NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY, MATHEMATICS AND ELECTRICAL ENGINEERING

Developing Role Figure Model based on UML Specification

Master Thesis

Fred Inge Henden

Spring 2004

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

FACULTY OF INFORMATION TECHNOLOGY, MATHEMATICS AND ELECTRICAL ENGINEERING

MASTER'S THESIS

Student's name: Fred Inge Henden

Area: Telematics

Title: **Developing Role Figure Model based on UML specification**

Description:

An UML specification for the TAPAS platform shall be produced. The specification shall give an overall description of the system and serve as a template for application development. A generic role figure specification template, which includes Class Diagram, State Diagram, Sequence Diagram etc., shall be provided. The role figure specification shall be translated into an XML-based manuscript and executed on the TAPAS support platform. A set of rules, guidelines and templates for role figure specification shall be specified for the design phase and will replace the vague Application Programming Interface available in plain text and Java method definitions.

Further, code generation from UML role figure specification to XML manuscript and Java code is to be performed, using a UML modeling tool. To experiment with the overall work, a forward engineering process shall be applied, starting from requirement analysis and specification, and ending with code generation and testing of the functionality. The application example shall be based on moving actors, which is conceptually referred to as Role Figure Mobility.

Start date: **20 January 2004**

Deadline: **15 June 2004**

Submission date:

Department: **Department of Telematics**

Supervisor: Mazen Malek Shiaa

Trondheim, 27 February 2004

Finn Arve Aagesen
Professor

Preface

This report is the result my master thesis at the Norwegian University of Technology and Science. The work with this thesis was carried out at the Department of Telematics during the spring semester of 2004 and has been a part of the TAPAS research project.

The thesis was suggested by my supervisor Mazen Malek Shiaa and the academic responsible has been professor Finn Arve Haagensen. I would like to thank them both for their advices and comments on my work.

I would also especially like to thank Mazen Malec Shiaa for always being available for questions during this semester. His advices and guidance is appreciated has been very valuable to me.

Trondheim, 21th of June 2004

Fred Inge Henden

Table of Contents

Figure List

Table List

Abstract

Telecommunication systems increase in complexity. The complexity and the inefficiency in the development process, installation and maintenance of the systems represent a major challenge in the future. An approach to face this challenge is to develop systems that are able to configure themselves in different environments and that supports dynamic introduction of new and distributed services.

TAPAS (Telematics Architecture for Plug and Play System) is a research project at the institute for telematics at NTNU. The project vision is to develop architecture concept for Plug-and-Play telecommunication equipment and services. The TAPAS concept is based on a theatre metaphor where actors play roles. An actor in TAPAS is a software entity responsible for providing and executing functionality. An actor constitutes a role figure by behaving according to a manuscript defining the behaviour of a particular role in a play. The TAPAS architecture requires a support system for software development, deployment, execution and management. The basic support system is currently implemented in Java and is based on Java RMI. A downsized version called MicroTAPAS is developed to support TAPAS for wireless devices with limited resources. Some example services are also developed for demonstration purposes.

The demands on short time-to-market from an idea is evolving to a complete service is available on the market is growing stronger, and the services are getting more complex. It is therefore important to be able to develop and introduce new services rapidly. The TAPAS architecture provides concepts which support dynamic introduction of new services. However, the existing support platforms and their APIs require extended knowledge of the system, to be able to develop new applications. In this thesis a template for application development on the TAPAS support platforms is created based on UML. UML models of the basic TAPAS support platform and the MicroTAPAS platform are created and a model of the Tele School example application is made. The platform models serve as an overall description of the system and are used to develop a template for application development. The template consists of a set of UML diagrams to describe a platform independent role figure model. This model is mapped to a UML class diagram specification which is used to generate Java code in the Rational Rose Real-Time modelling tool. A role figure specification in the representation language XML can also be obtained by a translation procedure from the role figure model.

The result of the thesis work is an application development environment for TAPAS based on UML and Java, where the complete modeling and implementation can be made using an UML modeling tool. Maintenance updates and further development of the TAPAS support platform can also be done in an efficient way by changing the model and generating new code. Experimentation with the overall work is done by developing an example application which is based on moving actors, conceptually referred to as Role Figure Mobility. The application is based on the electronic patient journal that is used in hospitals. It is developed to demonstrate how the concept of Role Figure Mobility can be used to make an application more flexible, and save time and effort for the people using it in their daily routines.

1 Introduction

Development of distributed systems and tele services is a complex and often time consuming task. Telecom systems are becoming more complex and heterogeneous. Qualified personnel are the critical factor for development, installation and deployment, as well as operation and maintenance of tele service software. The question of how to cope with this challenging situation in the future has been the motivation behind the TAPAS research project.

1.1 The TAPAS project

TAPAS (Telematics Architecture for Plug and Play System) is a research project at the institute for telematics at NTNU. The project vision is to develop architecture concept for Plug-and-Play telecommunication equipment and services [1]. TAPAS aims at developing an architecture for network-based service systems with:

A): flexibility and adaptability,
B): robustness and survivability,
and C): QoS awareness and resource control.

The goal is to enhance the flexibility, efficiency and simplicity of system installation, deployment, operation, management and maintenance by enabling dynamic configuration of network components and network-based service functionality.

Another objective is to gain experiences and knowledge by implementing those various features, both for demonstrating the implementation possibility and for validating the feature applicability. The project has emphasis on PhD and Master degree education and scientific results meant for publication.

1.2 Modelling of TAPAS

Developing a model for a software system before its construction is increasingly regarded as a necessary activity in information systems development. Good models are essential for the communication among members of the project teams and to assure that the system is possible to implement. Modelling activities have been a cornerstone in many traditional software methodologies for decades. As the complexity of systems increase, so does the importance of good modelling techniques. There are many additional factors of a project's success, but having a rigorous modelling language standard is one essential factor.

The use of object-oriented modelling in analysis and design started to become popular in the late eighties, producing a large number of languages and approaches. UML has taken a leading position in the area of object-oriented development partly through the standardisation of the language within the Object Management Group (OMG).

In this thesis UML shall be used to model the TAPAS support platforms and to model new applications in TAPAS. It shall be investigated if use of UML is an appropriate way to create a development process for the TAPAS applications, and the result of this work will be one of the steps towards a well established development process.

1.3 Outline of the thesis

Chapter 2 of this thesis gives an introduction to TAPAS and its concepts and architecture, and is intended for the reader that is new to TAPAS. The next chapter describes the basic concepts of UML with focus on the possibilities of modelling behaviour. A short introduction to object oriented methodology is also given. If the reader is familiar with UML and object-orientation this chapter may be skipped.

Chapter 4 describes how UML models are made for the TAPAS basic platform, the MicroTAPAS platform and the Tele School application respectively. These chapters present the models at a high level of abstraction and more detailed models can be found on the CD following this thesis.

Chapter 5 presents a template for application development on the TAPAS platform which is based on the models provided in the previous chapters.

In chapter 6 an example application is developed using the template provided and chapter 7 describes the testing which was carried out during the work with this thesis.
The last chapters include a discussion on the solutions chosen in this thesis and work to be done in the future and at the end the results are summarized in a conclusion of the thesis.

2 TAPAS concept and architecture

2.1 The theatre metaphor

TAPAS as a concept, is based on the theatre metaphor where actors perform roles according to predefined manuscripts, and a director manages their performance. Figure 2-1 shows the components in the theatre metaphor.

In TAPAS, actors are software components residing on different nodes in the network, representing the functionality that is executed at the node. The roles are modelled as extended finite state machines. A director is an actor with supervisory status and represents a domain, which is a set of nodes managed by a single director. TAPAS views service systems as predefined plays that consist of manuscripts defining roles. An essential part of the concept is that it shall be possible to plug generic actors into the system. The generic actors will then receive a manuscript which defines the actor's behaviour in the play that it will play a part in.

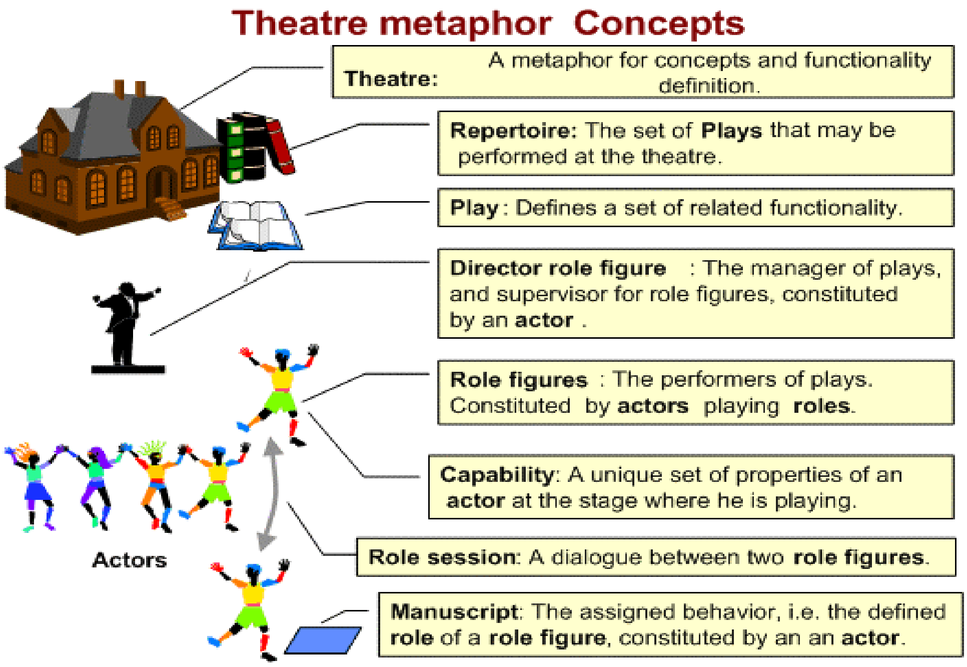

Theatre metaphor Concepts
Theatre: A metaphor for concepts and functionality definition.
Repertoire: The set of Plays that may be performed at the theatre.
Play: Defines a set of related functionality.
Director role figure : The manager of plays, and supervisor for role figures, constituted by an actor .
Role figures : The performers of plays. Constituted by actors playing roles.
Capability: A unique set of properties of an actor at the stage where he is playing.
Role session: A dialogue between two role figures.
Manuscript: The assigned behavior, i.e. the defined role of a role figure, constituted by an an actor.
Actors

Figure 2-1: The Theatre metaphor concept used in TAPAS

2.2 TAPAS architectures

TAPAS consists of four main architectures - the basic architecture, the mobility handling architecture, the dynamic configuration architecture and the adaptive service architecture [1]. Each of these architectures has different focus, and is aimed to solve specific issues related to handle the dynamic and adaptive nature of a plug-and-play networking system. These architectures require a support system for software development, deployment execution and management. Also, generic user functionality is required, to enable the flexibility features of the system. This support system is called the TAPAS platform. A brief overview of the basic architecture and the mobility architecture is given in the following sections, while these are the two architectures of interest in this thesis. More information on the dynamic configuration architecture and the adaptive service architecture can be found on the TAPAS project homepage http://tapas.item.ntnu.no.

2.2.1 TAPAS basic architecture

The TAPAS basic architecture is founded on the theatre metaphor. Generic actors in the nodes have the possibility to play different roles specified in corresponding manuscripts. Nodes can be network components and terminals. Actors are software units that can be executed on the nodes in the network. The roles are modelled as Extended Finite State Machines. Directors are specialised actors that manage actors in a domain and have a base of installed manuscripts and repertoires. The object model in Figure 2-2 illustrates the basic architecture.

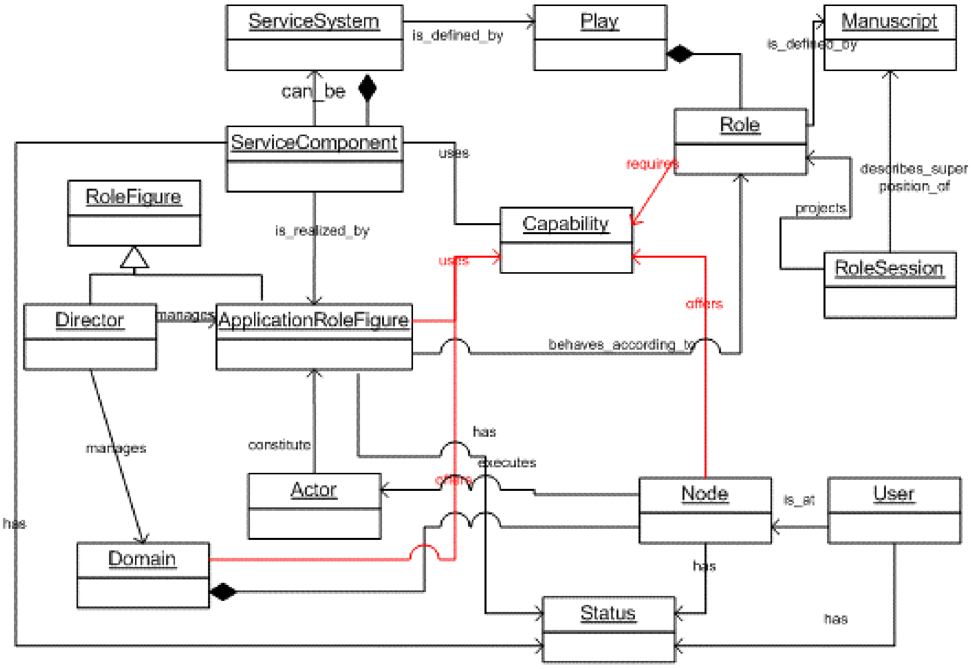


Figure 2-2: Object model of the TAPAS basic architecture

Actors constitute role figures that behave according to a *role*. An *actor* playing a particular *role* in a *play* is referred to as an *ApplicationRoleFigure*. The behaviour is defined by the *role's* corresponding *manuscript*. The *roles* have different requirements on *capabilities* and *status* of the *node* that the actor executes on. A *role figure* is realised in an executing environment and utilises the *capabilities* offered on a *node*. *Role sessions* are projections of an *actor's* behaviour when interacting with other *actors*. *Nodes* are part of a *domain*, which is managed by a *director*. The complete *service system* is defined by a *play* and consists of *service components*. A *service component* is realised by a *role figure*.

An *actor's* possibility to play different *roles* depends on the *capabilities* that the *role* requires and the *capabilities* offered by the *node* where the *actor* executes. *Capabilities* are the ability or power to do something. The *actor* will utilise these *capabilities* when executing on the *node*. *Capabilities* can be hardware properties like processing, storage, display and transmission resources (e.g. CPU, hard disk, screen resolution, bandwidth), extra equipment (e.g. printers) and software properties like data (e.g. user identification and authentication) and functions (e.g. different versions of platform).

The support functionality of the basic architecture is realised by a set of procedures: PlayPlugIn, PlayChangesPlugIn, PlayPlugOut, ActorPlugIn, ActorPlugOut, ActorBehaviourPlugIn, ActorChangeBehaviour, ActorBehaviourPlugOut, RoleSessionAction, ChangeActorCapabilities and Subscribe. These procedures are needed to provide the basic set of functionality given in the basic architecture.

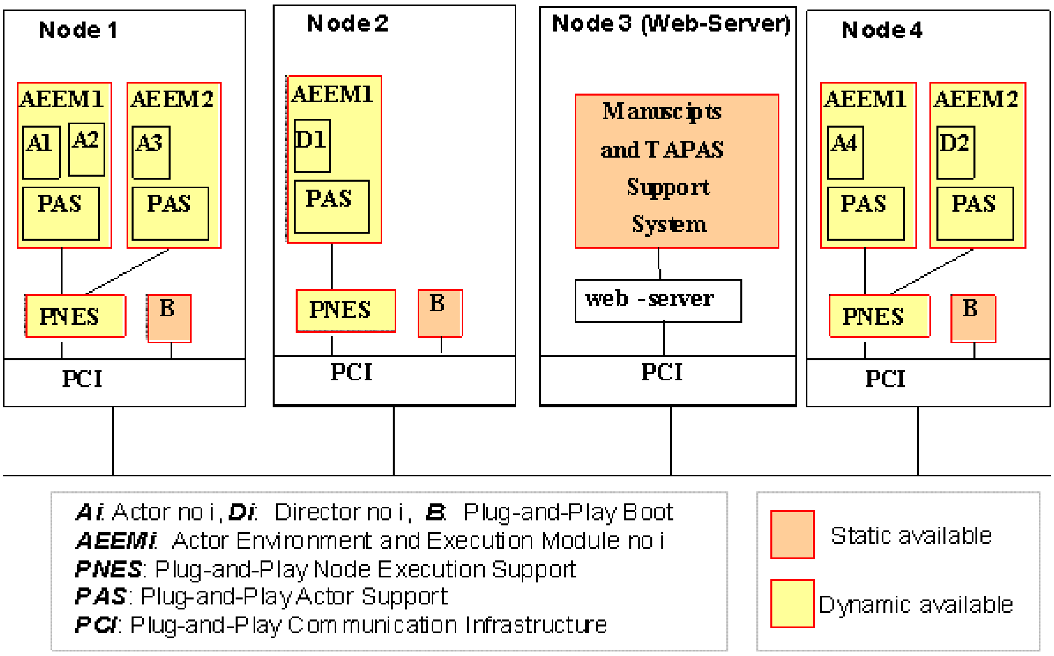

Node 1
AEEM1
AEEM2
A1
A2
A3
PAS
PAS
PNES
B
PCI
Node 2
AEEM1
D1
PAS
PNES
B
PCI
Node 3 (Web-Server)
Manuscipts and TAPAS Support System
web -server
PCI
Node 4
AEEM1
AEEM2
A4
D2
PAS
PAS
PNES
B
PCI
Ai: Actor no i, Di: Director no i, B: Plug-and-Play Boot
AEEMi: Actor Environment and Execution Module no i
PNES: Plug-and-Play Node Execution Support
PAS: Plug-and-Play Actor Support
PCI: Plug-and-Play Communication Infrastructure
Static available
Dynamic available

Figure 2-3: TAPAS system example

The structure of the support functionality is given by the illustration in Figure 2-3. This example shows actors executing on different nodes in a communication network. Each node has communication infrastructure support (PCI) that enables network communication between the nodes. PNES has to be present in every node that should be able to run TAPAS software. PNES is loaded by a static available bootstrap code that downloads the necessary code from the web server. PNES controls the execution of TAPAS software on a node, and routes requests between actors present at the node where PNES runs or other nodes. AEEM corresponds to a process or thread that executes a collection of actors with associated PAS. At the web server the manuscripts of roles and the basic support system is available and will be downloaded when needed.

2.2.2 TAPAS mobility handling architecture

The mobility handling architecture is an extension of the basic architecture and adds a new layer of functionality handling mobility functions. Mobility is an important aspect of dynamic and adaptive networking, and is needed for flexible service execution. The architecture is the basis of all functionality related to flexibility in personal, terminal and actor movement.

In TAPAS, four mobility features are supported: user, user session, terminal and actor mobility [1]. A user has user sessions and subscribed services that must be able to move along with the user as it changes access points. The user gets access to the network through a terminal. The terminal should be able to move in the network and still be able to access services and applications. Actors are instantiated functionality at a node that should be able to move along with its role sessions, state and variables. The illustration in Figure 2-4 shows the mobility concepts in TAPAS, and its relations.

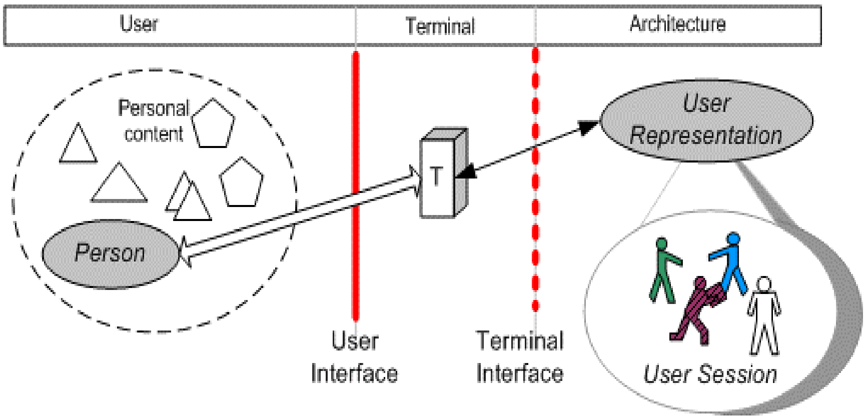


Figure 2-4: Mobility concepts in TAPAS

A user is represented by its personal content and can be related to a terminal through a user interface. The user can be identified by a username. The terminal interface can be identified by a network address and relates to a user representation that identifies the user in the system. A user may interact with the system, or services, within a defined user session. The double interface between the user and the system, with the terminal in the middle, enables a flexible way of representing users and terminals independently from each other.

