useful viewpoints like these towards modelling stakeholders' goals or scenarios that illustrate how goals are achieved. Multiple viewpoints can provide us multiple perspectives views and complete recognition of the requirements; they are complementary to one another. This semi-formal specification language may also act as a bridge for representation format transits from natural language to the future formal language.

Diagram notations can help communication among every participant in terms of its visualization of the specification language and furthermore makes the requirements measurable, development easier to control and whole engineering progress risk reduced. Moreover, convenient and better communicated requirements in turn accelerate the agreement of the final version of the requirements and guarantee the accurate recognition of it. There are many facts and cases like the following prove that whether requirements can be agreed to could determine the fate of projects. This is the reason we argue the adoption of the agent-oriented modelling.

Fact [39]:
Wastage on failed projects
E.g. 1997 GAO report: $145 billion over 6 years

Re-work from defect removal
E.g. Motorola: 60-80% of software budget (was) spent on re-work

Case:
Customer Database System [40]
In 1996 a US consumer group embarked on an 18-month, $1million project to replace its customer database. The new system was delivered on time but didn't work as promised, handing routine transactions smoothly but tripping over more complex ones.
Within three weeks the database was shot down, transactions were processed by hand and a new team was brought in to rebuild the system.
Problems:
The design team was over-optimistic in agreeing to requirements.

In addition, as requirements evolve, the agent-oriented modelling assists the

locating of the changed bit of the requirements and reduces the occurrences of the inconsistency made during the evolvement. There is evidence to prove the above point, for example, when a system function (before translated into a role) is proposed to be replaced by another one, there are at least two issues have to be considered before the replacement is made (after the translation): Whether the delivered request from which agent this role is associated with can be satisfied by the new role, or alternatively whether it is no longer necessary during the requirements evolving; Whether the role which is collaborated with this role can accomplish the task with the aid of the new role, or alternatively whether this task is no longer necessary during the requirements evolving. In other words, the relationship established in the diagram can be used to check the consistency of the new requirements and hence help to establish the new diagram. In the other hand, we are likely to omit such checks during the requirements amendment with descriptive requirements representation as these relationships are more implicitly expressed.

Agent-oriented modelling is quite useful for the management of changing requirements during requirements evolution. As long as we are able to identify the mutable requirements, we may delegate an agent or a group of agents exclusively to deal with each one of these and package immutable ones independently as stable agents. So we do not need to touch most of the core functionalities when we make changes to the requirements. We achieve greater efficiency as we have relatively fewer agents to trace and for version control due to fewer agents are affected during the requirements evolution. Alternatively, we may also delegate an agent manager in the whole system or in each agent group assumed the system is divided into several groups, making it act as a high level agent to propagate changes through messages passing between it and other agents. In this way, requirements changes are distributed through central manager agents. With either of the above two approaches, requirements are more configurable and easier to manage; less risk will be introduced and less harmful impact will be brought as requirements are evolving.

Agent-oriented modelling is also beneficial in terms of prioritising requirements. It will not cost us unnecessarily if vague and unimportant parts of requirements are not to be processed in high priority. We can delegate an agent with only one role to represent each of these portions, just give the known functionalities to the definition of the roles and relate these agents to outside world roughly. We can make rectification when we get more clear recognition of the whole system requirements. In this case, an agent may be developed into a group of agents; a single role in each agent may be refined to different roles representing different functionalities; internal relationships may also be established so that roles can be

collaborated with each other. In this way, we proceed in the absence of some knowledge, and it will enable us to get fast and valuable feedback from the early delivery, adjust our previous requirements and discover what we do not know. By modelling requirements as agent-oriented UML diagrams, it is convenient to abstract unclear part and identify them later; hence it is especially useful for Incremental Development Processes.