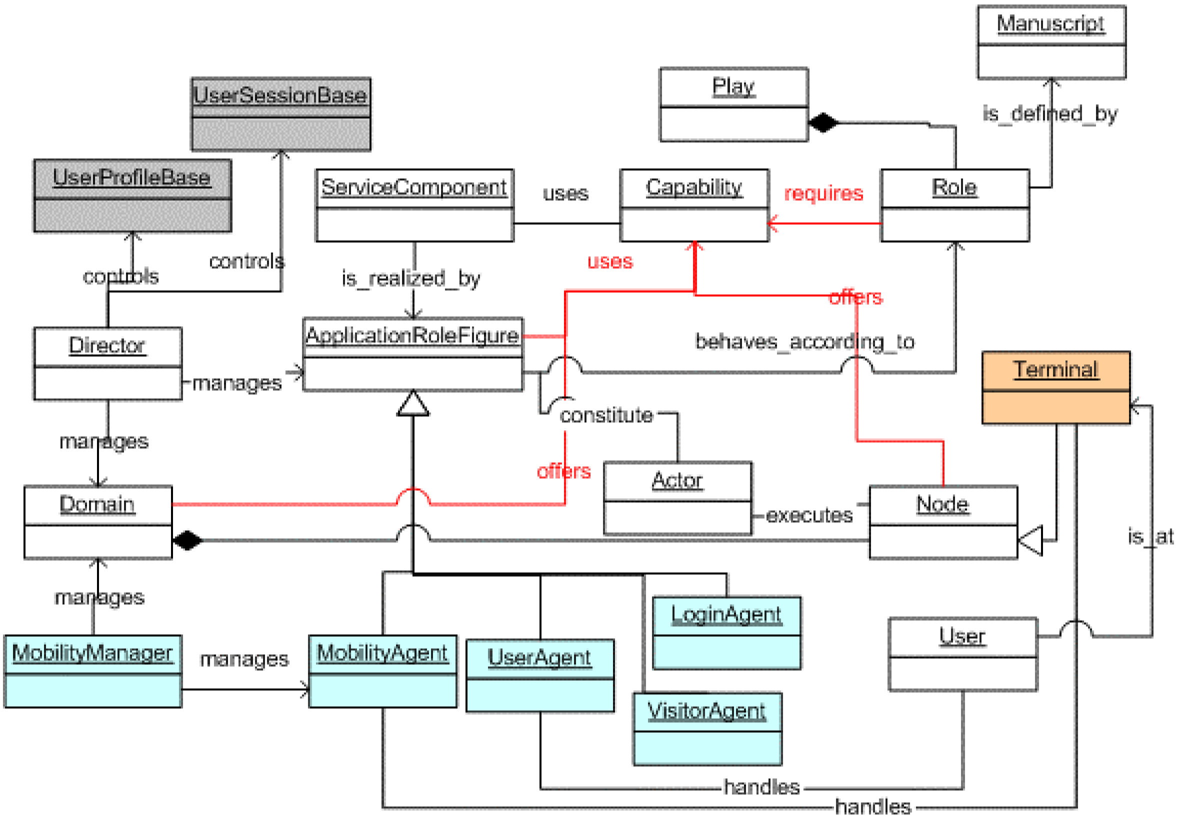


Figure 2-5 : Object model of the TAPAS mobility handling architecture

The TAPAS basic architecture is extended with emphasis on mobility, shown in Figure 2-5. In user session base, all user session information used by actors is stored. In user profile base, the information about users is stored. This information contains the user configurations and service subscriptions. The director of a domain controls both user profile base and user session base. In a domain, there is also one mobility manager. This object is responsible of managing actors and terminals mobility. Mobility agents that run in terminals aid this management and update the location-related information. A user agent manages user interactions with its home domain, and visitor agent manages user interactions with its visitor domain. The visitor agent is necessary because the user profile for a user is present in the user's home domain, but not in the visitor domain. A login agent enables controlled user access to services in the system.

2.3 TAPAS support systems implemented in Java

Two prototypes which implement the TAPAS support platform have been developed. In addition to the original TAPAS platform a downsized version is developed for wireless devices with limited resources [7]. The new and optimized version is called MicroTAPAS and it is based on the original TAPAS platform prototype, which means that no significant changes are made to the basic support functionality. New functionality is however added for mobility support. The platform has adopted two of the concepts in the mobility handling architecture; actor mobility and terminal mobility. The next two sections briefly describe the layered architectures of the two platforms.

2.3.1 The original layered design model

Figure 2-6 shows the layered design model for the basic TAPAS platform. Each layer in the model contains different support functionality, and a description of the different layers from [8] is summarized as follows:

- **Plug and Play communication infrastructure (PCI):** PCI uses Java RMI and 'rmregistry' for communication between nodes which constitutes a TAPAS domain.
- **Plug and Play node execution support (PNES):** This layer makes it possible to run TAPAS on a node and facilitates routing of communication between nodes.
- **Plug and Play actor support (PAS):** Makes it possible to create and execute actors within an operating system process. Additional functionality is routing between actors and PNES instances. Each PAS instance is a separate Java Virtual Machine (JVM) instance.
- **Director:** The director is responsible for management of plays, manuscripts and actors in its domain and communicates with other actors.
- **Plug and Play extended management (PXM):** Support of extended services not required for TAPAS basic support, but to satisfy specified operational properties and requirements.
- **Plug and Play extended support (PXS):** Required for the applications to utilize PXM functionality.
- **Plug and Play applications:** The collection of application actors. Instances created/moved by using ActorPlugIn/ActorPlugOut support functions. Interfaces PAS.
- **Non Plug and Play applications:** Functionality not defined according to ApplicationActor requirements, but is allowed to communicate with actors, and to utilize TAPAS support functionality.

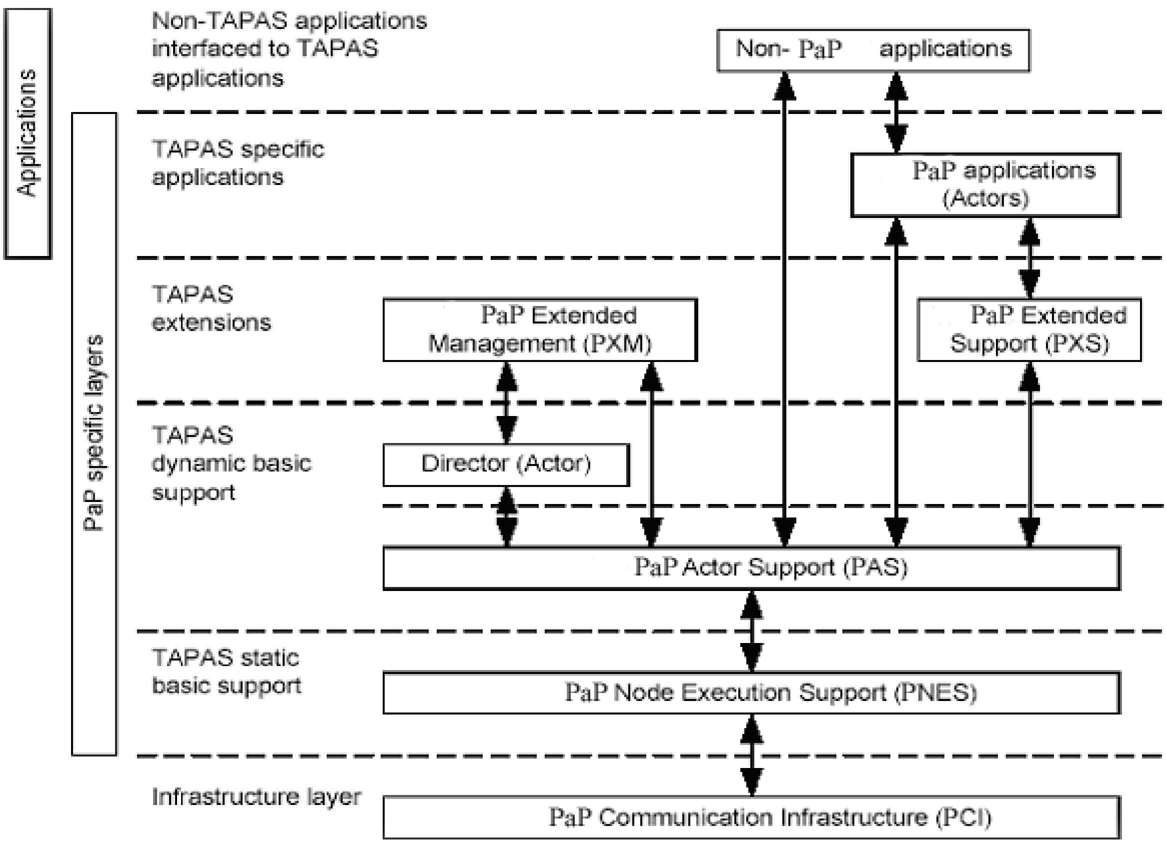

Applications
PaP specific layers
Non-TAPAS applications interfaced to TAPAS applications
TAPAS specific applications
TAPAS extensions
TAPAS dynamic basic support
TAPAS static basic support
Infrastructure layer
Non- PaP applications
PaP applications (Actors)
PaP Extended Management (PXM)
PaP Extended Support (PXS)
Director (Actor)
PaP Actor Support (PAS)
PaP Node Execution Support (PNES)
PaP Communication Infrastructure (PCI)

Figure 2-6 : TAPAS layered design model - architecture

2.3.2 The layered design model for MicroTAPAS

Figure 2-7 shows the layered architecture of the MicroTAPAS platform as presented in [7]. The major change in this architecture from the one shown in Figure 2-6 is that the PAS and PNES layers are merged into one PNES layer. Another important modification from the original design model that is not visible is that Java/RMI communication is replaced by plain socket communication in the PCI layer.

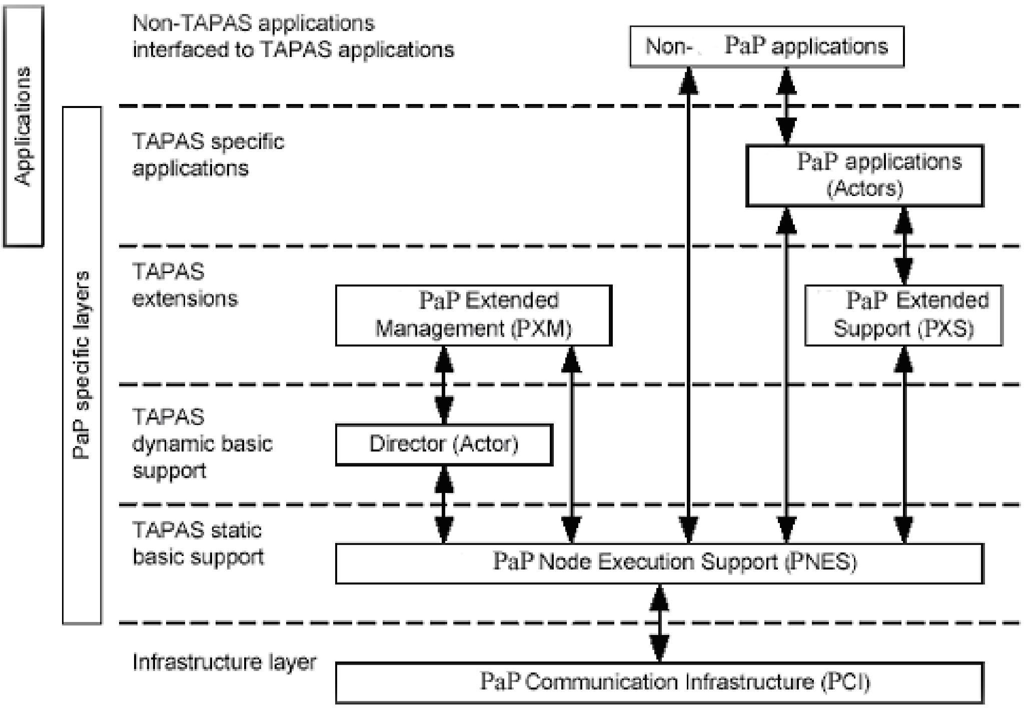

Applications
PaP specific layers
Non-TAPAS applications interfaced to TAPAS applications
Non- PaP applications
TAPAS specific applications
PaP applications (Actors)
TAPAS extensions
PaP Extended Management (PXM)
PaP Extended Support (PXS)
TAPAS dynamic basic support
Director (Actor)
TAPAS static basic support
PaP Node Execution Support (PNES)
Infrastructure layer
PaP Communication Infrastructure (PCI)

Figure 2-7 : MicroTAPAS layered design model - architecture

2.4 Example applications

A few example applications are developed to demonstrate use of the TAPAS platform. APIs and executable code for the Tele School and The Watcher application can be found at http://tapas.item.ntnu.no. The Tele School application is used to make a first time UML model for a TAPAS application and is described in section 4.3

2.5 Summary

This chapter has given an introduction to the concepts and architecture of TAPAS. The TAPAS concept is based on the theatre metaphor, where actors play roles in a play, managed by a director. An actor behaving according to a given manuscript, defining the functional behaviour of a role in a play, is referred to as a role figure. TAPAS consists of four main architectures - the basic architecture, the mobility handling architecture, the dynamic configuration architecture and the adaptive service architecture [1]. An overview of the two architectures that will be used in this thesis, the basic architecture and mobility handling architecture, have been given. Two prototypes for TAPAS support platforms exist. The TAPAS basic support platform is based on the basic architecture. MicroTAPAS is a downsized version of the basic support platform, which are developed to execute on small wireless devices. MicroTAPAS also realize two of the concepts of the mobility handling architecture; role figure mobility and terminal mobility.

3 UML modelling

The Unified Modelling Language has quickly become the de-facto standard for building Object-Oriented software, and a range of CASE tools which supports UML modelling are available at the market today. This chapter will give a quick introduction to UML and its concepts. Then the features of the available modelling tools will be investigated before the modelling tool used in this thesis work will be described.

3.1 Introduction to UML

The OMG specification [5] states:

"The Unified Modelling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."

UML defines the notation and semantics for the following domains:

- **The User Interaction or Use Case Model** - describes the boundary and interaction between the system and users. This model corresponds in some respects to a requirements model.

- **The Interaction or Collaboration Model** - describes how objects in the system will interact with each other to get work done.

- **The State or Dynamic Model** - State charts describe the states or conditions that classes assume over time. Activity graphs describe the workflows the system will implement.

- **The Logical or Class Model** - describes the classes and objects that will make up the system.

- **The Physical Component Model** - describes the software (and sometimes hardware components) that make up the system.

- **The Physical Deployment Model** - describes the physical architecture and the deployment of components on that hardware architecture.

UML provides several types of diagrams that, when used within a given methodology, increase the ease of understanding an application under development. UML defines a complete object-oriented notation. It does not specify a methodology to be used, and is hence referred to as a methodology independent notation. The UML diagrams offer a good introduction to the language and the principles behind it.

3.1.1 The UML diagrams

The underlying premise of UML is that all the different elements of a system can not be entirely captured by one diagram alone. UML provides a number of diagrams as a mechanism for entering model elements into the model and showing overlapping sets of models elements and their relationships. UML does not specify what diagrams should be created or what they should contain, only what they can contain and the the rules for connecting the elements. The diagrams in UML are [5]:

- **Use case diagram:** The use case diagram is used to identify the primary elements and processes that form the system. The primary elements are termed as "actors" and the processes are called "use cases." The use case diagram shows which actors interact with each use case. An example of a use case diagram is shown in Figure 3-1 below. The use case diagram can also show relations among use cases. The 'extends' relationship describes an alternative option under a certain use case. The 'uses' relationship shows that another use case is needed by a particular use case to perform a task.

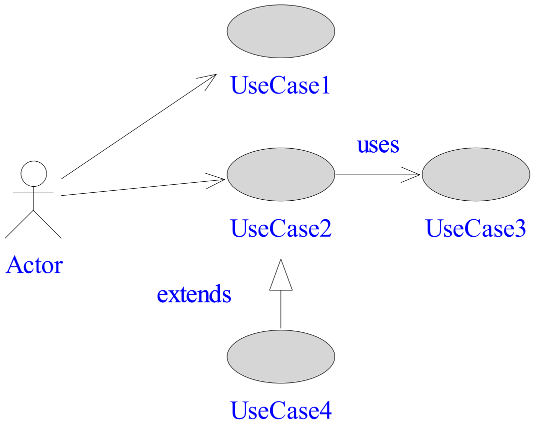


Figure 3-1 : Use case diagram

- **Class diagram:** The class diagram is used to refine the use case diagram and define a detailed design of the system. The class diagram classifies the actors defined in the use case diagram into a set of interrelated classes. Each class in the class diagram may be capable of providing certain functionalities. These functionalities provided by the class are termed "methods" of the class. Apart from this, each class may have certain "attributes" that uniquely identify the class. Figure 3-2 shows an example of a class diagram. The *dependency* relationship is used between class2 and class3 is a weak association and indicates some sort of dependency between the classes. In this case the relationship is *has* and the cardinality is added to indicate that one instance of class2 has one or more instances of class3. The *generalization* relationship is used between class1 and class2, which means that class2 inherits the attributes and operations of class1. *Composition* is used between class2 and class4 and class5. This indicates that class4 and class5 is parts of 'the whole' class2.

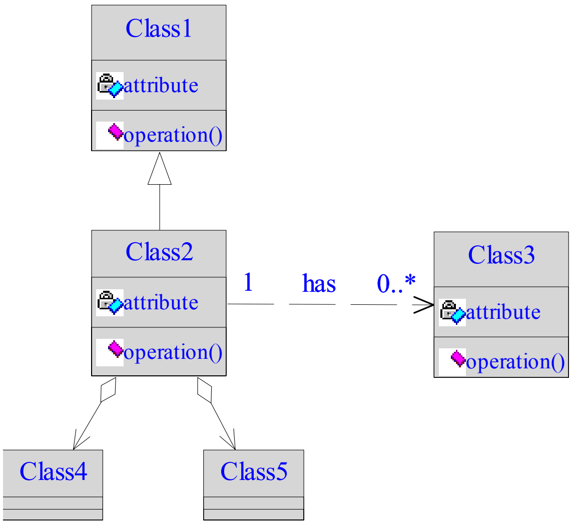


Figure 3-2 : Class diagram

- **Object diagram:** The object diagram is a special kind of class diagram. An object is an instance of a class. This essentially means that an object represents the state of a class at a given point of time while the system is running. The object diagram captures the state of different classes in the system and their relationships or associations at a given point of time. Figure 3-3 below shows how the relationship between class2 and class3 from the class diagram in Figure 3-2 can be specified in more detail for certain instances of the classes. The objects are specified with name of the instance and the type, i.e. the class name.

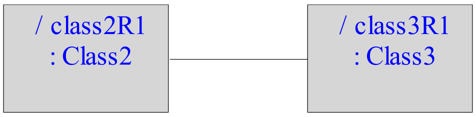


Figure 3-3 : Object diagram

- **State diagram:** A state diagram, as the name suggests, represents the different states that objects in the system undergo during their life cycle. Objects in the system change states in response to events. In addition to this, a state diagram also captures the transition of the object's state from an initial state to a final state in response to events affecting the system. Figure 3-4 shows an example on a state diagram. The arrows between the states indicate the event which caused the transition to a new state. The diagram also contains a decision point CP1 which can be either true or false, and a transition to a new state is executed depending on the result. State3 in the diagram is special compared to the other states in the way that it has a small symbol indicating that it has sub states. This means that state3 has its own state diagram describing its sub states. The point Final1 in Figure 3-4 is a termination point for this state diagram.

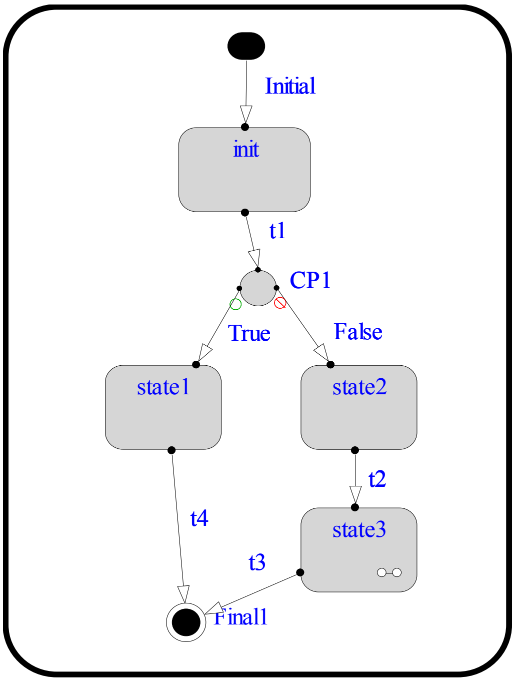


Figure 3-4 : State diagram

- **Activity diagram:** The process flows in the system are captured in the activity diagram. Similar to a state diagram, an activity diagram also consists of activities, actions, transitions, initial and final states, and guard conditions. In Figure 3-5 an example of a UML activity diagram is shown. Branches and forks are used to describe alternative and parallel flows respectively.

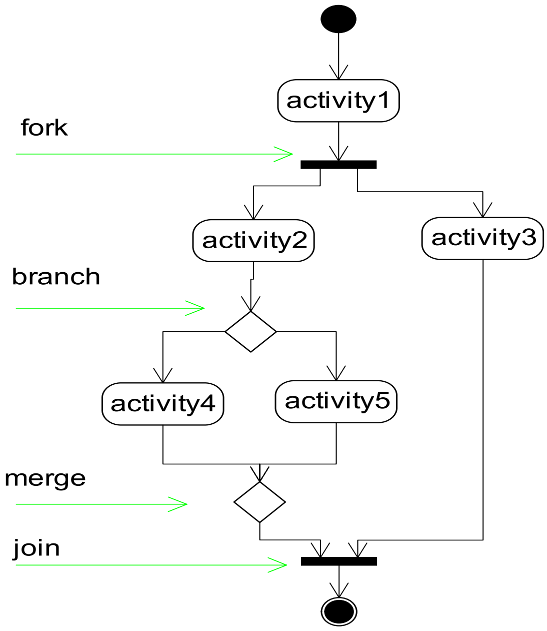


Figure 3-5 : Activity diagram

- **Sequence diagram:** A sequence diagram represents the interaction between different objects in the system. The important aspect of a sequence diagram is that it is time-ordered. This means that the exact sequence of the interactions between the objects is represented step by step. Figure 3-6 shows a simple sequence diagram. As can be seen in the figure message flow between instances can be synchronous or asynchronous. Transaction number 1 is an asynchronous message passed between two objects, while transaction number 2 describes a synchronous method call. It is also possible to describe the state of the instances and actions to be taken as a result of message reception.

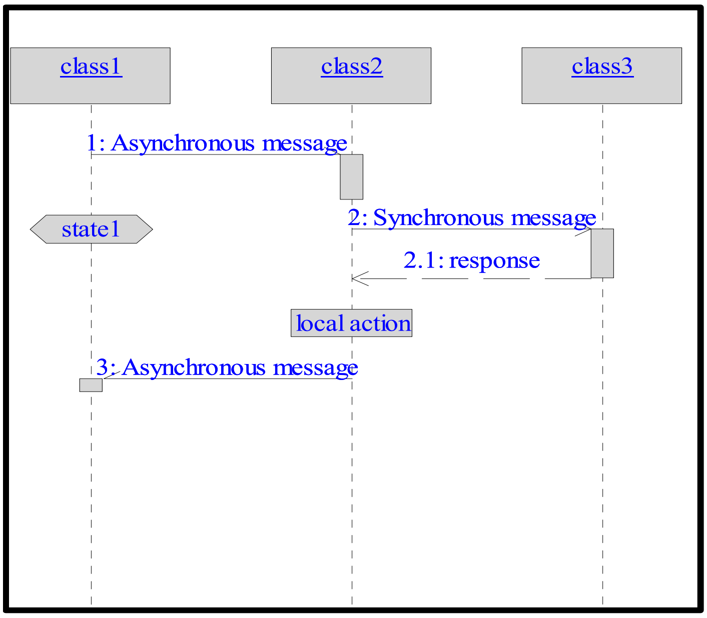


Figure 3-6 : Sequence diagram

- **Collaboration diagram:** A collaboration diagram groups together the interactions between different objects. The interactions are listed as numbered interactions that help to trace the sequence of the interactions. The collaboration diagram helps to identify all the possible interactions that each object has with other objects. Figure 3-7 shows how interactions between the objects are described with arrows. The sequence is numbered, but does not specify time like the sequence diagram.

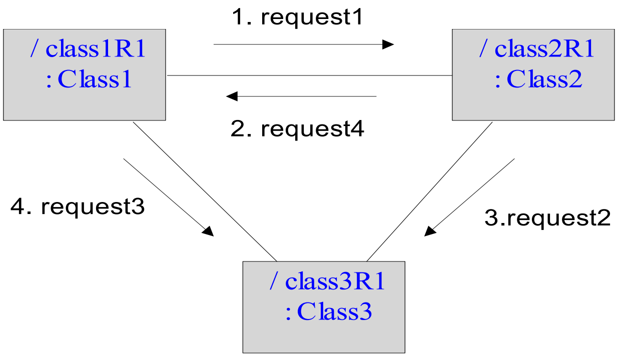


Figure 3-7 : Collaboration diagram

- **Component diagram:** The component diagram represents the high-level parts that make up the system. This diagram depicts, at a high level, what components form part of the system and how they are interrelated. A component diagram depicts the components culled after the system has undergone the development or construction phase. As can be seen in the example in Figure 3-8 the diagram shows the organization of the physical software components in a system. The dotted line shows that component1 and component2 depends on component3 in this system.

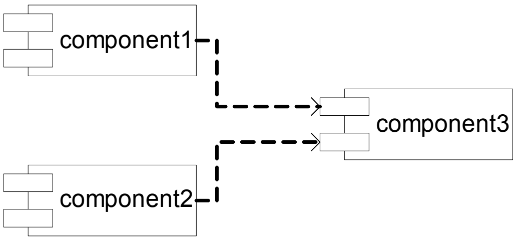


Figure 3-8 : Component diagram

- **Deployment diagram:** The deployment diagram captures the configuration of the runtime elements of the application. This diagram is by far most useful when a system is built and ready to be deployed. Figure 3-9 shows how the deployment diagram is used to describe how the software components are deployed on different nodes.

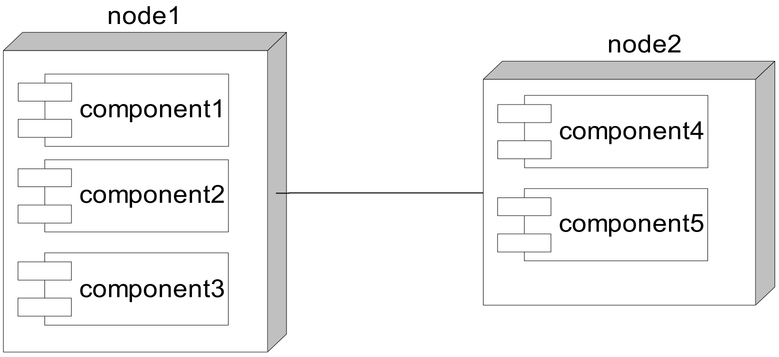


Figure 3-9 : Deployment diagram

3.1.2 Modelling of dynamic behaviuor

UML has its background from the enterprise applications. In these applications data base modelling has been important and UML contains well defined mechanisms for modelling of static data. Demand and support for formal modelling of dynamic behaviour is however lacking in UML. Telecom companies have for example used SDL[10] to develope theire complex real-time systems, which makes it possible to make a functional model of the system to be used for formal analysis, code generation and verification.

3.1.3 UML 2.0

The new version of UML is called UML2.0 and was standardized in June 2003. It provides some major improvements over UML 1.4, and has due to strong participation from the major telecommunication companies like Ericsson, Motorola and Simens taken a big step towards SDL. Due to the elements taken from SDL some significant improvements are made in the support of behaviour modelling. [11] summarizes the major improvements in UML2.0 by the following bullets:

- New concepts for describing the internal architectural structure of Classes, Components and Collaborations by means of Part, Connector and Port.

- Introduction of inheritance of behaviour in state machines and encapsulation of sub machines through use of entry and exit points.

- An improved encapsulation of components through complex ports with protocol state machines that can "control" interaction with the environment.

- Improvements of actions and activities and the use of flow semantics instead of state machines.

- Interactions are improved with better architectural and control concepts such as composition, references, exceptions, loops and alternatives. Interaction Overview Diagrams also gives better overview.

- The new concepts; Part, Connector and Port are introduced to increase the architectural support and makes it possible to describe a class behaviour as a collaboration of behaviour of the internal instances of the class.

State machines in UML2.0 are used to model discrete behaviour triggered by events such as signals, timeouts, operation calls and change in values. The triggers cause a transition if the trigger is specified for the current state of the state machine. State machines can also be used to express the protocol through a port. The two different kinds of state machines are referred to as behavioural state machines and protocol state machines. In UML1.4 there were no limitations on how to enter and exit composite states. In UML2.0 entry and exit points are named points that are placed in the frame of the state machine. Another important new feature of UML is that behaviour can be inherited and specialized.

3.2 UML modelling tools

A range of CASE tools supporting UML are available, offering various features to the designer of the UML models.

3.2.1 Modelling tools features

The primary use of a UML tool is to enable the user to draw diagrams and create a model. However, more features can be expected from the CASE tools supporting UML available on the market today. Below is a summary of some common UML tool features:

- **UML diagram support:** The UML tool should support all the nine diagrams that make up UML.
- **Forward engineering:** The characteristic of automating the generation of source code is called forward engineering. Tools which support forward engineering are able to create source code from the classes with the methods stubbed out. This stub code can be filled with actual code by the developer.
- **Reverse engineering:** Reverse engineering is exactly the opposite of forward engineering. In reverse engineering, the UML tool loads all the files of the application/system, identifies dependencies between the various classes, and reconstructs the entire application structure along with all the relationships between the classes. Reverse engineering is a feature normally provided by sophisticated and high-end UML tools.
- **Round-trip engineering:** An important rule in software design is that no design remains unchanged. This is as true for small systems as it is for large systems. It becomes very difficult to keep the design of the system updated with the changes in the source code. The round-trip engineering feature enables the UML tool to synchronize the model with the changes in the application code.
- **Documentation:** A UML tool must necessarily provide some way for the designer to document design decisions in the diagrams by using simple things such as annotations or comments. In addition to this, the UML tool should support the generation of reports/listings of the different design elements of the diagram.
- **Version control:** Configuration management is an integral part in the building of software systems. Considering that the design of a system is a very important artefact of the software lifecycle, maintaining versions and baselines of the system design is a desirable feature to have in UML tools.
- **Collaborative modelling environment:** A collaborative design effort needs to be properly synchronized by the UML tool. Some UML tools provide support for a collaborative modelling environment with capability to compare different versions designs for differences or even merge different versions of a design.
- **Integration with popular Integrated Development Environments (IDE):** This feature would enable the UML tool to be updated with the changes in the source code made in the IDE.
- **Test script generation:** In addition to generating stub code, the tool also generates test scripts that can be used for testing how the generated class functions.

3.2.2 Selecting a modelling tool

Many factors may have an influence when selecting a modelling tool. Some tools are open source and some tools require rather expensive licenses, not automatically implying that they are any better than the ones which are free of charge. An important issue when selecting the tool is how it will integrate with other tools and that code for the desired programming language can be generated.

As stated in section 3.1.3 UML2.0 introduces many enhancements for modelling of behaviour. However, very few tools on the market are jet supporting the new version of UML. At the time of writing Telelogic G2 claims to be the only modelling tool to fully support UML2.0, but the tool does not yet support generation of Java code (version 2.2).

Since the TAPAS platforms are implemented in Java, code generation to Java is regarded an essential feature and a tool supporting this feature will be preferred. Hence, a modelling tool from Rational Software is chosen which supports generation of Java code and UML1.4.

3.3 The Rational Rose Real-Time modelling tool

The Rational Software Company has played a significant role in the development of UML. Their CASE tool Rational Rose is one of the best known UML tools and comes in several editions. Rational Rose Real-Time is a modelling tool tailored to meet the demands of real time systems. In the following sections the features offered by this tool will be described.

3.3.1 Real-Time features

In addition to supporting the core UML constructs, Rational Rose Real-Time uses some of the extensibility features of UML to define some new constructs which is specialized for real-time system development. The new constructs allow code generation of elements which uses services provided in a Service Library such as concurrent state machines, message passing and timing services. These are services that most real time systems must implement. Adding of these services allows the designer to concentrate on the system design instead of e.g. concurrency issues.

So, how does these new constructs fit with modelling of TAPAS? Since TAPAS model actors their behaviour as state machines which communicates by message exchange, it looks like using these features will be a good idea. Modelling the TAPAS applications as independent and concurrent state machines in this tool would be very quick and code generation features for these state machines are very good. However, some of the TAPAS concepts are not easily adoptable to these constructs. The first problem is that additional classes are generated for the service library which handles scheduling of tasks, message passing etc. One of the reasons for the good code generation features in this tool is that a support system for the real time modelling is included in the model. In TAPAS this would result in quite large overhead in the code representing the manuscripts. The real time is based on running of one state machine in a separate thread and TAPAS has its own organization of threads and scheduling. Clearly, these two support systems are not easily integrated. Another argument for not using the real time constructs is that they are not standardized, but are special features added to UML 1.4 by Rational Rose. It is not a good idea to depend on a particular tool when developing the TAPAS models.

Due to the results of the discussion above the real time features of the tool is not used in this thesis. Instead, the standard UML features of the tool are used. The next chapter describes the standard features of the Rational Rose Real-Time tool.

3.3.2 Standard UML features

Section 3.2.1 summarizes some of the features which can be expected by the UML modelling tools. The diagrams supported by the Rational Rose Real-Time modelling tool are listed below:

- use case diagrams
- class diagrams
- state diagrams

- collaboration diagrams
- sequence diagrams
- component diagrams
- deployment diagrams

Of these diagrams, the class diagrams are essential to create an executable model. As can be seen activity and object diagrams are not supported by this tool.

Further, forward engineering and reverse engineering, are supported for the standard UML constructs as well as for the real-time extensions.

3.4 Methodology

A methodology formally defines the process that is used to gather requirements, analyze them, and design an application. As mentioned UML does not specify a methodology, which means that UML provides constructs to be used during the development, but it does not say anything about how the work shall be done. There are many methodologies for software development, each differing from the other in some way. Selecting a methodology for a project very much depends on what type of application is to be developed. Some methodologies are developed to be used for small embedded systems, with high demands for safety, while others are best suited for large scale business applications. Some methodologies support a large number of developers while others are best suited for a small team or just a single person.

When selecting a methodology, it must usually be customized to fit the organization adopting it. Some processes are quite heavy and does not apply well to small organizations. In this case, customizing may mean picking what is needed from the original process. Some very large companies may even need to extend the original process to meet their development requirements. However, most processes are founded on some basic ideas for software development. A basis for many object-oriented development processes are the use case driven approach [12].

3.4.1 The use case driven approach

In the book Object-Oriented Software Engineering – a use case driven approach [12] an object-oriented development process is described. This book is by many considered a classic and was the first book to put forth the idea that the customer's requirements, as expressed within use cases, should be the most important driving force in software development. The process described goes from the requirements documentation to a finished and maintainable software product and is referred to as the use case driven approach. Use case driven means that the use case defined are the basis for the entire development process. This is illustrated in Figure 3-10 below.

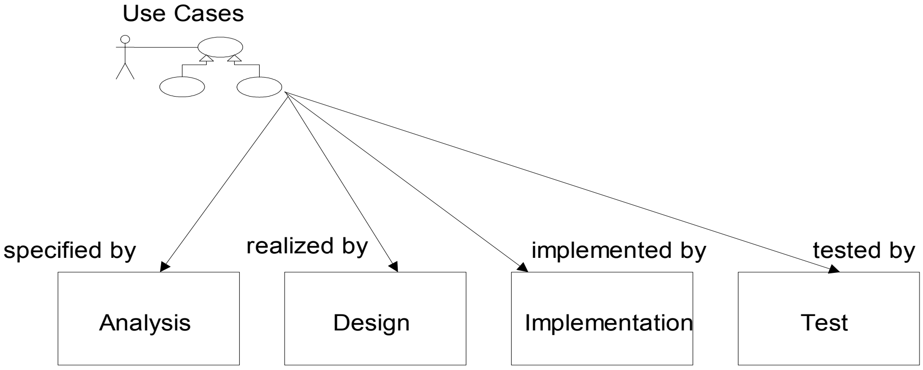


Figure 3-10 : The use case driven approach

The four phases specified are: Analysis, design, implementation and test. As can be seen the use cases are input to all phases of the development process:

- **Analysis** - is intended for defining the requirements and the relationships between the required functions of the system. Use cases are further specified to find the analysis classes to realize the use cases.

- **Design -** takes the models and preparations of the Analysis phase and applies them in terms of the actual environment where the computer program will run. In the design phase, the use cases are realized by classes of the programming language chosen.

- **Implementation** – the classes from the design phase are implemented.

- **Test** – the implementation is tested according to the specified use cases.

3.4.2 The Unified Process

After creating UML as a single complete notation for describing object models, the creators of UML turned their focus to the development process. The process described in [12] does not address issues as project management and development tools. A new process called the Unified Process was developed to address these issues and to introduce a more complete methodology. The Unified Process is actually more like a generic process framework that developers can customize by adding and removing activities based on the particular needs and available resources for a project.

One of the key aspects of the Unified Process is the use case driven approach described in the previous section. The key aspects of the process are:

- Use case driven
- Architecture centric
- Iterative and incremental

The Unified Process specifies that the architecture of the system being built, as the fundamental foundation on which that system will rest [14], must sit at the heart of the project team's efforts to shape the system, and also that architecture, in conjunction with the use cases, must drive the exploration of all aspects of the system.

The third fundamental aspect of the Unified Process is its *iterative and incremental* nature [14]. An iteration is a mini-project that results in a version of the system that will be released internally or externally. This version is supposed to offer incremental improvement over the previous version, which is why the result of an iteration is called an increment.

One well known process which is an example of a specialized version of the Unified Process that adds elements to the generic framework is the Rational Unified Process (RUP). Some links to object-oriented processes can be found at the OMG recourse page: http://www.uml.org/

3.5 Summary

This chapter has briefly described UML. Since one of the main objectives in this thesis is to develop a specification for the TAPAS role figure, main focus has been on description of behaviour. The main features of UML2.0 have been described, which shows that UML2.0 has been improved with respect to modelling behaviour. Some of the features to expect from the CASE tools for UML are described and a description of the modelling tool which is used in this thesis work was given. Finally, the use case driven approach and some processes for object-oriented software development were introduced.

4 TAPAS models

This chapter describes how UML models are made for the TAPAS basic support platform, the MicroTAPAS platform and the Tele School example application. The models will be used to create a template for future development of TAPAS applications.

4.1 Model of the TAPAS support platform

The TAPAS support platform which provides the middleware functions in TAPAS is developed in Java J2SE and communication is realized by Java RMI. Documentation of the implementation is insufficient and only a textual API exists. The implementation of the support system is quite complex and it is hard to figure out how it works by reading code and APIs. In addition to complete the documentation of the system a complete model will also make further development and changes to the platform easier to implement. In this chapter a UML model is made for the TAPAS support platform.

4.1.1 Use cases

As described in section 3.1.1 the use case diagram is used to describe the requirements of the system and the functionality that it provides. For the TAPAS support platform we start by identifying the use cases. To make the use case diagrams, the functions that the platform offers and the users of these functions needs to be found. The procedures needed to provide the basic set of functionality given in the basic architecture are: PlayPlugIn, PlayChangesPlugIn, PlayPlugOut, ActorPlugIn, ActorPlugOut, ActorBehaviourPlugIn, ActorChangeBehaviour, ActorBehaviourPlugOut, RoleSessionAction, ChangeActorCapabilities and Subscribe. The users of the procedures are: an actor in TAPAS, the director or an instance outside TAPAS. An overview of the use cases for the support platform is shown in Figure 4-1.

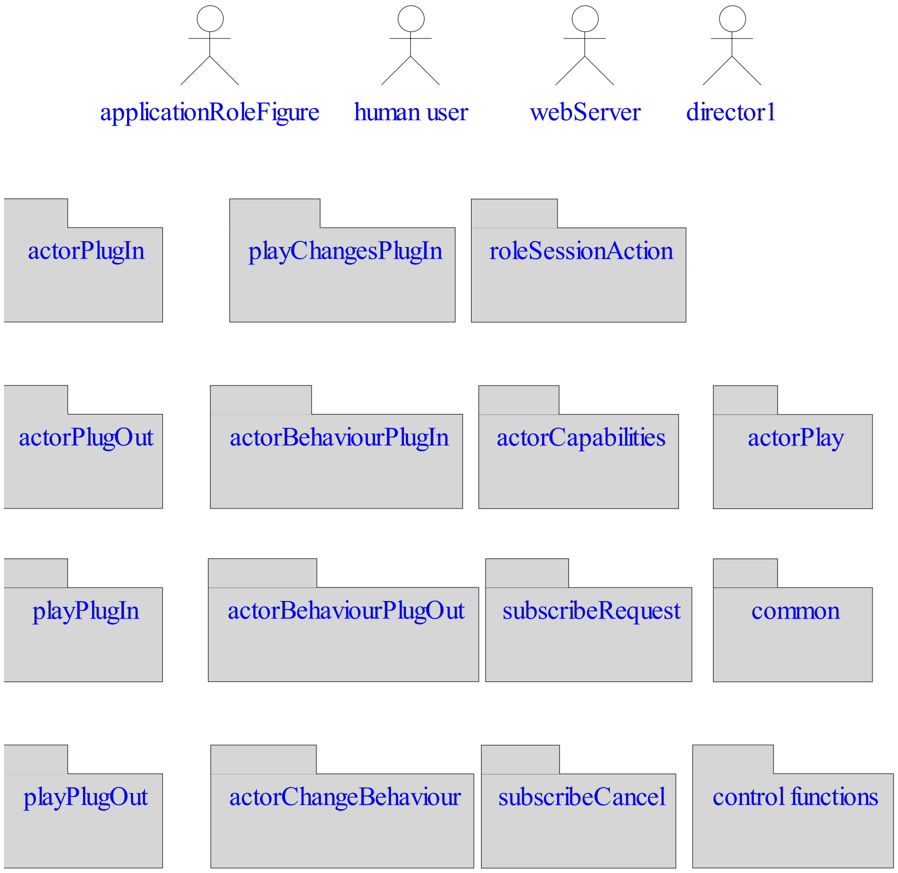


Figure 4-1 : Use cases for the TAPAS support platform

As can be seen in the figure, each TAPAS procedure is contained in one package and some packages also contains common use cases and control functions (i.e. for communication with the instance through a consol). A forth user of the system, the web server, is also specified. The TAPAS platform uses a web server for storing of the code base and the available plays, thus this is a passive external user of the system.

As an example Figure 4-2 shows the use cases which are needed to complete the actorPlugIn procedure. The use cases are described in the actorPlugIn package. The actorPlugInReq use case is composed of the use cases *send actorPlugIn* and *request to actor,* which is a generic use case for sending a request to another actor. The director performs the plugin of the actor to the PNES which results in the creation of a new actor and downloading of the manuscript from the web server.

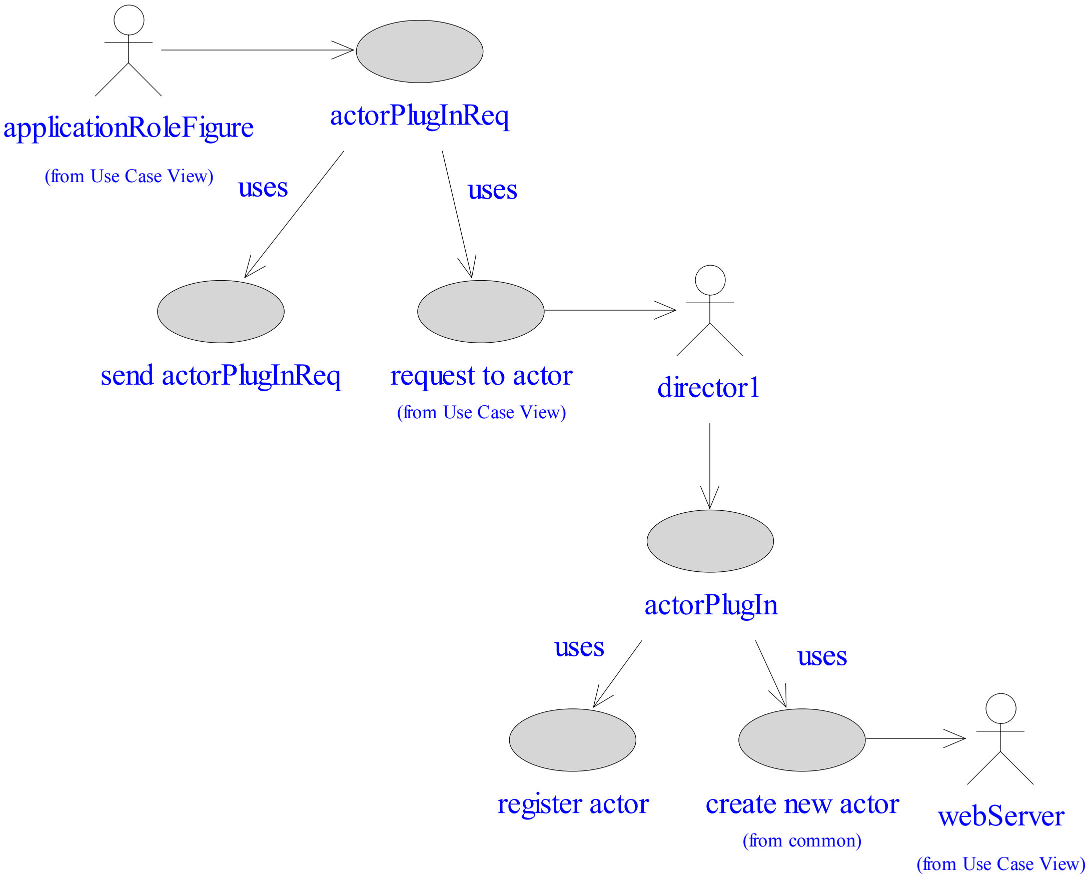


Figure 4-2 : The actorPlugIn procedure

4.1.2 Class diagrams

Before looking into the use cases in more detail, a description of the static structure of the implementation must be made. Figure 4-3 shows a class diagram for the classes used to realise the PAS and PNES layers described in section 2.3.1. The PAS, PNES and DebugServer classes implement objects in the TAPAS architecture that will communicate with objects residing on other nodes. In the basic version of the support platform communication is handled by Java RMI. The RMIServer class is a base class which inherits the UnicastRemoteObject class and provides the functionality needed for RMI communication. As can be seen in Figure 4-3 the PAS, PNES and DebugServer extend the RMIServer class.