This benefit becomes more apparent if we adopt the Twin Peaks Model [41] proposed by Bashar Nuseibeh [20]. The idea is early understanding and construction of the software architecture to provide a basis for discovering further requirements and constraints. The author argues that start a Software System from either requirements or architectures would invariably results in a production of artificially frozen requirements documents for use in the next step in the development life cycle or the creation of a system with constrained architectures that restrict users and handicap developers by resisting inevitable and desirable changes in requirements. Achieving a separation of requirements and design is also often difficult, candidate architectures can constrain designers from meeting particular requirements, and the choice of requirements can influence the architecture that designers select or develop. By providing an incremental development process, spiral life-cycle model addresses many such drawbacks as developers repeatedly evaluate changing project risks to manage unstable requirements. An even finer-grain spiral life cycle may develop software architectures that are stable, yet adaptable in the presence of changing requirements by interleaving the development of requirements and its architecture concurrently.

The Twin Peaks model is an adaptation of the spiral life-cycle model, it addresses requirements specification and design issues simultaneously and produces progressively more detailed requirements and design specifications, as suggested in the following figure.

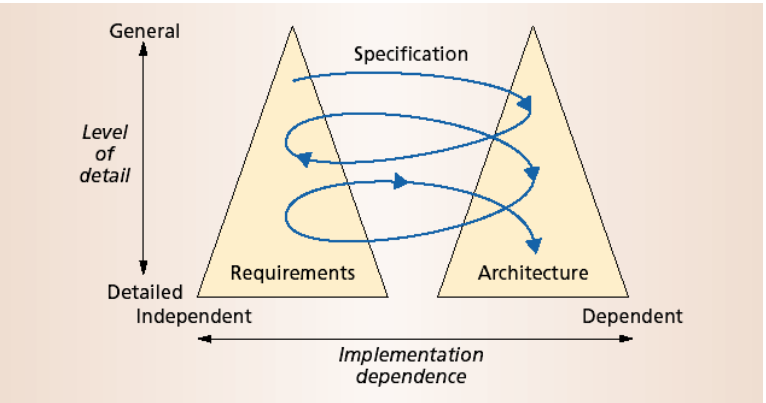


Figure 11. Twin Peaks model

We argue that with the integration of agent-oriented requirements approach and agent-oriented architecture style our development will be more stable and flexible while requirements are mutable. On completion of the requirements modelling, an agent with its roles in requirements representation can be developed into an agent or a group of agents as autonomous software units providing services in design; a relationship between agents in requirements is represented as a communication channel through which these software units cooperate and pass messages to achieve a certain goal in design; an internal relationship inside an agent in requirements denotes that there is an internal invocation in such a software unit in design. Since each agent in the architecture has a corresponding agent in requirements, under the model of Twin Peak, as we use agent-oriented requirements modelling and architecture style together, it costs much less effort when we develop them concurrently, less risky when requirements change occurs and more easy to maintain both of them. In other words, requirements and design processes during the Software Development are integrated seamlessly and their architecture fits each other.

By prioritising requirements, making as simple a design as possible, delivering a running system with most essential agents and eventually getting fast feedback we adopt the methodology of Kent Beck's Extreme Programming (XP). Twin Peaks also shares much in common with XP, such as the goal of exploring implementation possibilities early and iteratively. XP focuses on producing code—sometimes at the expense of the wider picture of requirements and architecture, hence it is not scalable. Agent-oriented requirements and design approaches together with the Twin Peaks model are complementary to XP and solve XP's lack of scalability in that they are inherently iterative, supply blueprints of the system, achieve modularity by the use of agents and enabled to adopt tested components derived from well understood architecture, which can facilitate incremental development of large-scale systems. As a result, we propose an overall Software Development Process driven by the development of an agent-oriented requirements document along with an agent-oriented design architecture simultaneously under the Twin Peaks model and producing code with the XP methodology, the process being iterative. The integration of these techniques thus brings us enhanced traceability as the requirements are linked to both design and implementation closely. Although the *agent* referred in this paper is not the same as that referred by Michael Wooldridge [5] and Nick Jennings [9], they are connected by this integration, and an *agent* is understood in three ways as we consider it in the field of RE as a unit to carry domain knowledge; in the field of design architecture as an running software; in the field of implementation as a class. The aim of this technique integration is to boost rapid Software Development with toleration

of changing requirements, brings high productivity, and help to accomplish a high quality system within tight time and constraint budget.