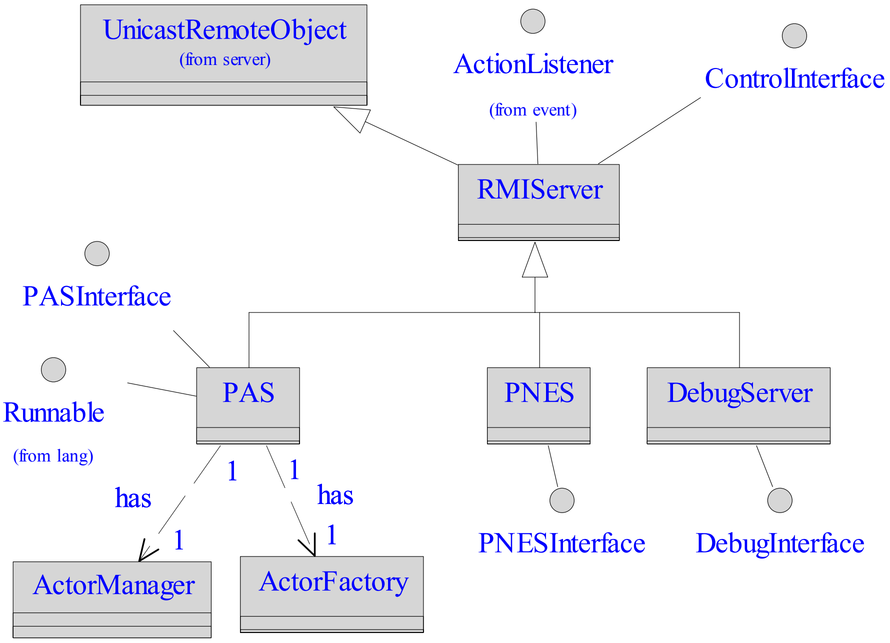


Figure 4-3 : Class diagram for the TAPAS support entities

The methods which are offered by PAS, PNES and DebugServer are specified in the PASInterface, PNESInterface and DebugInterface classes.

The next class diagram, shown in Figure 4-4, describes how the actor entity is realized. The two types of actors that are defined are the ApplicationActor and the DirectorActor. An ApplicationActor resides in one PAS which belongs to an instance of PNES, while one DirectorActor controls the PAS. For the DirectorActor class an implementation called Director1 is made which will be the director provided with the support platform. The implementations of the ApplicationActor will be defined in the specific applications.

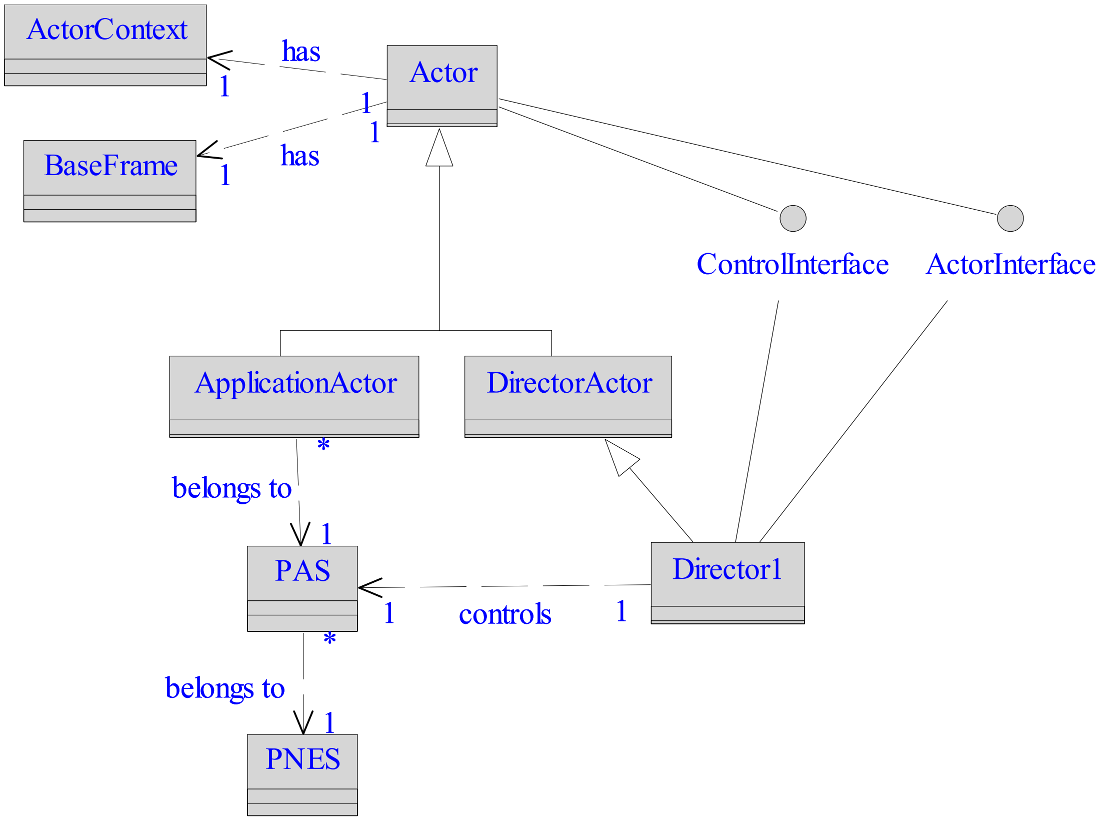


Figure 4-4 : Class diagram for the actor in TAPAS

Each actor is represented by a graphical interface which is implemented by the BaseFrame class. The ActorContext class holds the actor's context data, which are references to the classes that the actor relates to.

The actor and the related classes are shown in Figure 4-5. As can be seen the actor plays a part of a play and also keeps a reference to the play that it is currently a part of. The actor has a collection of role sessions to other actors which it communicates with in the RoleSessionCollection class and a CapabilitySet which describes the capabilities of the actor.

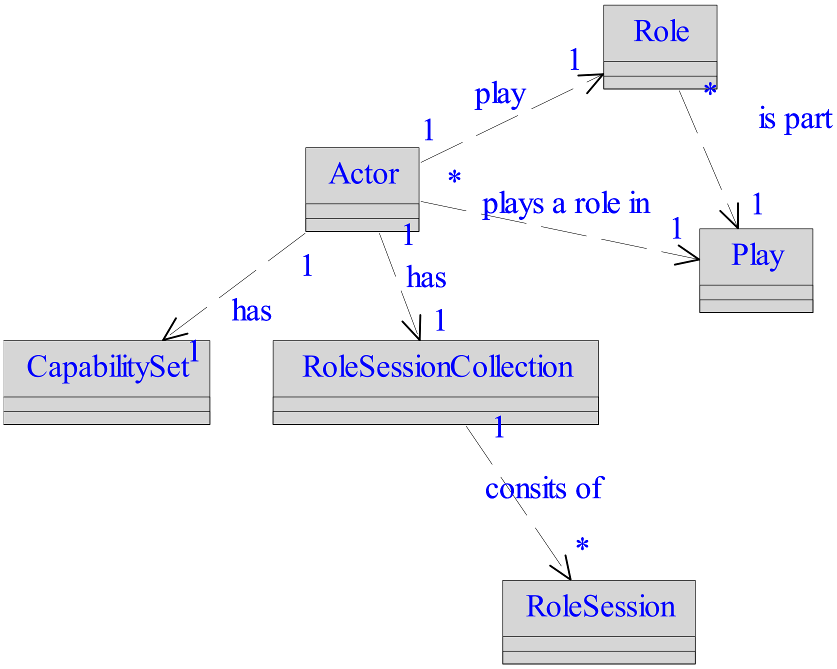


Figure 4-5 : Class diagram showing the actor relations

The director holds a ReportoireBase which contains the plays and the corresponding manuscripts for the domain which the director manages. It also has a PlayingBase which contains the playing actors in the domain. This is described in Figure 4-6 below.

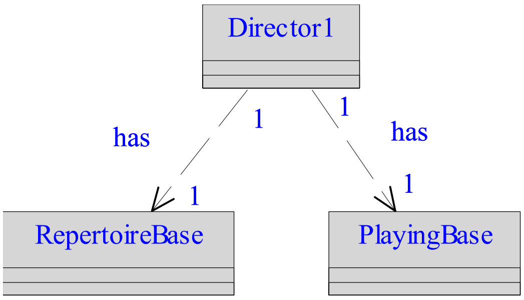


Figure 4-6 : Class diagram of director1

4.1.3 Data types

The most important data types used in the TAPAS basic support platform is shown in Figure 4-7.

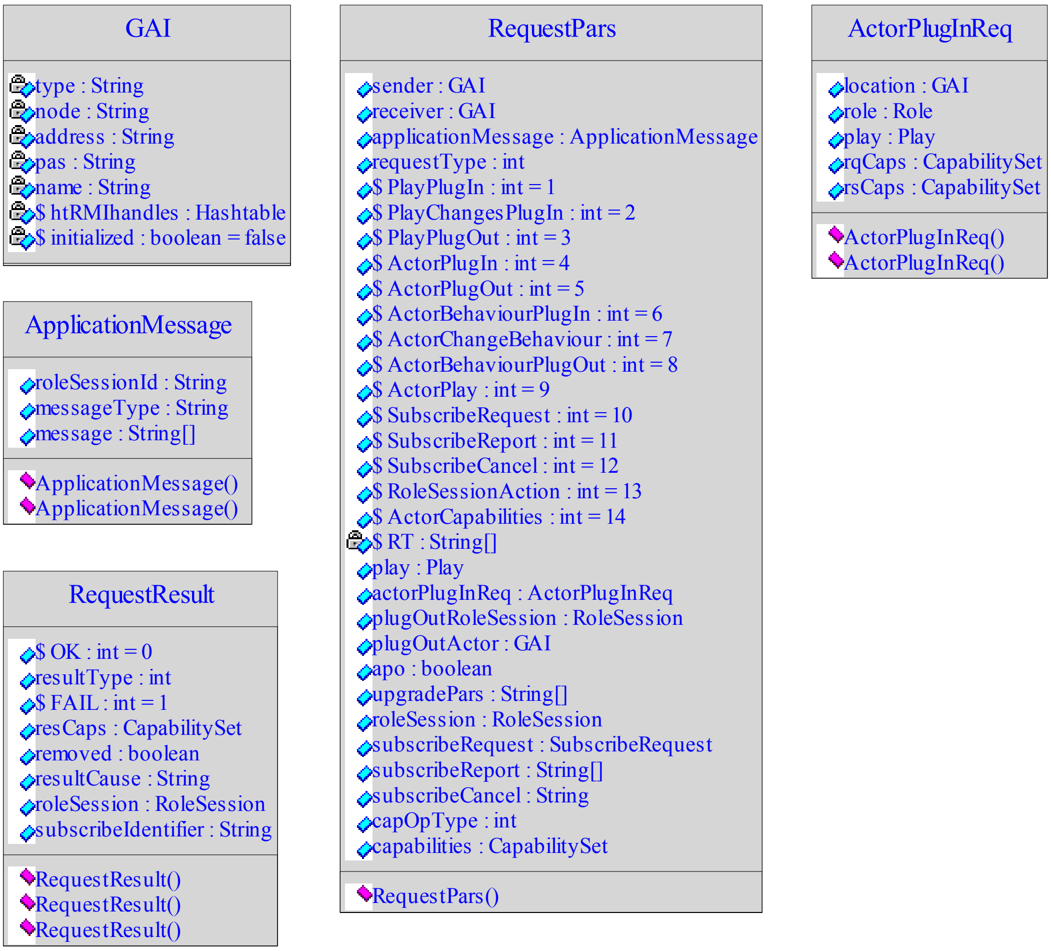


Figure 4-7 : Data types

The GAI (Global Address Identifier) class represent the location identifier of addressable entities in TAPAS. The four addressable entity types are: PNES, PAS, Actor and RoleSession. A GAI consists of several parts specific to the addressable type. A PNES instance is identified by a PNES identifier value. A PAS instance is described by a PNES identifier and a PAS identifier. Further, an actor instance also needs an actor identifier and a RoleSession instance needs all these identifiers plus a RoleSession identifier. By using this address scheme all entity instances can be uniquely identified in a global domain.

The RequestPars identifies the TAPAS request and the RequestResult contains the RequestResult. The ActorPlugInReq needs some data specific to this request and is thus implemented as an own class. The ApplicationMessage class is the message format used for communication between Role Figures.

4.1.4 Sequence diagrams

The use cases defined in the previous section can be described in more detail by sequence diagrams. The use cases are realized by requests, where the input parameter is of the data type RequestPars. The request returns a parameter of the type RequestResult, which either has the value OK or FAIL. The chain of requests is broken whenever one of the requests fails and FAIL is returned all the way back to the initiator of the first request, with information for debugging purposes. The data types; RequestType and RequestResult were described in section 4.1.3

Each use case has request chains which often consist of more than ten requests. Each use case may then have quite a few scenarios where exceptional flows are executed. To model all these possible sequences would provide a very large number of sequence diagrams and would not give a good overview of the system. Sequence diagrams are therefore only made for the main flows. When reading a diagram for a use case's main flow, it will be quite obvious what exceptional flows which may occur.

As an example the sequence diagrams for the use cases which are needed to complete the actorPlugIn procedure are described. Figure 4-8 shows the sequence diagram for the *send ActorPlugInReq* use case. As can be seen the new RoleSession started is added to the ActorContext the common use case *request to director* is performed in the next step.

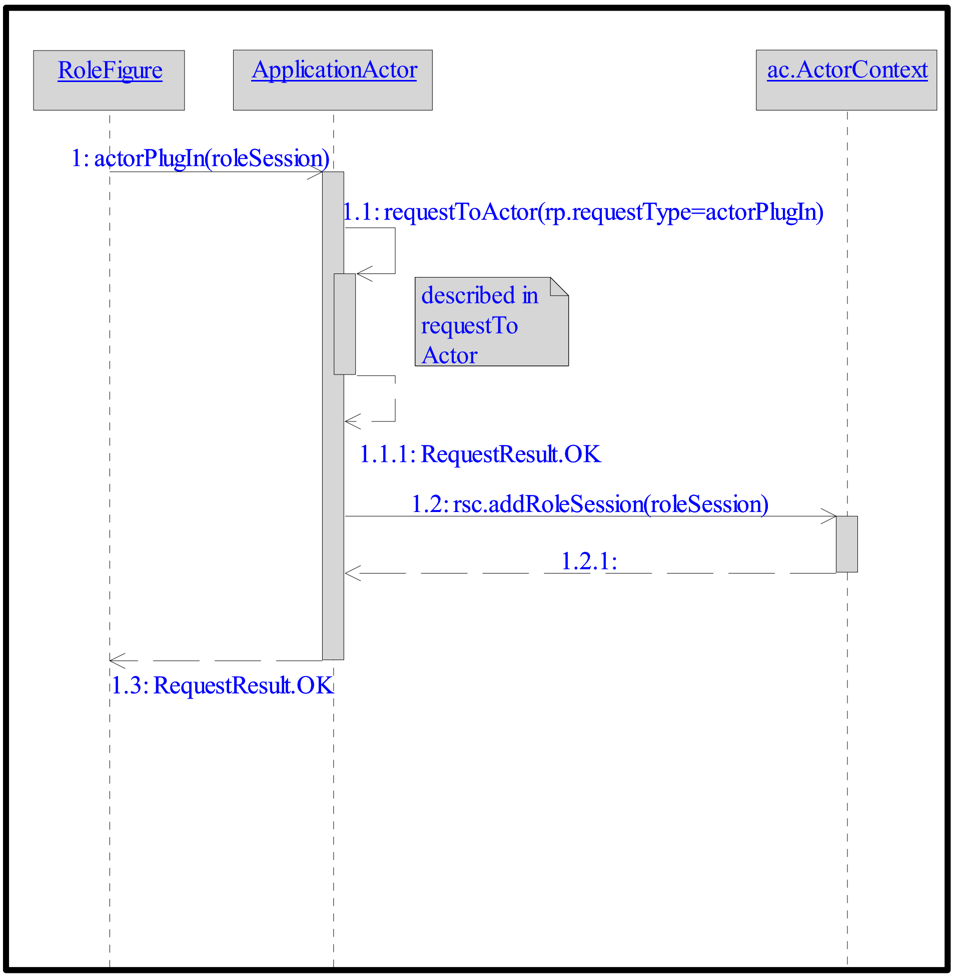

RoleFigure
ApplicationActor
ac.ActorContext
1: actorPlugIn(roleSession)
1.1: requestToActor(rp.requestType=actorPlugIn)
described in requestTo Actor
1.1.1: RequestResult.OK
1.2: rsc.addRoleSession(roleSession)
1.2.1:
1.3: RequestResult.OK

Figure 4-8 : Sequence diagram for send ActorPlugInReq

The sequence diagram for the use case *request to actor* is shown in Figure 4-9. This is the generic use case for a request to another actor. The actor to be requested in this case is the director of the domain. The request is sent to the PAS instance which the actor belongs to. If the requested actor resides in the same domain, the ActorManager is requested to get the identifier of wanted actor. If the requested actor does not belong to this PAS the request is forwarded to the correct PNES instance and then to the correct PAS instance within this PNES. In this scenario the requested actor belongs to the same PAS.

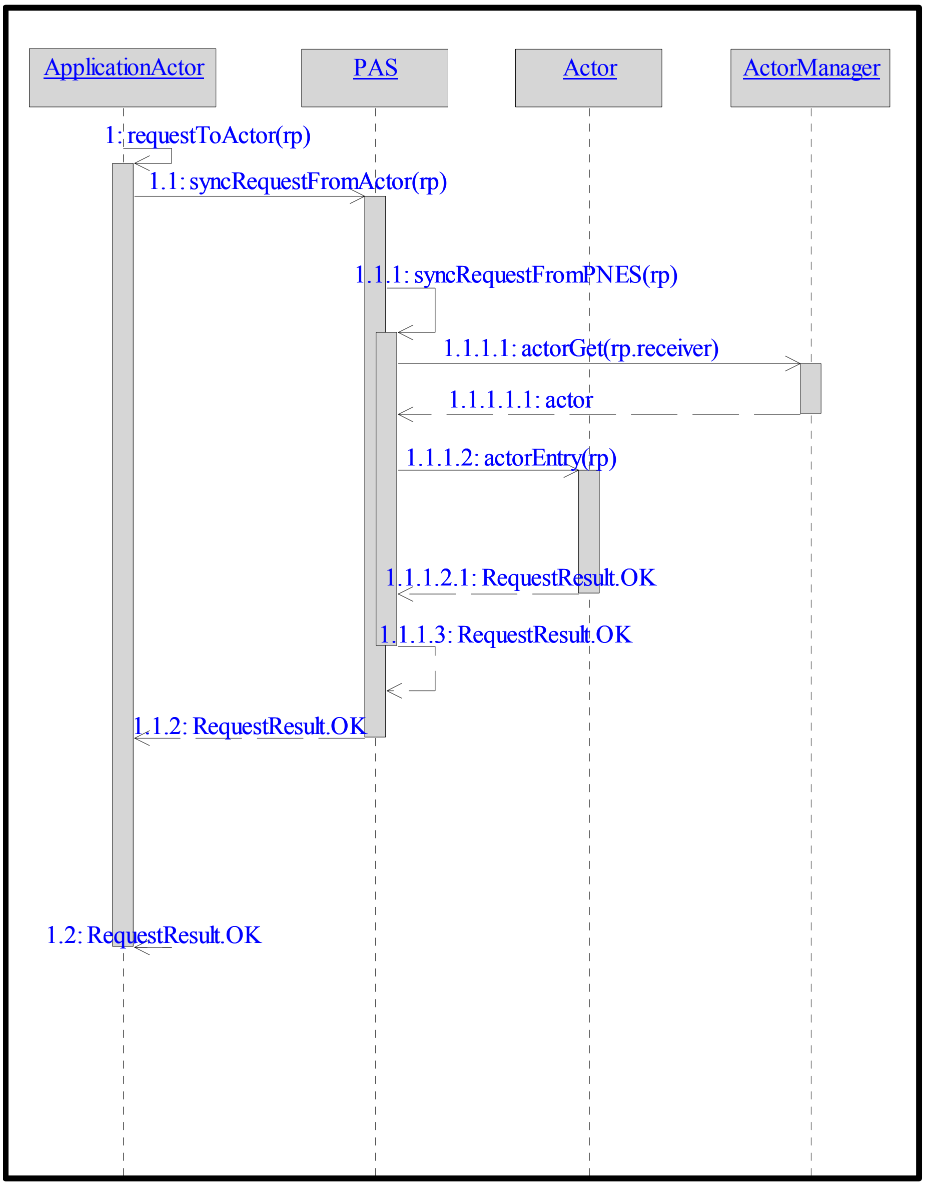

ApplicationActor
PAS
Actor
ActorManager
1: requestToActor(rp)
1.1: syncRequestFromActor(rp)
1.1.1: syncRequestFromPNES(rp)
1.1.1.1: actorGet(rp.receiver)
1.1.1.1.1: actor
1.1.1.2: actorEntry(rp)
1.1.1.2.1: RequestResult.OK
1.1.1.3: RequestResult.OK
1.1.2: RequestResult.OK
1.2: RequestResult.OK

Figure 4-9 : Sequence diagram for request to actor

Figure 4-10 shows the sequence diagram for *actorPlugIn.* In this sequence the director actor plugs the new actor into the PAS instance of its domain. The use case *create actor,* which is shown in Figure 4-11, is used to load the correct class from the web server. When the actor is loaded an instance is started and the actorPlugIn procedure is completed.