7 Conclusion and Potential Improvement

By assigning logically related tasks to the component of agents and enable agents, roles and messages passing between agents to carry requirements knowledge we transform the descriptive requirements to the agent-oriented UML diagram; requirements knowledge described in natural language is captured and represented as UML components interconnected with each other, along with their XML definitions constructed in the diagram. The transformed document presents domain knowledge more visibly than the traditional representation style and makes it easier for developers to grasp the whole system structure. However it is not sufficient to be a thorough requirements document on its own as some conceptual information are not transformed, additional efforts have to be made to deal with complex work flow process during the design phase and this can not be done without a complete recognition of the original document, implicit knowledge have to be extracted from it as a supplement; nevertheless, the translated document can help requirements analysis and accelerate a fully understanding of it.

For example, according to the documentation of role AcceptLateAddition, which is from the original requirements document and shown in Figure7, the section “Outputs” describes as, after an invocation of this role, a new train journey may be outputted to other parties, this entails us to invoke some agents to update their knowledge; the same situation applies to “Information Used”, we may have to request the latest information from other parties which are not referred to in the original requirements so explicitly. These semantics are hard to reflect in the transformed modeling diagram but if ignored during design or implementation, fatal errors might occur.

As a result, during the requirements transformation for a certain role, some agents and roles may be involved in the whole process implicitly. We may add to more concepts like “Organization” or “Goal” to tackle such problems so that we may prevent omissions and incomplete modeling. We suppose an Organization is a group of agents that form a subsystem to accomplish certain goals; a Goal is a main responsibility of the system which may be accomplished by a series of roles played one another. In this scheme, agents are organized to control the work flow and achieve each goal; single tasks of each role are not separated but integrated. In addition, implicit relationships may also be established and extra communication channels discovered under

the direction of work flow. A goal-oriented diagram may supplement our agent-role-relationship diagram to capture requirements from another perspective. Dataflow in and out among components may probably be integrated in this diagram to illustrate how a goal is accomplished. In this way, our agent-oriented modeling will be more comprehensive and able to represent the original requirements more precisely.

8 Open issues & Further work

Agents in this project are not granted intelligence so they are not able to behave intelligently to exchange knowledge, adapt themselves to mutable environments and control the system running process. Agents in many research areas are supposed to do so. This is partly because we focus much on the transformation of functions from original requirements to roles and concern too little about the agents and how they cooperate to accomplish goals in this project. I will continue this work during my PhD research on S-PAD [42] Incremental Software Development. In that project, I will take account of the design issue, on how to develop an architecture of intelligent agents from agent-oriented requirements, how they come together to plan the increments during the Incremental Development and how the model can adapt itself to changes after each delivery in practice. In addition, agent-oriented requirements are supposed to be extended to capture more information that can not be modeled by a single agent-role-relationship diagram presented in this paper. More concepts like “Organization” and “Goal” which have already been pointed out in the previous section may be introduced to establish another perspective viewpoint of requirements, new notations and diagrams may be designed, more knowledge and state information may be maintained by agents to enable them to make decisions and cooperate with the external world.

There are some additional considerations about interaction message contents. They may be changed requirements or other conceptual information, to adapt agents/roles to represent updated UML diagrams, this in practice might be done with the help of XMI [43]; an alternative way to do this might be, as proposed, to delegate an agent manager in an organization of agents, to deliver agent messages in XML streams, which may be extracted from a portion of XML documents, or generated dynamically by the agent manager. In this way, as some additional functionalities are required as the new requirements reflect, some new roles which a suitable agent may play can be exported by the agent manager to that agent through notification messages formatted in XML streams. This solution reduces the overhead of altering many agent definitions by assigning the responsibility to a single

manager agent.