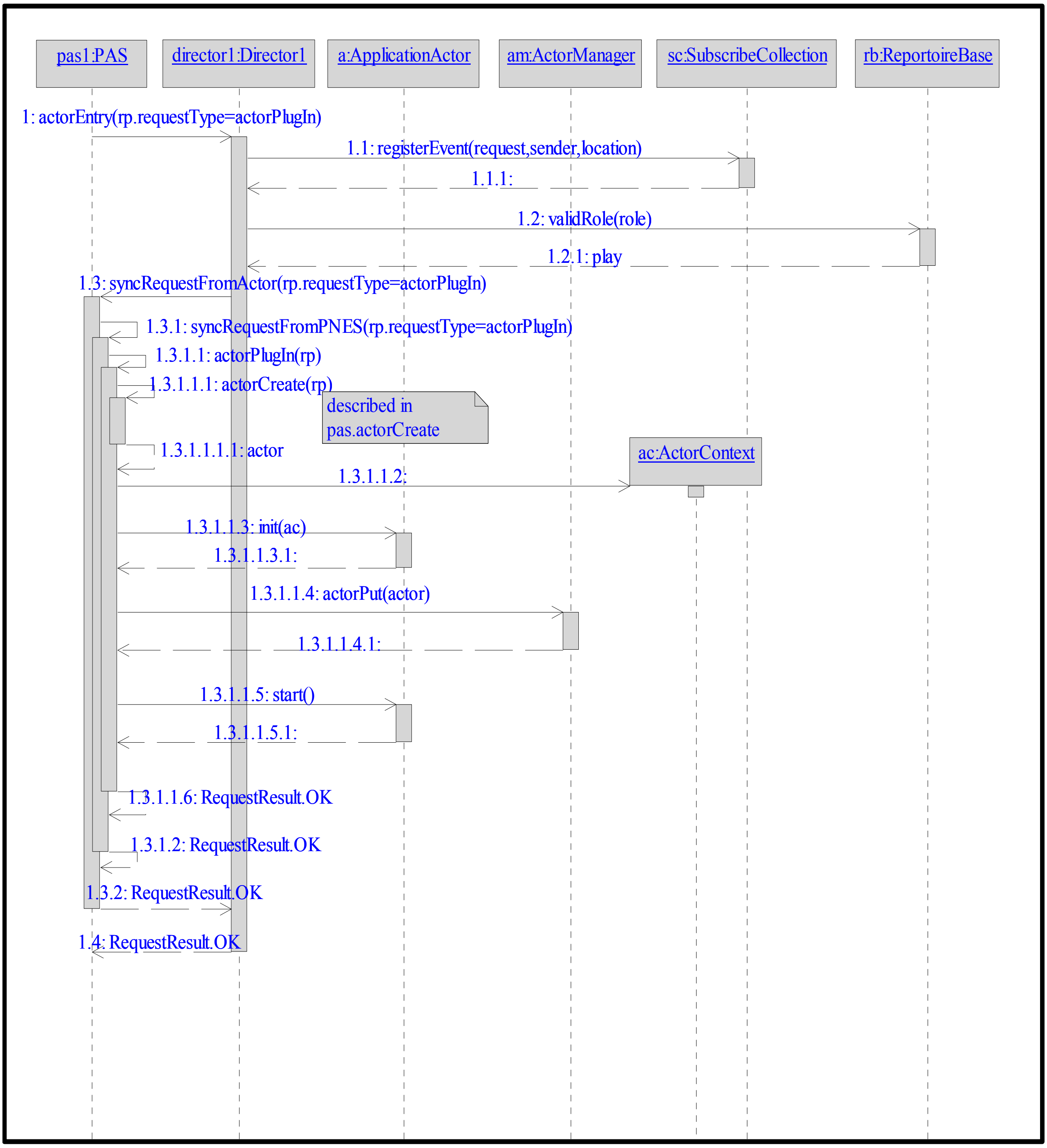


Figure 4-10 : Sequence diagram for register actor

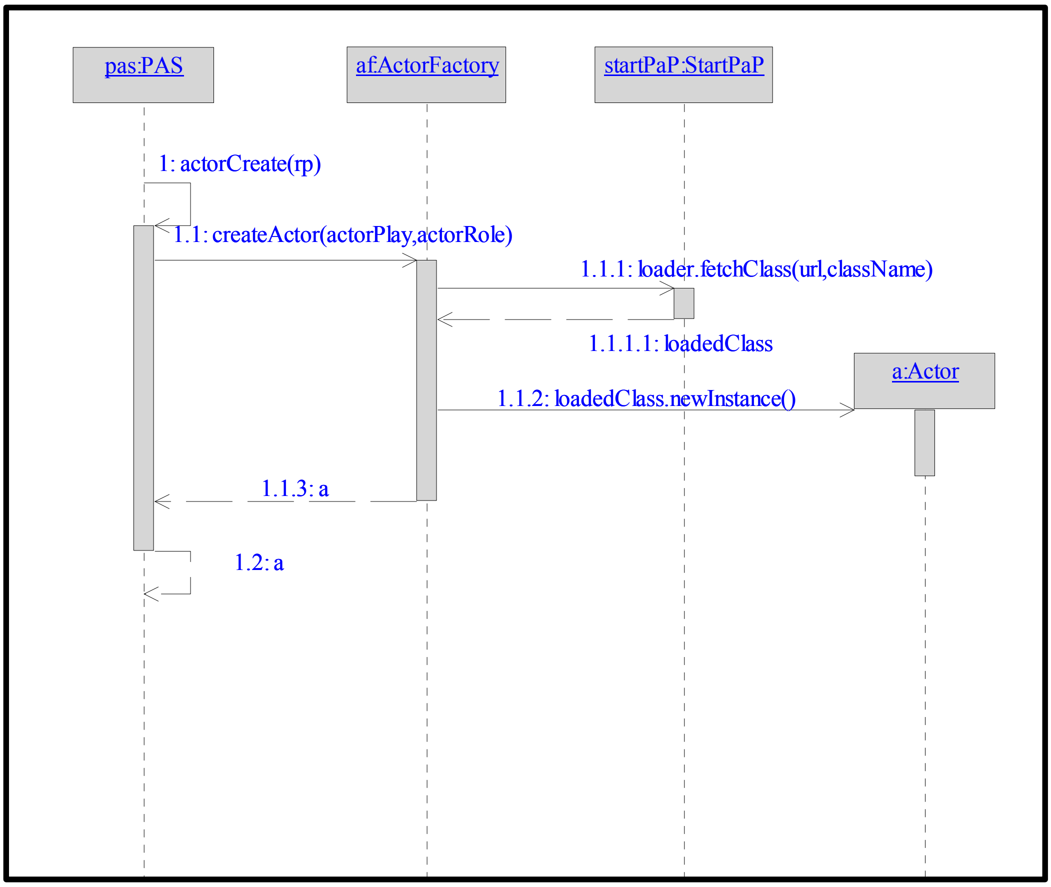

pas:PAS
af:ActorFactory
startPaP:StartPaP
1: actorCreate(rp)
1.1: createActor(actorPlay,actorRole)
1.1.1: loader.fetchClass(url,className)
1.1.1.1: loadedClass
a:Actor
1.1.2: loadedClass.newInstance()
1.1.3: a
1.2: a

Figure 4-11 : Sequence diagram for create new actor

4.1.5 Reverse engineering

The model of the TAPAS support platform is based on the existing Java prototype, and is completed by reverse engineering of the existing code. The concept of reverse engineering was explained in section 3.2.1. The reverse engineering feature of the modelling tool is actually synchronization of code from the generated files when all methods and attributes are defined in the model. This means that the model is updated with the latest changes in code within the methods from the generated files.

One effective approach to complete the model would then be to replace the generated files with the existing source files of the prototype, and then use the synchronization function to import the code into the model. This approach was however not successful. The synchronization procedure failed with 'build failed' error and no additional information was given. After a lot of time was spent on investigating the documentation and searching the internet for tips on solving the problem, still without progress, a manual approach was taken. The reverse engineering process to complete the model was performed manually by copying the code to be reused from the prototype source files into the models code fields. This process was quite time consuming, but successful. The final result is a complete model, from which executable files can be built and tested on target nodes.

4.1.6 Components and deployment

Deployment and relation between the deployed components can be described by UML component diagrams and deployment diagrams. Figure 4-12 shows the components for the TAPAS basic support platform. The PaP component contains the code for the support platform and is dependent on the externalJava component which contains the external Java classes that the support platform uses.

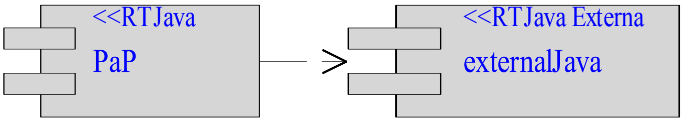


Figure 4-12 : Components of the basic TAPAS platform

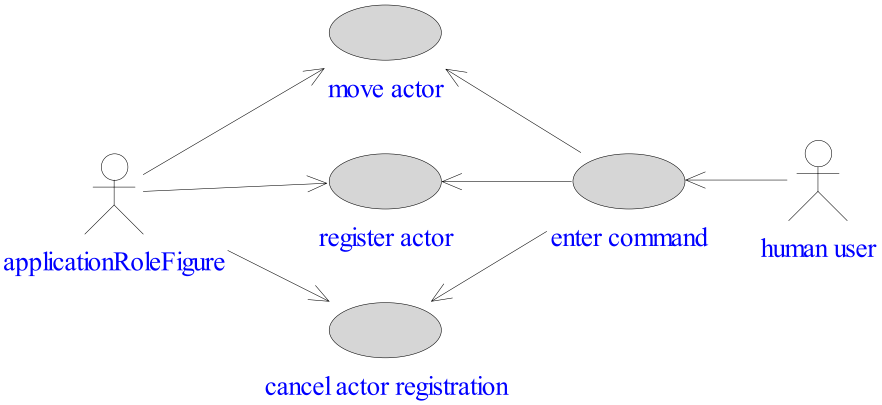
Deployment of the support platform software on the TAPAS nodes is dynamic in the way that the software is downloaded from a web server when needed by the nodes. The platform can run on all devices which have the Java J2SE runtime environment installed. Thus it does not make sense to describe the deployment of the support platform using the deployment view in the modelling tool, which focuses on exactly what processors and hardware the software components are deployed on. It can however be useful to use the deployment diagrams to describe some scenarios on what components of the TAPAS applications which are running on the different TAPAS nodes. This will be discussed in the modelling of the applications later on.

4.2 The MicroTAPAS model

To be able to run TAPAS on wireless devices with limited recourses a downsized version of the basic TAPAS platform was developed and a prototype made in [7]. Later mobility support for this platform was added [6]. This chapter will give a short description of the MicroTAPAS platform and the supported mobility functions and describe how an UML model is made.

4.2.1 Use Cases

Use cases for the MicroTAPAS platform are quite similar to the use cases for the TAPAS basic support platform, since the same TAPAS functions are implemented in this platform as the previous one. In this section focus will be on the mobility functionality introduced in the MicroTAPAS mobility extension package. The new use cases for mobility are summarized in Figure 4-13.



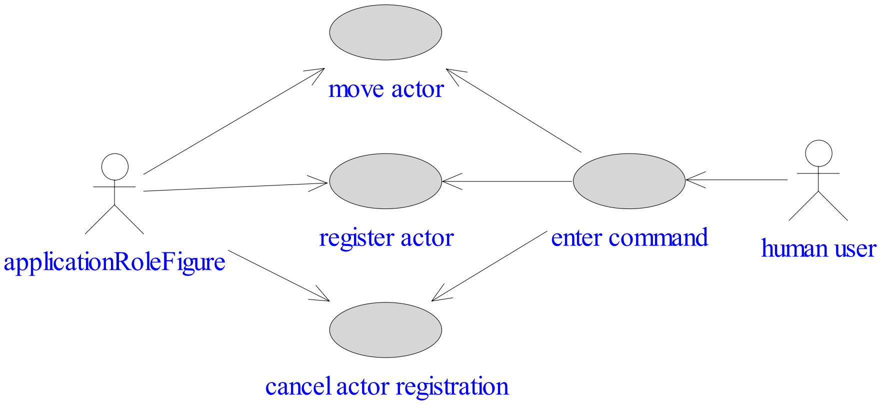


Figure 4-13 : Mobility use cases in MicroTAPAS

The move actor use case describes the actorMove procedure in the TAPAS mobility architecture. The use cases register actor and cancel actor registration are used to register the actor with the mobility manager at a new domain, and cancel the registration in the old domain, when the terminal is moved.

4.2.2 Class diagrams

The MicroTAPAS support platform differs from the original TAPAS support platform in that the PAS and PNES layers are merged into one PNES layer, and that sockets are used instead of Java/RMI as communication between nodes. Figure 4-14 describes how the MicroPNES class uses sockets to communicate with other instances of MicroPNES residing on other nodes. The ComCenter class offers methods to send and receive requests from other nodes. A ServerSocket instance is created to listen for incoming requests and create a socket when a request is to be sent. The ConnectionHandler class is responsible for one connection while a request is handled by the MicroPNES instance, so that the response can be returned to the same socket from which the request was received.

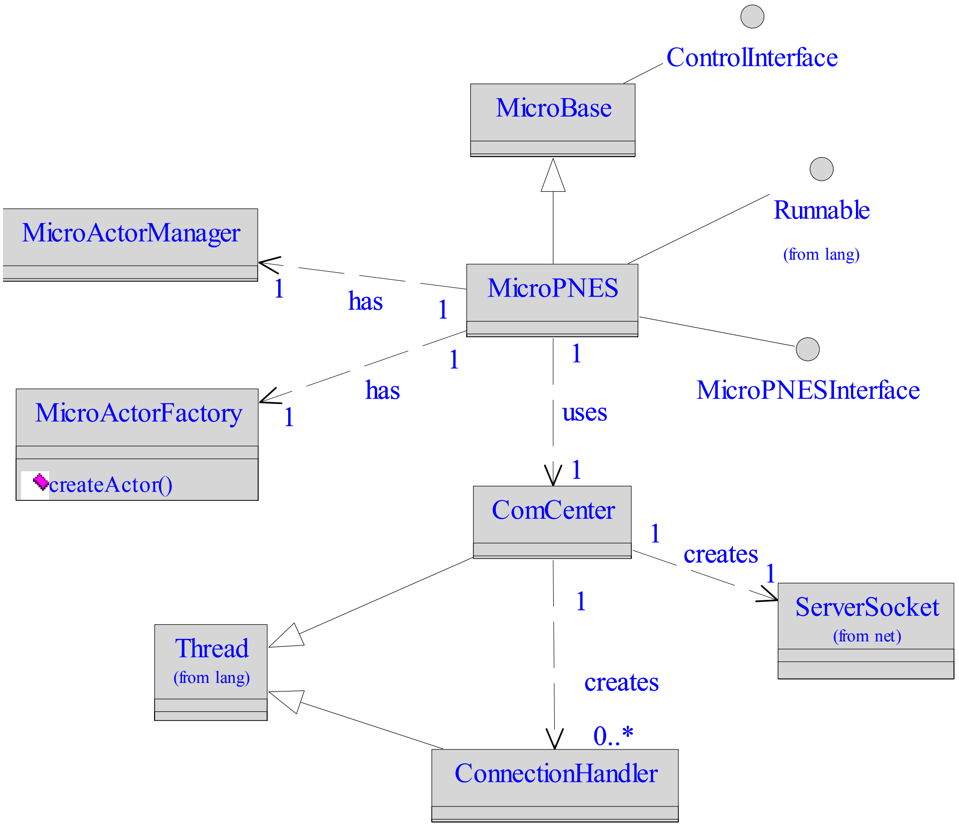


Figure 4-14 : Class diagram for MicroPNES and its supporting classes

The realization of the actor entity is in principal the same as for the original TAPAS support platform. Figure 4-15 describes the realization of the MicroActor.

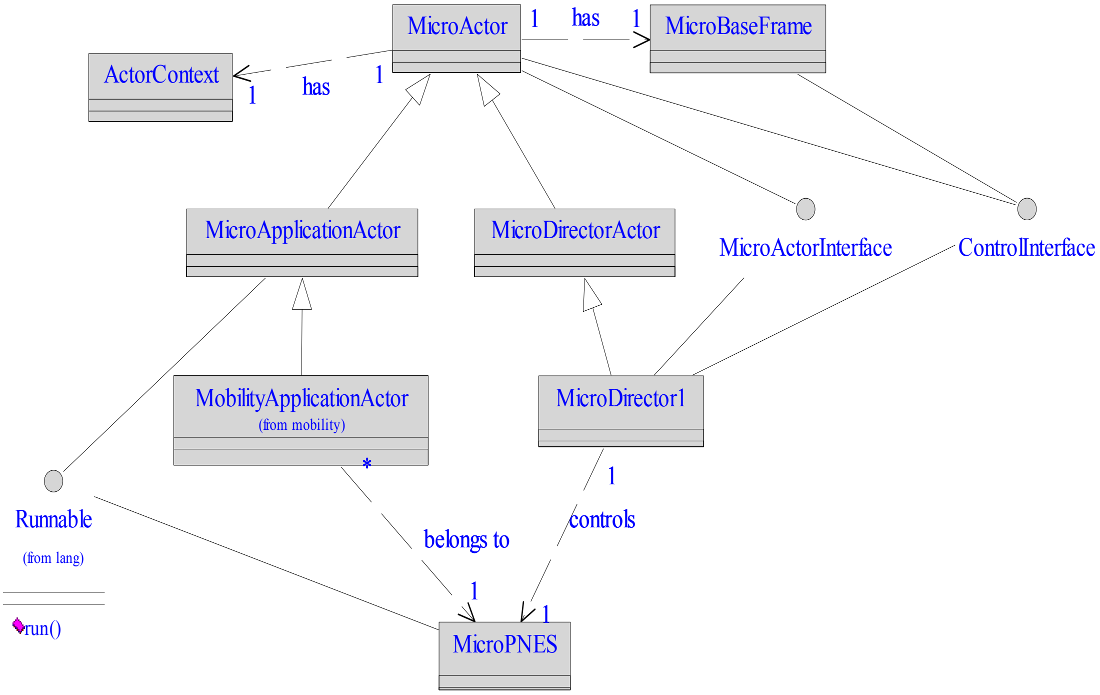


Figure 4-15 : Class diagram for the MicroActor

The main classes of the mobility support functionality are shown in Figure 4-16. The two main entities of the mobility extension are the MobilityAgent and MobilityManager [6]. These to classes contain some similar functionality which is put in the class MobilitySupportActor. MobilityAgent1 and MobilityManager1 inherit this class, which is an extension of the MicroActor class.

The MobilityManager and the MobilityAgent are responsible for handling of the mobility related administration functions. Such functions are: keeping track of actors and terminals, facilitate in ActorMove and TerminalMove operations and monitoring connection status of the actors to the MobilityManager. Where the ActorMove and TerminalMove operations are used to move the actor to a new location and move a terminal to a new domain [6].

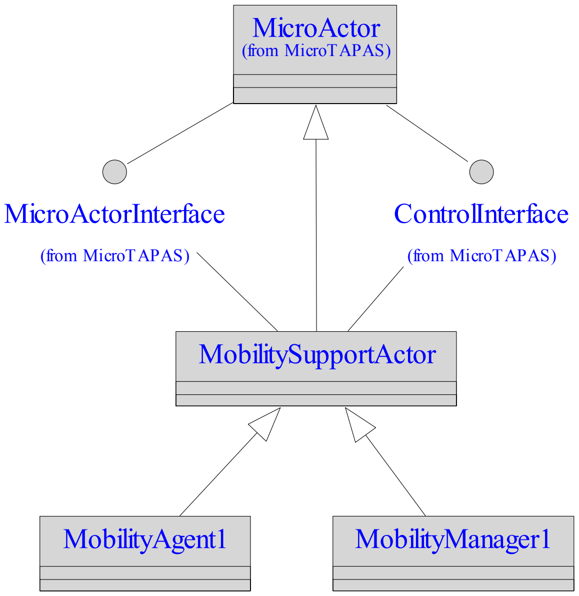


Figure 4-16 : Mobility extensions for MicroTAPAS

4.2.3 Sequence diagrams

An example of a sequence diagram for the use case actorMove is shown in Figure 4-17. This diagram has a higher level of abstraction than the ones presented in the previous section, which were on method level. This can be done because the procedures used are described in details in the description of the basic support platform. Moving of the abstraction level provides a better overview of the functionality.

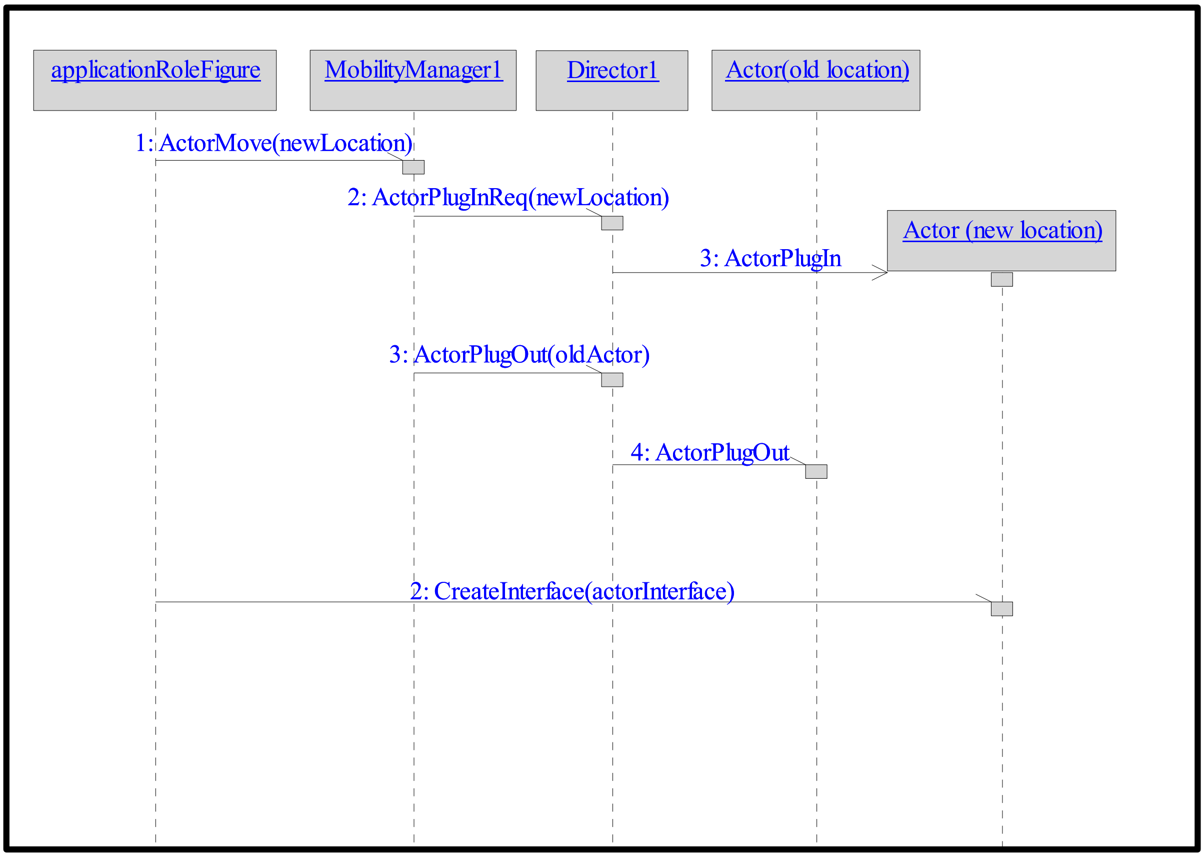

applicationRoleFigure
MobilityManager1
Director1
Actor(old location)
1: ActorMove(newLocation)
2: ActorPlugInReq(newLocation)
Actor (new location)
3: ActorPlugIn
3: ActorPlugOut(oldActor)
4: ActorPlugOut
2: CreateInterface(actorInterface)

Figure 4-17 : Sequence diagram for the use case ActorMove

4.2.4 Reverse engineering

As for the TAPAS basic platform this model is completed with reverse engineering of code from the Java implementation.

4.2.5 Components and deployment

The MicroTAPAS software component is dependent on the external Java functions in the externalJava component as described in Figure 4-18.

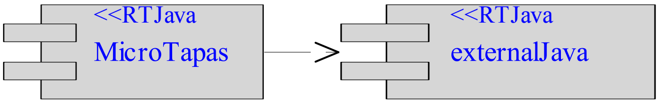


Figure 4-18 : Component diagram for MicroTAPAS

4.3 Model of the Tele School application

To learn how an application is built on top of the TAPAS support platform, a UML model of the Tele School example application will be made. By modelling an existing application, the type of diagrams best suited for modelling of a TAPAS application can be found. This model will show how the role figures of the Tele School application best can be modelled, and will be used to form a template for development of future role figure models. The template will then be used in the development process of the TAPAS applications.

When the role figure model for the Tele School application is created, it can be used to learn how the UML based role figure model can be translated to Java code and XML. A mapping procedure will be defined for translation from the platform independent role figure model to a UML model which can generate Java code. The following sections describe how the Tele School application is modelled using the use case driven approach described in section 3.4.1.

4.3.1 Use cases

The application is a learning center which teachers and student can log on to and get access to several services. The use cases that can be identified for the Tele School application is shown in Figure 4-19.

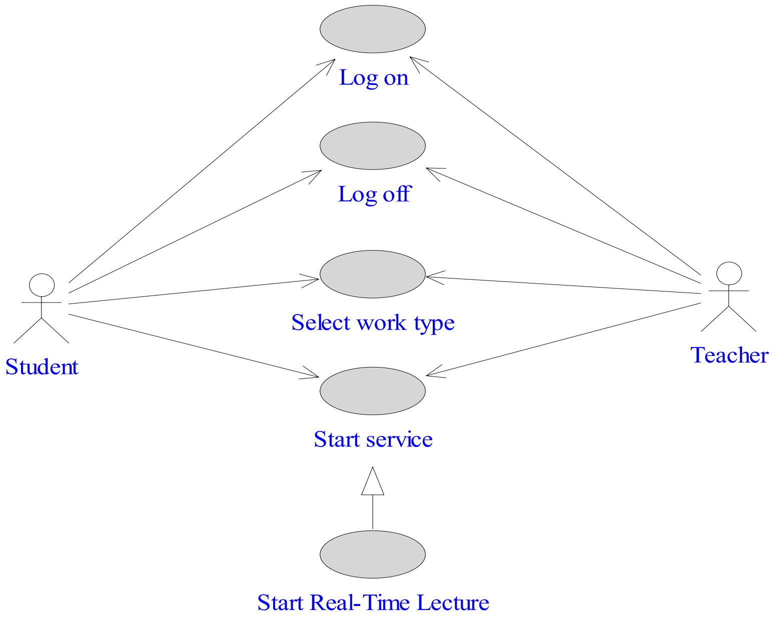


Figure 4-19 : Use Case diagram for the Tele School application

4.3.2 Class diagrams

The Tele School play is realized by four actors playing four Roles. The SchoolClient and the SchoolClientInterface actors represent the client which will be located on the user's terminal and the client's user interface. The SchoolServer actor handles management of the users and the ScoolRTLServer is a server used for the Real-Time Lecture service.

Role	Description
ShoolRTLServer	Provides functionality specific for real time lectures.
SchoolServer	Defines the behaviour of the server for all clients running TeleSchool.
SchoolClient	Defines the behaviour of for the users of the application i.e. students and teachers.
SchoolUserInterface	Presents the user interface for the students and teachers.

Table 4-1 : Role Figures in the Tele School application

A description of the Roles in Tele School is summarized in Table 4-1. A high level class diagram for the application can be found in Figure 4-20.

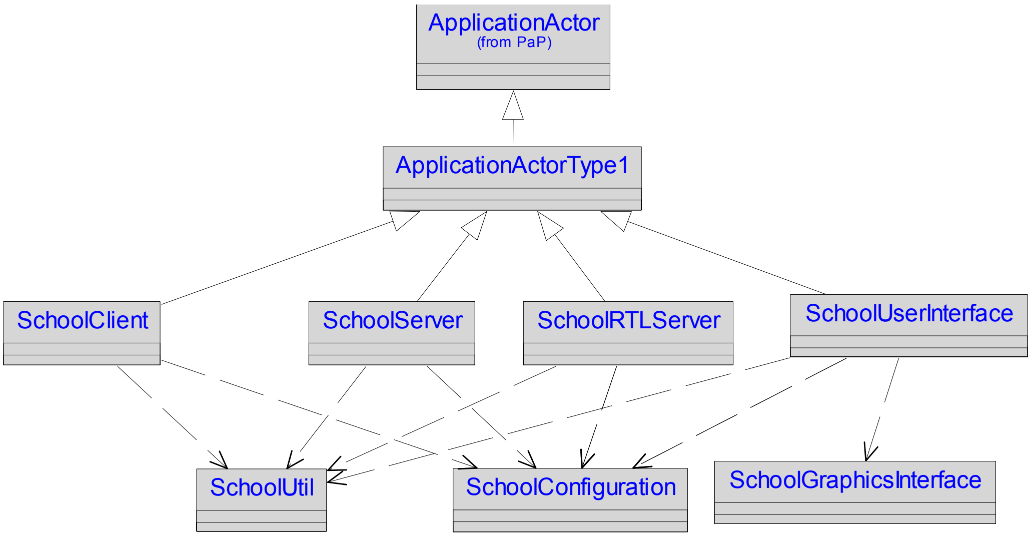


Figure 4-20 : Class diagram for the Tele School application

4.3.3 Sequence diagrams

The interactions between the actors in TAPAS are realized by asynchronous message exchange. The ApplicationMessage class defines the message format and messages are sent in RoleSessionAction requests to the TAPAS support platform, which handles all requests in a separate thread of execution. Sequence diagrams can be used to describe how the actors interact and what TAPAS support functions are executed. As an example of a sequence diagram the use case Log on from Figure 4-19 is described in Figure 4-21. In this diagram the calls to the TAPAS support functions are modelled as synchronous function calls while the messages between the actors are modelled as asynchronous messages.

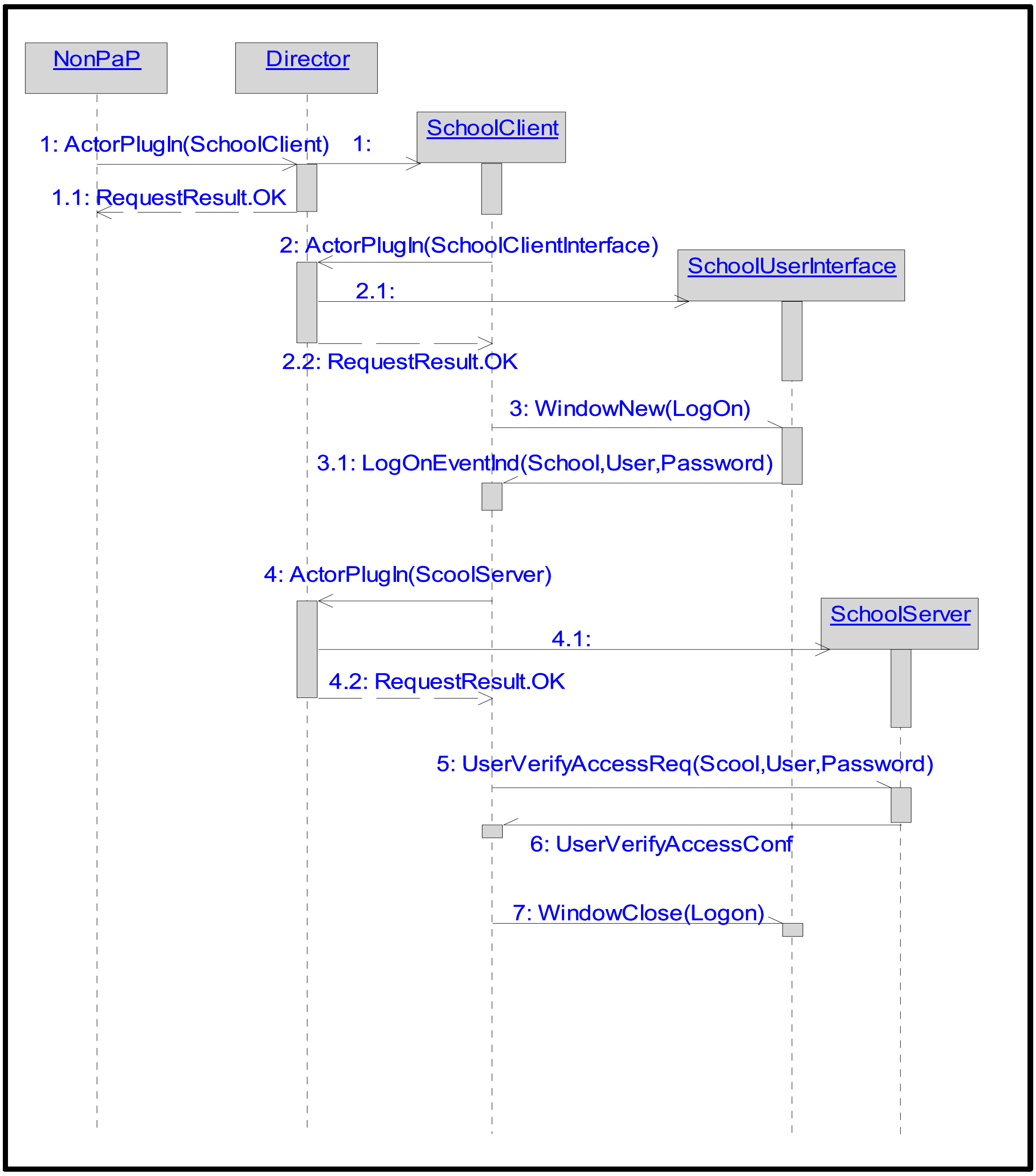


Figure 4-21 : Sequence diagram for the Tele School application

4.3.4 State diagrams

The behaviour of the Role Figures can be described by UML state diagrams. Figure 4-22 shows the behaviour of the SchoolClient Role Figure. The state diagrams in UML are state oriented, which means that they are based on showing states and the actions which triggers a transition to a new state.

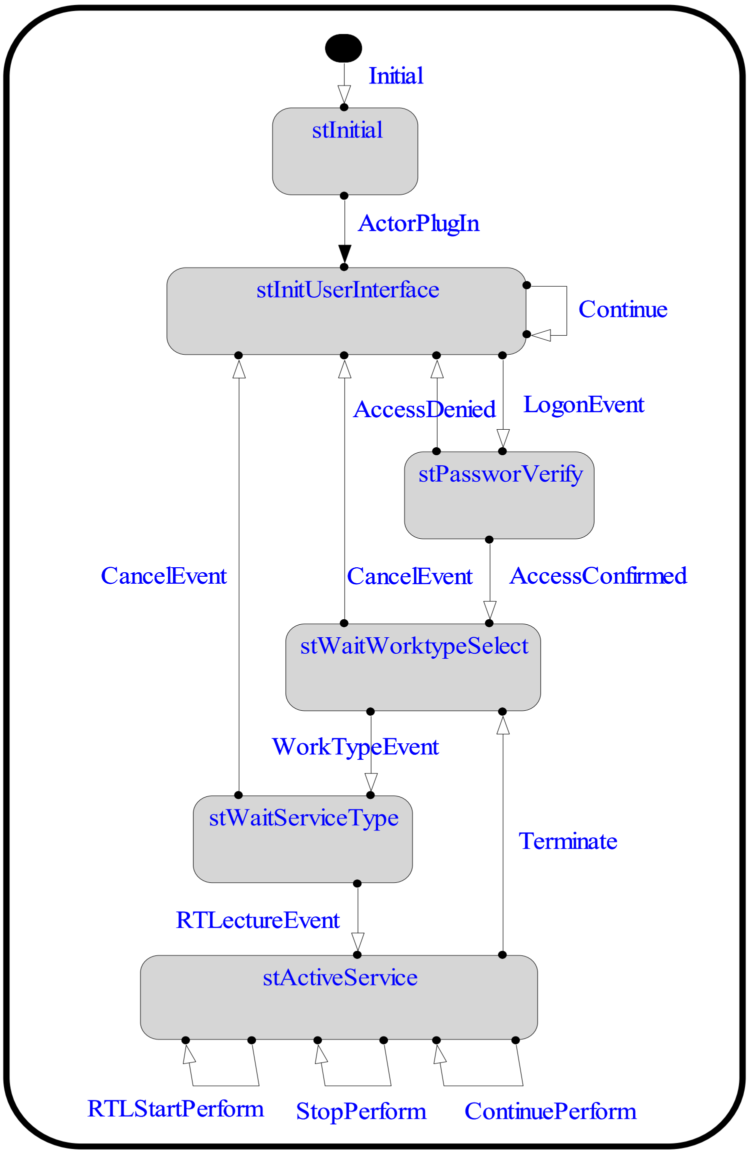


Figure 4-22 : State diagram for the Tele School application

4.3.5 Reverse engineering

As for the models described in the previous sections, the model of Tele School application is completed using reverse engineering.

4.3.6 Components and deployment

A component diagram for the Tele School application is shown in Figure 4-23. The School component depends on the PaP and externalJava components.

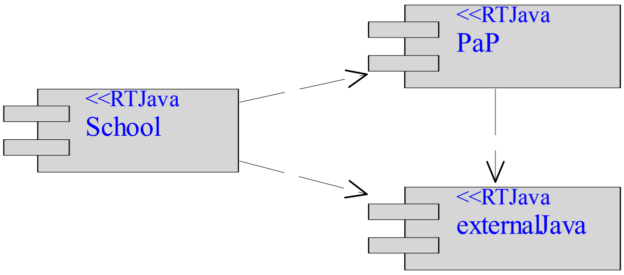


Figure 4-23 : Component diagram for the Tele School application

UML deployment diagrams, described in section 3.1.1, can be used to describe how the application is intended to be deployed when the actors are distributed on the TAPAS nodes. Figure 4-24 shows how the Tele School application may be deployed. The diagram shows how the actors are distributed on the TAPAS nodes.

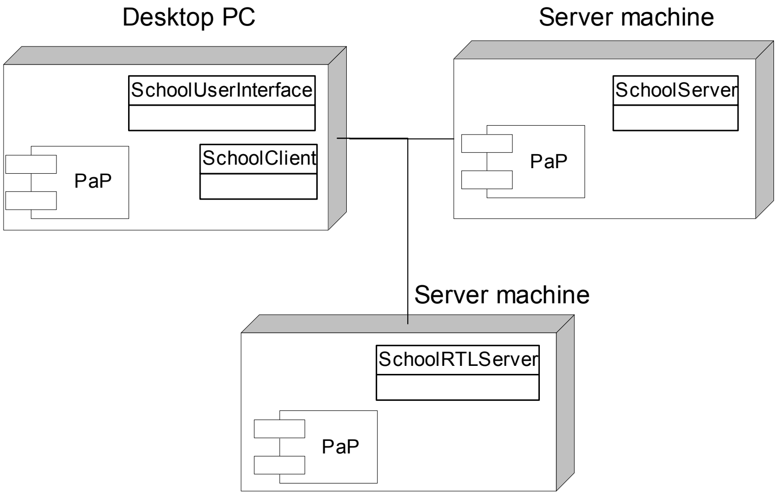


Figure 4-24 : Deployment diagram for the Tele School application

4.4 Summary

In this chapter the UML models made for the basic TAPAS support platform, the MicroTAPAS platform and the Tele School example application have been described. The models made for the support platforms consist of use case diagrams, sequence diagrams, class diagrams and component diagrams. In the model of the Tele School application state diagrams are used to model the behaviour of the role figures and deployment diagrams are used to describe how the actors are distributed on different nodes. The models are completed by reverse engineering the code from the existing prototypes.

5 Role Figure model template

As the demand on short time to market is getting stronger for introduction of new services and applications, the requirements for an effective and reliable development environment is increasing. For the application developers to start using a platform like TAPAS and to succeed in developing applications, the descriptions of how to use it must be good, and the possibilities for rapid development must exist. Usually, there is no need to know all details on what is going on ‘under the hood’ to develop successful applications. By introducing a template which abstracts the details of the platform for the application developer, focus can be held on the design of the application itself and not so much effort made to understand how the platform works. UML is a modelling language that most software designers is familiar with and will be used to model the applications. To provide a complete development environment the modelling tool Rational Rose Real-Time is used. The features of the tool where described in chapter 3.

In this chapter a template for application development in TAPAS is presented based on the results of the previous chapters. Software Engineering is a wide topic and a wide range of processes are described for the software development. It is not within the scope of this thesis to develop or to integrate on of the extensive and well known processes available. The steps for development of TAPAS applications presented in this chapter are based on the use case driven approach described in section 3.4.1.

The following sections will propose some steps for developing applications on the TAPAS platform using UML, based on a set of template diagrams for the design phase and design patterns to be used when implementing the application in Java or XML.

5.1 Specification of requirements

The two following sections describe the specification of requirements for the TAPAS application. The requirements are divided into functional and non functional requirements.

5.1.1 Functional requirements

The first step in the system development is to specify what the system shall be able to do. In TAPAS the application running on top of the support platform is referred to as the *play* (see section 2.1) and at this stage the play is specified. Usually some vague idea exists on what the system shall do and who the potential users will be. By starting with the use case specification the designer of the system will be able to specify the users, which in UML is denoted actors, and the functionality of the system. This will include functional requirements only. The UML diagrams are useful for communication among the designers and between the designers and the users of the system. At this stage it is important to get a common understanding of the system functionality.

The Use Case diagram shall be used to specify the functional requirements at a high level of abstraction. The result is a high level description of the *play*.

5.1.2 Non-functional requirements

Usually, some non-functional requirements exit for the application. These requirements will be depending on the implementation which at this stage is some steps ahead. However, requirements need to be specified as early as possible in the process and non functional requirements may be essential for the application. Examples on typical non functional requirements are:

- Usability – requirements for the user interface etc.
- Performance – requirements for access time, system performance etc.
- Security – security functions required.

To specify the non functional requirements a textual description shall be given. How to fulfil these requirements is not further describe in this template and some of them can only be proved to be satisfied by testing on the target system. However, it is important that they are kept in mind during the design and implementation phase, because it is important that the decisions made in these steps are influenced by the non functional as well as the functional requirements.

5.2 Specifying the Role Figures

In TAPAS the service will be realized by a set of Role Figures constituted by actors playing certain roles. The next step in the development cycle will be to specify a set of Role Figures to realize the play which was specified by the use cases in section 5.1.1. When the Role Figures are found a class diagram is used to describe the Role Figures most important attributes and the support classes that will be needed. At this stage the actual behaviour of the Role Figure is not to be considered, just the main attributes and the other classes that it relates to. The following sections describe the two steps of specifying the Role Figures. The first step is to figure out which Role Figures we need and the second one is to specify them using a class diagram.

5.2.1 Finding the Role Figures

The TAPAS platform is developed with grate flexibility in mind and it is up to the application designer to decide how the play shall be realized. The Role Figures to be specified will be able to run on all TAPAS nodes. TAPAS nodes may vary from powerful servers to small wireless devices, which leave the designer of the application with possibilities for developing a wide range of applications. At this step focus shall be on finding the actors needed to realize the play.

It shall also be decided if the actors need some of the mobility features implemented in the MicroTAPAS platform. The concept of terminal and role figure mobility is described in section 2.2.2. When the actors are found and the support platform to be used is decided the next step will be to describe the actors. To be able to start quickly and keep focus on the application functionality, template classes are made which can be used when starting the static design of the application.

5.2.2 The ApplicationRoleFigure template class

In section 2.2.1 the Role Figure, of the type to be used in an application, is referred to as the ApplicationRoleFigure. For a description of how the ApplicationRoleFigure is realized the Tele School application is a good example. In section 4.3.2 the class diagram for the Tele School application shows how the ApplicationRoleFigures are realized by inheritance of the actor class. In this example the ApplicationRoleFigures are not directly extensions of the actor class but of the ApplicationActor1 which is a special version of ApplicationActor with some additional functionality. The ApplicationActor is then the type of actor which is used to participate in the play. The other type of actor is the Director. The relationship between the different kinds of actors can be seen in Figure 2-1 for the basic support platform and in Figure 4-15 for the MicroTAPAS platform.

To summarize, the ApplicationRoleFigure inherits the type of actor that is required for the application. If the basic support platform is to be used the ApplicationActor class shall be extended. If the MicroTAPAS platform is to be used the MicroApplicationActor needs to be extended and if mobility functions are required the MobilityApplicationActor described in Figure 4-16 shall be used.

The first template class to be made is the one for the basic support platform and is shown in Figure 5-1. As can be seen in the figure the ApplicationRoleFigure inherits the TAPAS functions from the ApplicationActor class which was described in section 2.2.1, and can use these functions by calling its super class. The ApplicationRoleFigure probably need some supporting classes as well. This may be classes to realize user interface, file access etc. The support classes shall also be specified in this class diagram and is denoted ApplicationRoleFigureSupportClass in the template below.