Some open issues emerge from the propositions in the above statements: If we have an agent manager in the agent-oriented requirements representation system, there will be inevitably another distinct kind of conceptual message which is to organize and update agent knowledge as requirements change. Another kind of message semantics is needed in this situation. Will it bring confusions if we accept both this conceptual message and the normal agent interaction message in a single requirements diagram? Or alternatively we can build on top of the basic diagram illustrated in this paper a higher-level diagram to deal with those things like incoming new tasks or new subtasks under an existing task assignment. The construction of this diagram is to automate the adaptation of lower-level diagrams. To construct this higher-level diagram we are to connect the agent manager to those mutable agents and they communicate to make the requirements changes deployed in automation. Which is the appropriate scheme to apply deserves further research.

References:

[1] Gerd Wagner
The Agent–Object-Relationship metamodel: towards a unified view of state and behavior
Information Systems 28 (2003) 475–504
http://www.elsevier.com/locate/infosys

[2] Nicholas R. Jennings
Agent-Oriented Software Engineering
Dept. Electronic Engineering, Queen Mary & Westfield College, University of London
http://www.ecs.soton.ac.uk/~nrj/download-files/cairo.pdf

[3] A. Newell
The Knowledge Level Artificial Intelligence 18 87-127

[4] Michael Wooldridge
Department of Computer Science, University of Liverpool
http://www.csc.liv.ac.uk/~mjw/

[5] Michael Wooldridge
Department of Computer Science, University of Liverpool
http://www.csc.liv.ac.uk/~mjw/research/

[6] Lind J.
Issues in Agent-Oriented Software Engineering
The First International Workshop on Agent-Oriented Software Engineering (AOSE-2000), 2000

[7] Amund Tveit
A survey of Agent-Oriented Software Engineering
Department of Computer and Information Science,
Norwegian University of Science and Technology
http://www.jfipa.org/publications/AgentOrientedSoftwareEngineering/

[8] Parunak H. V. D. and Odell J.
Representing Social Structures in UML
In Proc. of the fifth international conference on Autonomous Agents, Forthcoming, 2001

[9] Nick Jennings
Dept of Electronics and Computer Science, University of Southampton
http://www.ecs.soton.ac.uk/~nrj/abse.html

[10] Wooldridge M. J., Jennings N. R. and Kinny D.
The Gaia methodology for agent-oriented analysis and design
Autonomous Agents and Multi-Agent Systems, September 2000

[11] Wooldridge M. J., Jennings N. R. and Kinny D.
A methodology for agent-oriented analysis and design
In Proc. of the third international conference on Autonomous Agents
Pages 69-76, 1999

[12] DeLoach S. A.
Systems Engineering A Methodology and Language for Designing Agent Systems
In Proc. of Agent Oriented Information Systems, pages 45-57, 1999

[13] Wood M. F. and DeLoach S. A.
An Overview of the Multiagent Systems Engineering Methodology
The First International Workshop on Agent-Oriented Software Engineering (AOSE-2000), 2000

[14] Michael Wooldridge
Agent-Based Software Engineering
Mitsubishi Electric Digital Library Group
September 19, 1997
http://www.csc.liv.ac.uk/~mjw/pubs/iee-se.pdf

[15] G. Booch
Object-oriented analysis and design with applications
Addison Wesley 1994

[16] H. A. Simon
The sciences of the artificial
MIT Press 1996

[17] A. S. Rao and M. Georgeff
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, June 1995

[18] W. Vasconcelos, D. Robertson, J. Agusti, C. Sierra, M. Wooldridge, S. Parsons, C. Walton, and J. Sabater
A lifecycle for models of large multi-agent systems
Division of Informatics, University of Edinburgh
Department of Computer Science, University of Liverpool, etc
http://www.csc.liv.ac.uk/~mjw/pubs/aose2001.pdf

[19] Zave, P.
Classification of Research Efforts in Requirements Engineering
ACM Computing Surveys, 1997, 29(4): 315-321

[20] Bashar Nuseibeh
Computing Department, Faculty of Maths & Computing,
The Open University
http://mcs.open.ac.uk/ban25/