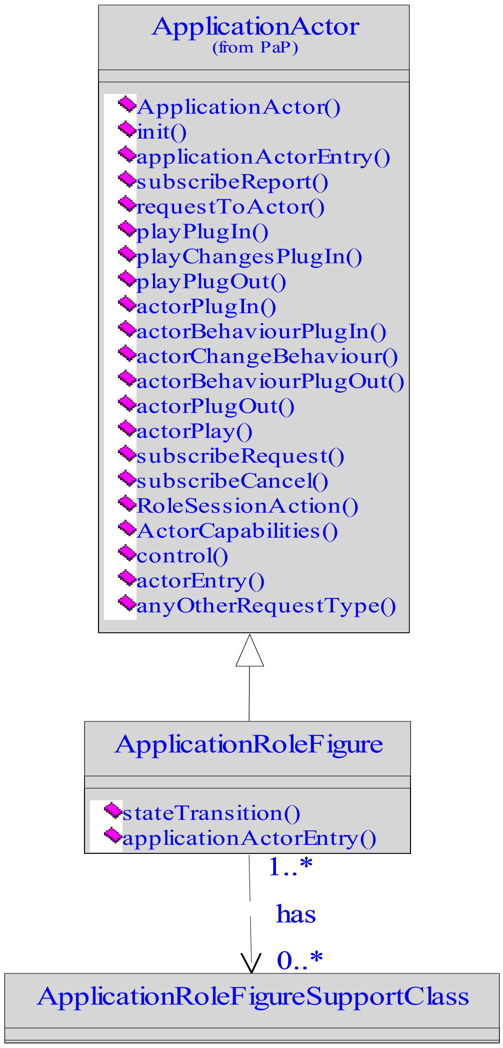


Figure 5-1 : Template classes for the Role Figure

As for the TAPAS basic support platform a template is made for the MicroTAPAS platform. The template is shown in Figure 5-2 and is quite similar to the one in Figure 5-1. The reason is that there is actually not any new functionality in this platform. The platform will however enable TAPAS for small wireless devices which increases flexibility and may be the whole idea that the application to be developed is founded upon.

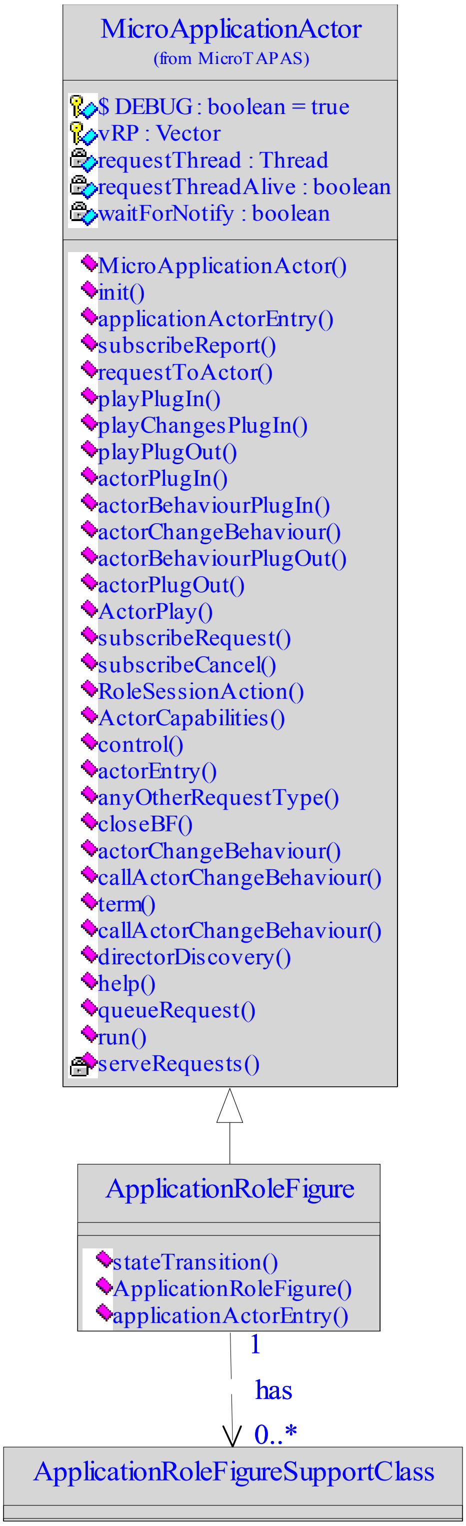


Figure 5-2 : Template class for the MicroTAPAS platform

The last template is the one which is based on the mobility features added to MicroTAPAS. The template described in Figure 5-3.

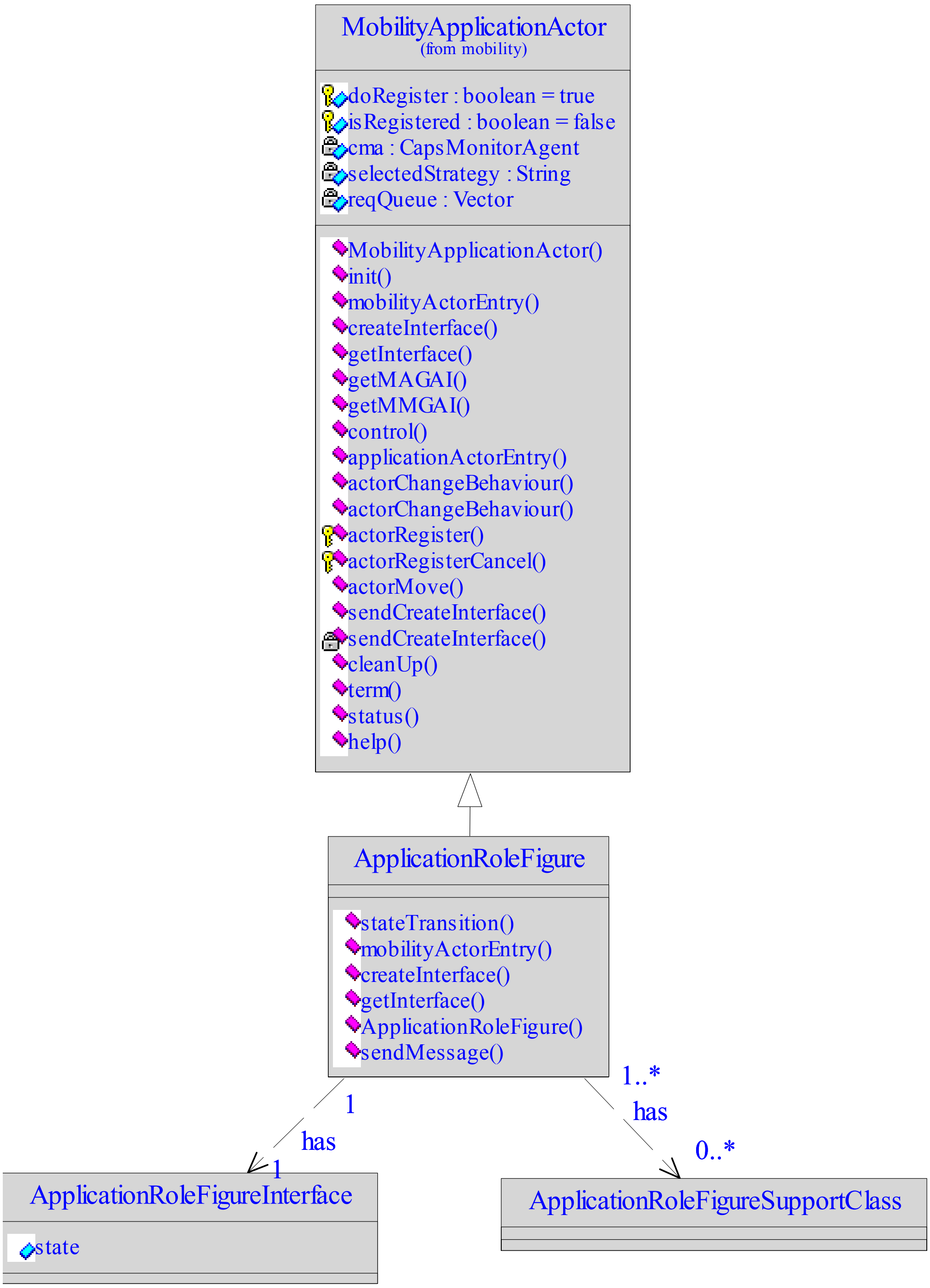


Figure 5-3 : Template classes for the Role Figure to be used with mobility extensions

As can be seen in the figure the ApplicationRoleFigure extends the MobilityApplicationActor and inherits methods for the mobility functions such as actorMove() which is used in the procedure of moving the Role Figure to a new terminal. In this template the ApplicationRoleFigure also has the interface class ActorRoleFigureInterface which is a class containing data to be kept and used for recreation of the actor at a new location. Such data may for example be the state of the Role Figure and other application specific data needed for the recreation.

5.3 Description of the functionality

When the Role Figures are specified the next task will be to specify how they shall interact to realize the complete service behaviour. The functionality is described by sequence diagrams describing each use case. The sequence diagrams shall describe the main flow, alternative flows and exceptional flows of the use case. Figure 4-21 shows the main flow of the use case Log_on for the Tele School application. The diagram only shows the exchange of application messages and the TAPAS functions. It is a good idea to keep the abstraction at that level to avoid getting to complex sequence diagrams. At this level the most important thing is to describe the interactions between the actors.

5.4 Behaviour specification

When the classes to be used and the functionality are specified, the behaviour of the Role Figures involved in the play is to be described.

5.4.1 The ApplicationRoleFigure state machine

In TAPAS behaviour of the actors are described as finite state machines. As for the model of the Tele School application in section 4.3.4 the state diagram is the UML diagram best suited for this purpose. First a state diagram is made for the Role Figure. Figure 5-4 shows the template to be used for the state diagram of the ApplicationRoleFigure class.

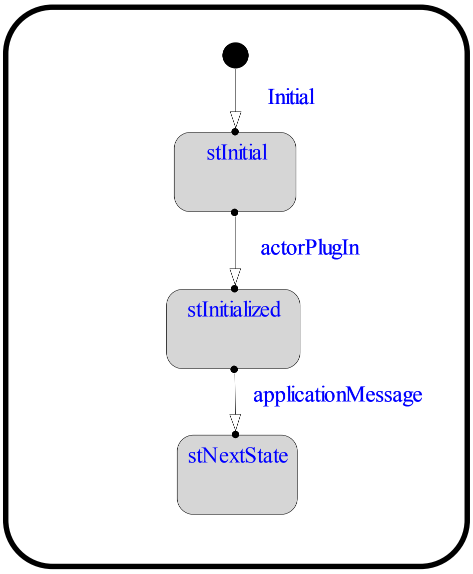


Figure 5-4 : Template for the Role Figure state diagram

The required behaviour for the Role Figure can be found be analysing the sequence diagrams where the Role Figure is involved. By analysing the reaction on each message reception a state diagram can be made, describing the complete behaviour. The diagrams in the tool used in this thesis work does not show the messages sent in the state diagram which would give a better overview, but the action taken on the message reception can be found in the model by clicking on the arrow in the diagram. The window that appears also has a filed to describe the action by text or to fill in the actual code.

State diagrams for the Role Figures using the mobility feature in MicroTAPAS can be made on the template in Figure 5-5 below. Here a check is added to find out if this actor is moved when it is plugged in. If it is moved it shall stay in the initial state and wait for the interface to be created. The actorMove procedure is described in Figure 4-17.

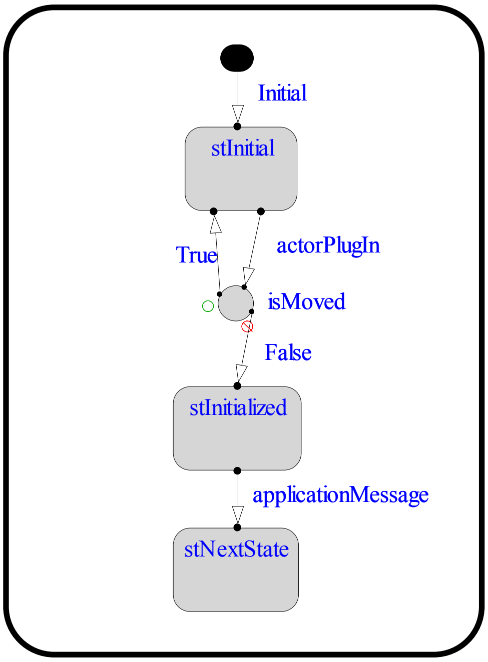
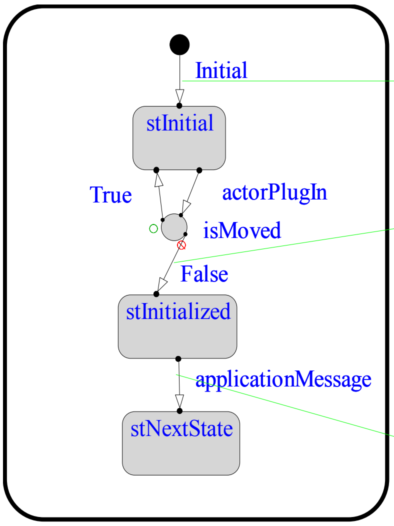


Figure 5-5 : Template state diagram for mobility

5.4.1.1 Mapping from state diagram to Java code

Since a number of UML elements are supported by the object oriented languages, generation of code is relatively straight-forward. Generation of code for the dynamic part of the UML model to Java is not as easy as for the static part. As discussed in section 3.1.2, the main problems are that UML does not have a unified diagram which describes behaviour. Another problem is that the concepts from these diagrams are not supported by Java. To implement the behaviour described in the state diagram, code must be written as for the other methods in a traditional object oriented manner.

The method stateTransition in the ApplicationRoleFigure class is the method which implements the behaviour of the Role Figure. The mapping of from the state diagram to a code template is described in Figure 5-6. Each time a message is received in a RoleSessionAction, the stateTransition method is called and the transition to be taken is depending on the current state and the message type.

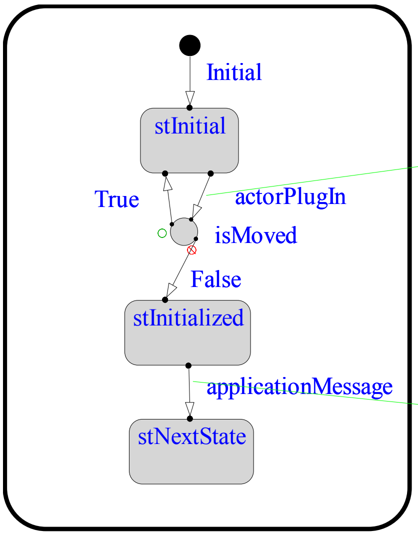


```
switch(ai.state){

    case stInitial:
    if(ActorContext.microPNES != null){
    context.play.playLoc+context.play.playId, context);
            initialRoleSession = (context.rsc.initialRoleSession());
            rr = new RequestResult(RequestResult.OK, "ActorPlugIn was successful", pRP.ID);
                        .
                        .
            if(pRP.isMoved){
        // Stay in this state, while waiting for createInterface
                ai.state = stInitial;
            } else {
                        .
                        .
                        .
                ai.state = stInitialized;
            }
        }else{
    rr = new RequestResult(RequestResult.FAIL, "ParentPNES is null", pRP.ID);
    }
    break;

    case stInitialized:
        if(pRP.requestType == RequestPars.RoleSessionAction){
            appMsg = pRP.applicationMessage;
            if(appMsg.messageType.compareToIgnoreCase("applicationMessage") == 0){
                        .
                        .
                        .
                ai.state = stNextState;
            } else if(appMsg.messageType.compareToIgnoreCase("nextApplicationMessage") == 0){

                ai.state = stNextState;
            }

    }
        break;

    case stNextState:
        if(pRP.requestType == RequestPars.RoleSessionAction){
            appMsg = pRP.applicationMessage;
            if(appMsg.messageType.compareToIgnoreCase("applicationMessage") == 0){
                        .
                        .
                        .
                ai.state = stAnotherState;
            } else if (appMsg.messageType.compareToIgnoreCase("nextApplicationMessage") == 0) {
                        .
                        .
                        .
                ai.state = stAnotherState;
            }
        }
        break;
```

Figure 5-6 : Mapping from state diagram to code

5.4.1.2 Mapping of state diagram to XML manuscript

An approach to make XML descriptions of manuscripts is introduced in [9]. The XML descriptions are as for the Java implementation based on finite state machines. Figure 5-3 describes how the UML state diagram can be mapped to the state description part of the XML manuscript description.



```
<init_state> stInit </init_state>

<!–Description of state stInit (initial state)–>
<state name="stInitial">

        <input msg="INITIAL_TRANSITION">

                <!–First action:–>
                <action>
                        <!-Specification of action–>
                </action>
   .
            .
                <output>
                        <!-Specification of message sending–>
                </output>
                <next_state> stInitialized </next_state>
        </input>
</state>

<!–Description of state stInitalized –>
<state name="stInitialized">
        <input msg="applicationMessage">

                <action>
                        <!-Specification of action–>
                </action>
                    .
              .
                <output>
                   <!-Specification of message sending–>

                <msg type="NewOutputMsg">
                        <param>
                                <name> name </name>
                                <value> value </value>
                        </param>
                        <dest> destination </dest>
                </msg>
        </output>

        <next_state> stNextState </next_state>
</input>

<!–Description of state stNextState –>
<state name="stNextState">
```

Figure 5-7 : Mapping of state diagram to XML manuscript

5.4.2 The Role Figure Support Classes

As mentioned in section 5.2.2, additional classes will probably be needed to realize the functionality of the application. This may for example be classes used to realize graphical interfaces etc. Methods in these classes will be called from the implemented state machine and it is natural to fill in the code for these classes when mapping the state diagram for the Role Figure to state machine code.

5.5 Building and deploying the application

When the implementation is completed the remaining steps of the development process is to deploy the software components and test the application. In the description of the Tele School application components diagrams and deployment diagrams where used to describe how the software components where related and how the actors where distributed. In Rational Rose Real-Time the component diagram is required when building the application to define the external components needed. A component diagram is therefore required and needs to be a part of the UML template.

As mentioned in section 4.3.6 the deployment diagram in the UML tool is intended to specify the hardware used for deployment. For TAPAS this is not relevant since the deployment is dynamic and not hardware dependent. The description of how the actors are distributed on different kind of nodes is however helpful to describe the application and is also recommended to be used.

The testing of the models developed in this thesis is described in chapter 7. As described for the use case driven approach the test cases are selected to test the use cases of the application.

5.6 Workflow

The traditional way of software development has been to complete the each phase of the development process before starting on the next phase, with the termination of the development phases as milestones in the process. This way of working is referred to as the waterfall model. Given the sophisticated software systems of today an iterative approach is becoming increasingly popular, and is an important element of 'modern' methodologies like the Unified Process described in section 3.4.2, the Rational Unified Process [16] and Extreme programming (XP) [15]. This approach has some strong benefits compared to strongly sequential development. First off all it allows for an increasing understanding of the problem through successive refinement in increments, and to grow an effective solution over multiple iterations. The iterative process will help clarify the risks in the development at an early stage. This is essential when developing software for a new hardware platform, but is also very convenient on well known platforms. The fact that something is working and that progress can be demonstrated at an early stage is satisfying for both the developer and the customer.

During the development of the example application which is described in the next chapter an iterative approach was taken to the steps described in this template. This has proved to be a good approach even for a small project like the one in this example. When the designer is new to a platform like the TAPAS support platform, which is probably the case, it is important to get something to work as soon as possible. This will reveal any weaknesses in the design and verify that the work is taking the correct path, through testing at an early stage.

The recommended work flow is therefore to complete the development for one (or more logically connected) use case(s) at the time. When the system is up and running at the target nodes with the first use case implemented, the process is repeated in iterations until the application development is complete.

5.7 Summary

This chapter has introduced a simple development process for the development of TAPAS applications. A summary of the steps in the development process and the different elements of the provided template is given below:

- **Analysis**: In this phase functional and non-functional requirements are specified. The functional requirements of the application are specified as use cases. The TAPAS support platform is chosen. The result of this phase is a requirements model for the application, which specifies what the application shall be capable of doing. In TAPAS terms; the play is specified.

- **Design**: The steps for the design phase are:

    1. The Role Figures which are going to realize the play are specified.
    2. The specified use cases are described in detail by sequence diagrams. The objects of the sequence diagrams are the Role Figures of the play and other actors which have supporting roles, such as the Director and the MobilityManager.
    3. The behaviour of the Role Figures is specified by state diagrams.

    The result of this phase is a complete design model. Specifying how the specified use cases are realized. In TAPAS terms; how the play is going to be realized by a set of Role Figures.

- **Implementation**: In this phase the design model developed in the previous step is transferred to executable code. Mapping from state diagrams, describing the Role Figure behaviour, to Java code or XML is done.

- **Test**: The specified use cases are tested on target nodes.

6 An application based on Role Figure Mobility

An application is developed to test the development process and the template introduced in the previous chapter. The application is based on the concept of Role Figure Mobility. This chapter describes how this application is developed using the template introduced in chapter 5. The support platform used is MicroTAPAS, since this is the platform currently supporting the Role Figure Mobility.

6.1 The PatientRecords Application

The application developed in this example is intended for the health care service. Most hospitals today have an electronic patient journal, which means that the patient data is stored on a central unit and can be accessed from different work stations at the hospital. The next step for an application like this, apart from developing the journal's internal structure and user interface, is to make it available from mobile terminals. This will enable the hospital personnel to access the patient journal from i.e. a PDA in any situation and at any location within the coverage of the wireless network. A nurse will for example be able to access the journal at the patient's rooms. However, when in the office or at another location where an PC is available, working on the PC will be preferred due to the larger display, ease of entering data and the extended processing power. It will then be convenient to move the application from the PDA to the PC. When leaving the office we may need to move the application back to the PDA. Moving the application between the terminals means that it is not necessary to log off and close down the program and then start over again at a new terminal.

The concept of Role Figure Mobility implemented in MicroTAPAS will provide the functionality to move a Role Figure from one terminal to another and recreate the Role Figure at the new location. An electronic patient journal is of course an application which requires strict security considerations. Also, data storage shall be safe and very reliable. This example will not consider these issues and will be a strongly simplified version compared to what will be used at a hospital. The intention of this example is to demonstrate the development of an application using the template developed and the Role Figure Mobility concept. The concept of Role Figure Mobility is explained in section 2.2.2.