[21] Bashar Nuseibeh, Steve Easterbrook
Requirements Engineering: A Roadmap
Department of Computing, Imperial College
Department of Computer Science, University of Toronto
http://mcs.open.ac.uk/ban25/papers/sotar.re.pdf

[22] Jackson, M. & Zave, P.
Domain Descriptions
1st International Symposium on Requirements Engineering (RE'93)
San Diego, USA, 4-6 January 1993, pp. 56-64

[23] Easterbrook, S. M.
Resolving Conflicts Between Domain Descriptions with Computer-Supported Negotiation
Knowledge Acquisition: An International Journal, 1991, 3: 255-289

[24] Robinson, W. N. & Volkov, S.
Supporting the Negotiation Life-Cycle
Communications of the ACM, 1998, 41(5): 95-102

[25] Boehm, B., Bose, P., Horowitz, E. & Lee, M. J.
Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach
17th International Conference on Software Engineering (ICSE-17)
Seattle, USA, 23-30 April 1995, pp. 243-254

[26] Bennett, K. H. & Rajlich, V. T.
Software Maintenance and Evolution
In this volume, 2000

[27] Boehm, B. W.
Software Engineering Economics
Englewood Cliffs, NJ: Prentice-Hall

[28] Nakajo, T. & Kume, H.
A Case History Analysis of Software Error Cause-Effect Relationships
Transactions on Software Engineering, 1991, 17(8): 830-838

[29] Bohner, S. A. & Arnold, R. S. (Ed.).
Software Change Impact Analysis
IEEE Computer Society Press, 1996

[30] Estublier, J.
Software Configuration Management: A Roadmap
In this volume, 2000

[31] Jackson, M.
Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices
Addison Wesley, 1995

[32] Darke, P. & Shanks, G.
Stakeholder Viewpoints in Requirements Definition: A Framework for Understanding Viewpoint Development Approaches
Requirements Engineering, 1996, 1(2): 88-105

[33] Finkelstein, A. & Sommerville, I.
The Viewpoints FAQ: Editorial - Viewpoints in Requirements Engineering
Software Engineering Journal, 1996, 11(1): 2-4

[34] Eric S. K. Yu
Why Agent-Oriented Requirements Engineering
Faculty of Information Studies, University of Toronto
http://www.cs.toronto.edu/pub/eric/REFSQ97.html

[35] FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS
FIPA Modeling: Agent Class Diagrams
http://www.auml.org/auml/documents/CD-03-04-24.doc

[36] FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS
FIPA Modeling: Interaction Diagrams Working Draft
Version 2003-07-02
http://www.auml.org/auml/documents/ID-03-07-02.pdf

[37] James Odell, H. Van Dyke Parunak, Bernhard Bauer
Extending UML for Agents
http://www.jamesodell.com/ExtendingUML.pdf

[38] Radovan Cervenka
Modeling Notation Source
MESSAGE (Methodology for Engineering Systems of Software Agents)
Version: 03-03-12
http://www.auml.org/auml/documents/MESSAGE.pdf

[39] Steve Easterbrook
Requirements Engineering
Introduction Seminar Notes
Department of Computer Science
University of Toronto
http://www.cs.toronto.edu/~sme/CSC2106S/slides/01-intro.pdf

[40] Dave Robertson
How Software Projects Fail
Software Engineering with Objects and Components 2 module lecture note
Division of Informatics, University of Edinburgh
http://www.dai.ed.ac.uk/dai/teaching/msc/seoc2/slides/failures.ps.gz

[41] Bashar Nuseibeh
Weaving Together Requirements and Architectures
The Open University
http://www.doc.ic.ac.uk/~ban/pubs/computer2001.pdf

[42] S-PAD (Software Planning for Agile Development)
Software Engineering, School of Computer Science,
Queen's University Belfast
http://www.cs.qub.ac.uk/~Des.Greer/research.html

[43] Stephen Cranefield, Martin Purvis
Extending Agent Messaging to Enable OO Information Exchange
Number 2000/07 April 2000 ISSN 1172-6024
Department of Information Science, University of Otago
http://www.otago.ac.nz/informationscience/pubs/publications.html