6.1.1 Requirements specification

6.1.1.1 Functional requirements

As described in the template in section 5.1.1, the functional requirements are specified using a use case diagram. The use cases for the PatientRecords can be seen in Figure 6-1.

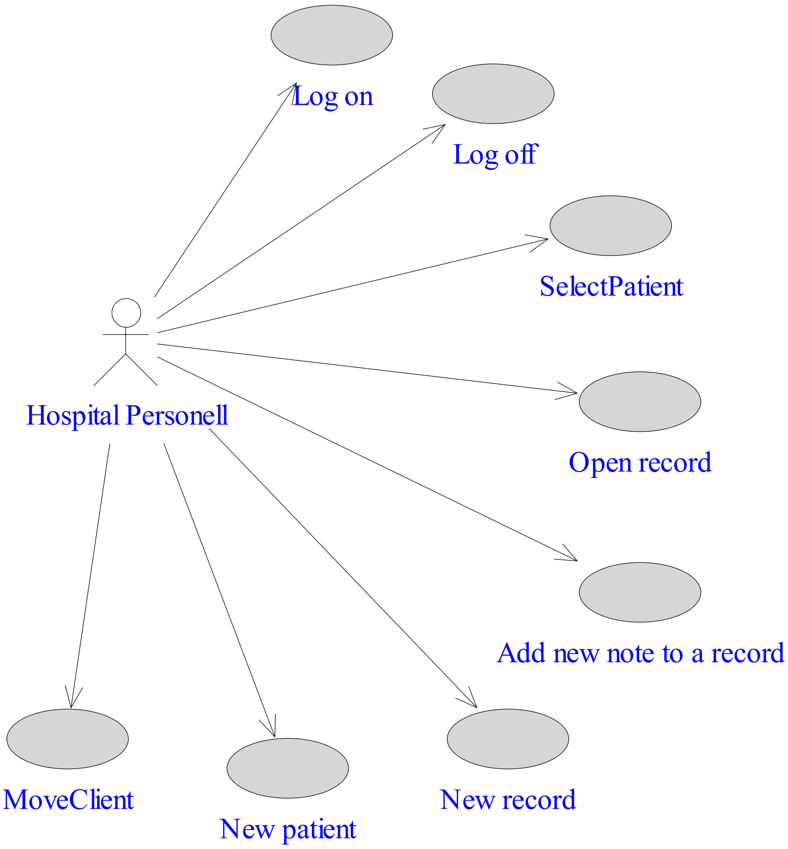


Figure 6-1 : Use cases for the PatientRecords application

6.1.1.2 Non functional requirements

The non functional requirements for the application are summarized below:

- Usability – the client shall be easy and intuitive to use
- Performance – the access time towards the server shall be short and not delay the users work.
- Design – the design shall be in the way that the new functionality easily can be added to the application and that some parts, like the record data base can be replaced with new technology.

6.1.2 Specifying the Role Figures and support classes

This application will be of the type client server. It is natural to let one Role Figure represent the client, and one Role Figure represents the server. The client role figure will reside on the terminal, which will be a PC or a PDA. The server Role Figure will probably run on the same machine as the record database. To make the design flexible to future development, it is smart to have an own actor for the user interface. The database shall also have a well defined interface which will make it easy to update the data base solution or replace the data base with an entire new one. Since the application requires the possibility of moving of one of the Role Figures, the MicroTAPAS platform and the mobility features will be used. Figure 6-2 shows a high level class diagram for the application.

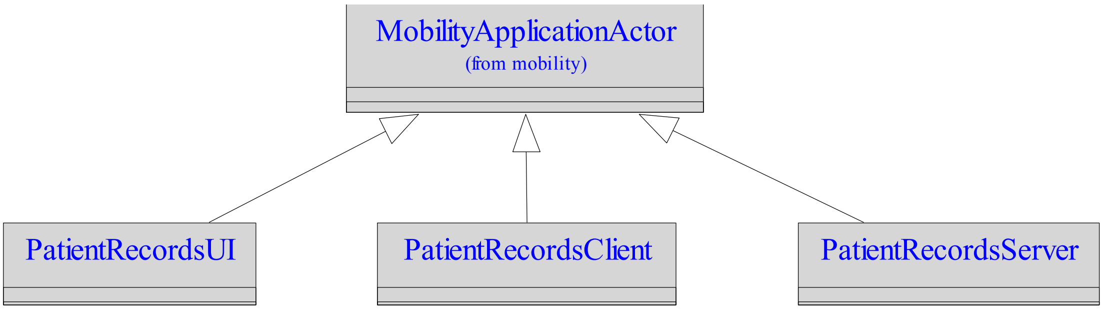


Figure 6-2: Class diagram for the PatientRecords application

6.1.3 Description of the functionality

In this section the use cases are broken down into more detailed sequence diagrams. The diagrams in this section shows how the use cases presented in section 6.1.1.1 is realized by using the classes specified in section 6.1.2.

6.1.3.1 Logon and logoff

A sequence diagram showing the Logon sequence is shown in Figure 6-3.

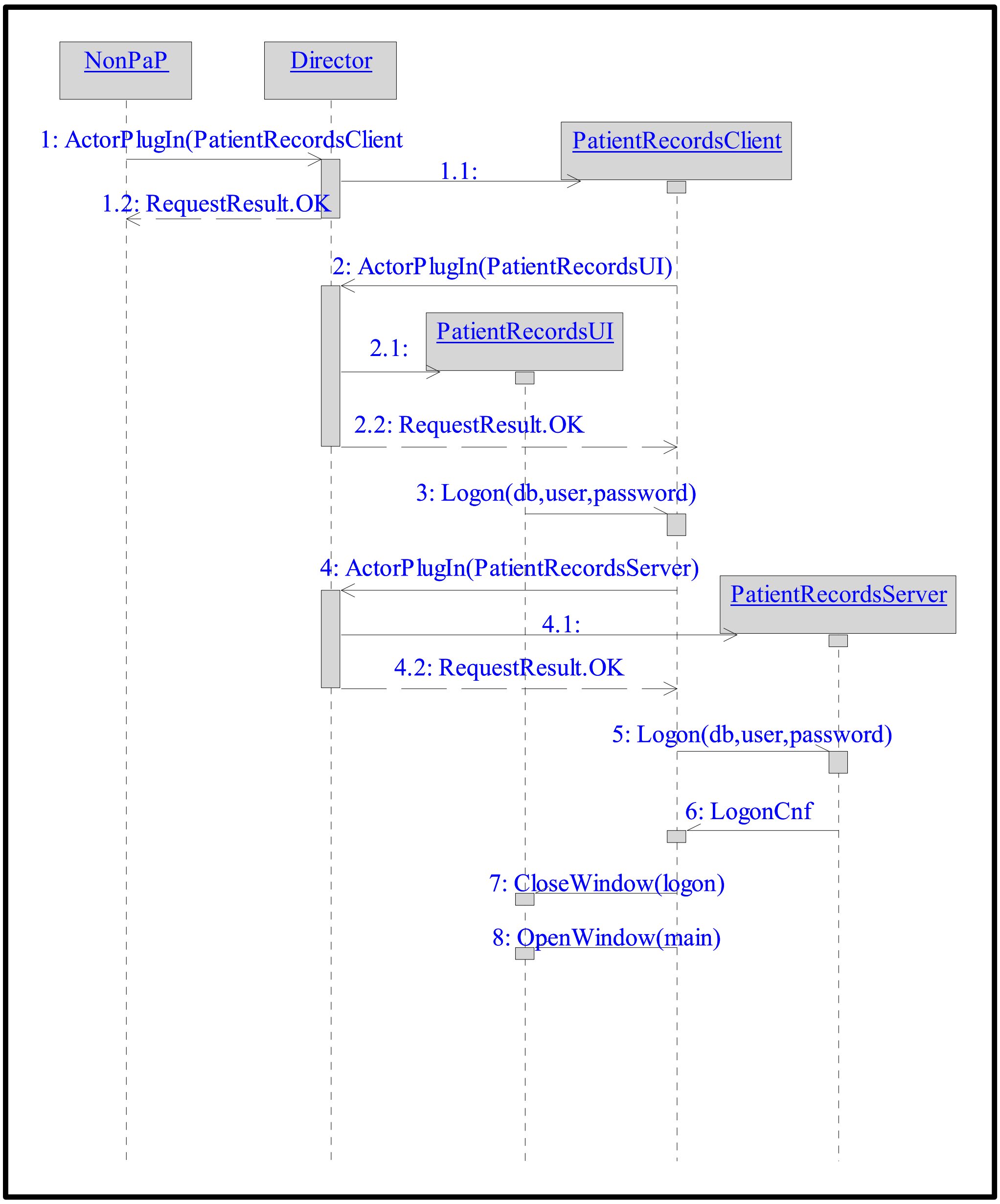


Figure 6-3 : Log on sequence for PatientRecords

6.1.3.2 Select patient

It shall be possible to select a patient from a list of patient. When the patient is selected, the records available for this patient will be displayed in a list of records. The sequence executed for selection of patient is shown in Figure 6-4.

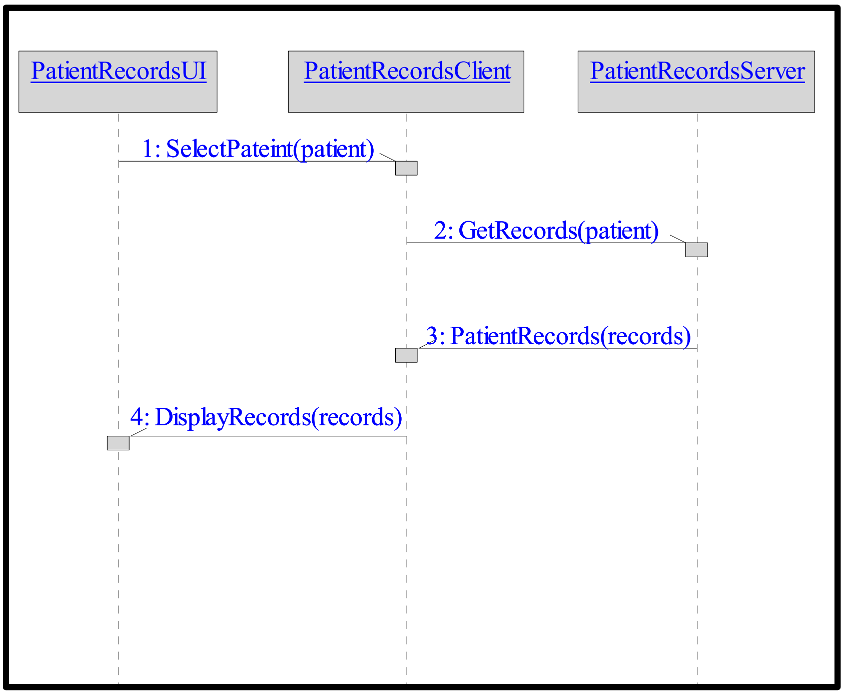


Figure 6-4 : Sequence diagram for select patient

6.1.3.3 Open record

When a patient is selected, one of the records available can be selected from the list of records and a new window will be opened for this record. The record is only open for reading and the existing content can not be changed. To add information to the record, 'add new note' must be selected from the pull down menu 'File'. This sequence is described in section 6.1.3.5. The sequence executed for selection of record is shown in Figure 6-5.

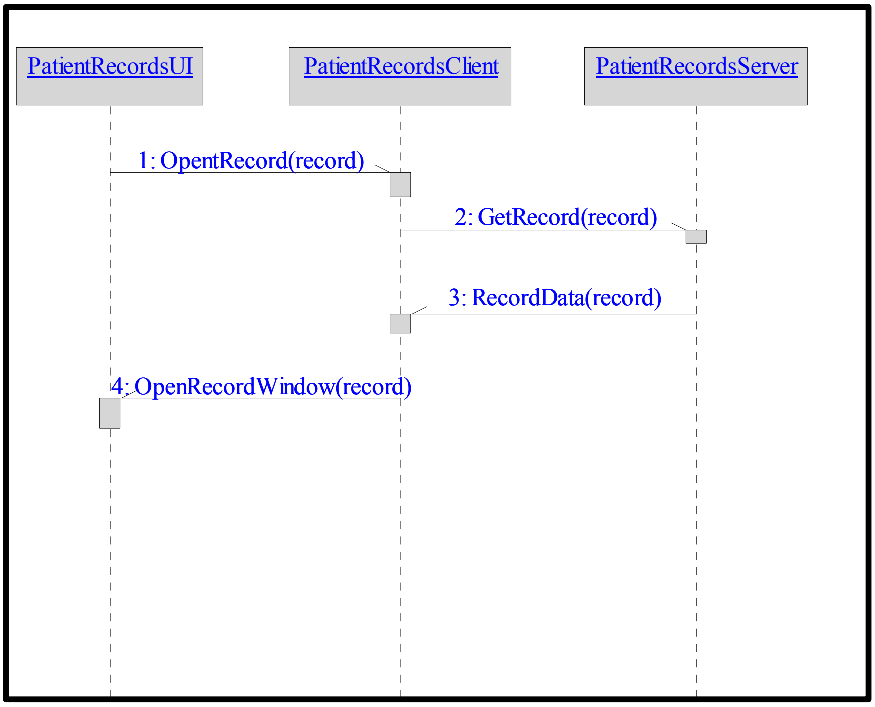


Figure 6-5 : Sequence diagram for open record

6.1.3.4 Create a new record

A new record can be created for the selected patient. Figure 6-6 shows the sequence for creation of a new record.

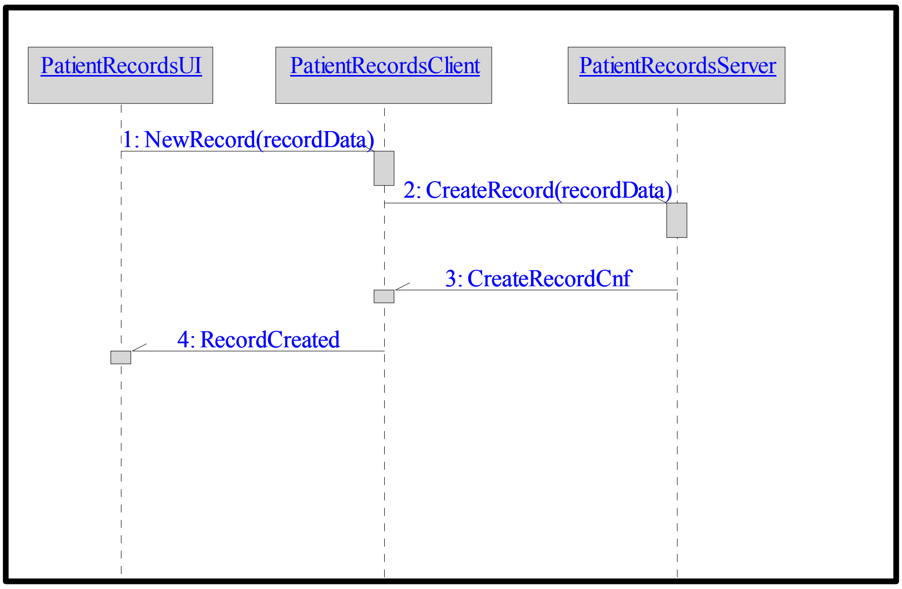


Figure 6-6 : Sequence diagram for creation of new record

6.1.3.5 Add a new note to a record

When a record is open and a new note is to be added to this record, a new window shall be opened where information can be edited. The window shall have an own filed for the signature of the person adding the note to the record. The sequence for adding a note is described in Figure 6-7.

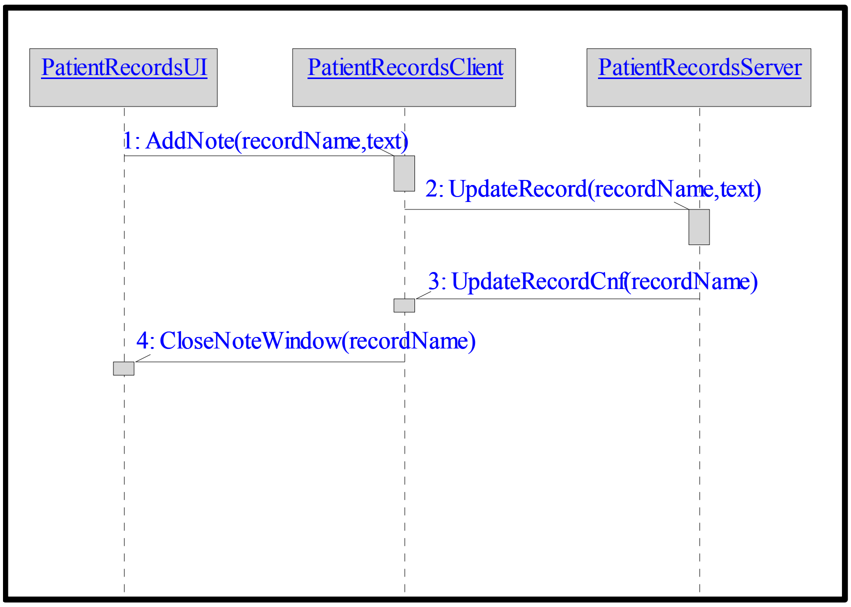

PatientRecordsUI
PatientRecordsClient
PatientRecordsServer
1: AddNote(recordName,text)
2: UpdateRecord(recordName,text)
3: UpdateRecordCnf(recordName)
4: CloseNoteWindow(recordName)

Figure 6-7 : Sequence diagram for add new note

6.1.3.6 Register a new patient

It shall be possible to add a new patient to the database. Figure 6-8 shows a sequence diagram for adding a new patient.

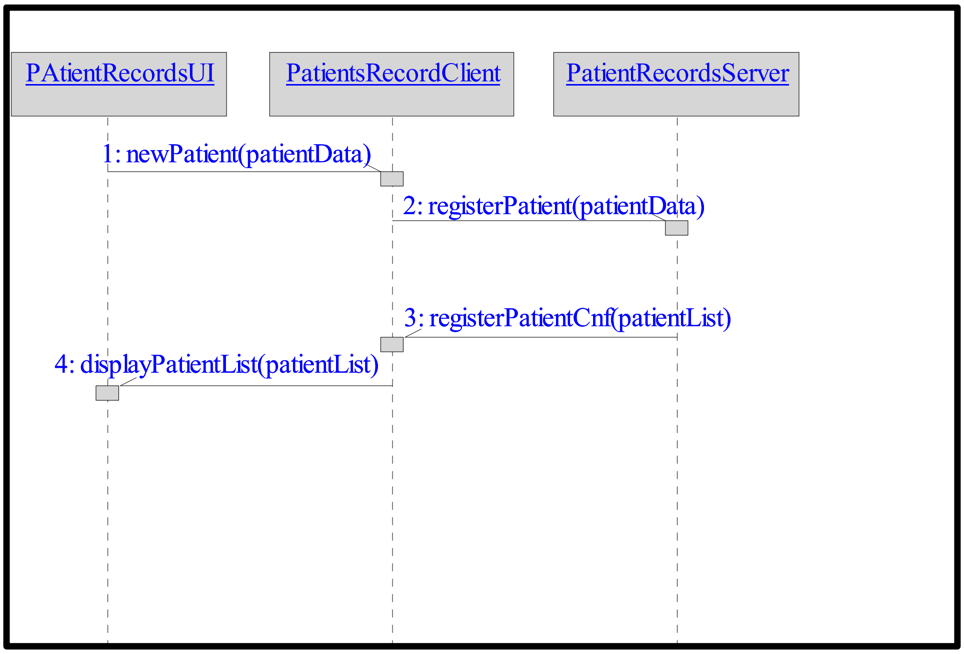

PAtientRecordsUI
PatientsRecordClient
PatientRecordsServer
1: newPatient(patientData)
2: registerPatient(patientData)
3: registerPatientCnf(patientList)
4: displayPatientList(patientList)

Figure 6-8 : Sequence diagram for adding a new patient

6.1.3.7 Moving the client to a new terminal

As long as the user is logged on to the application it shall be possible to move the terminal client to a new terminal by selecting the new terminal from a 'pull down' menu. The sequence executed for the move of the terminal client is shown in Figure 6-9 below. As can be seen in the diagram the ActorMove function from the MicroTAPAS mobility feature is used to perform the moving of the client.

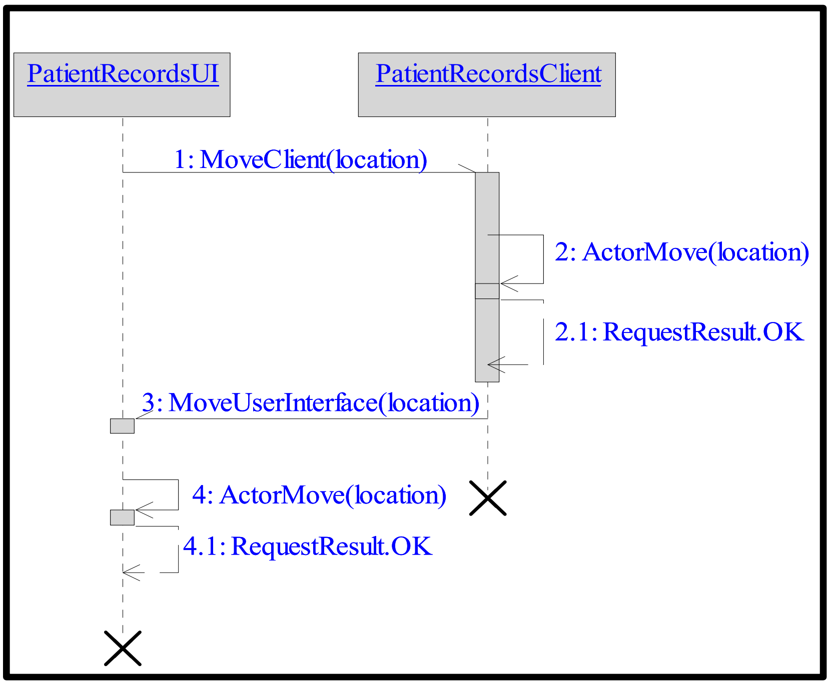


Figure 6-9 : Sequence diagram for move terminal

6.1.4 Behaviour specification

This section describes the behaviour of the Role Figures specified in the previous section by state diagrams.

6.1.4.1 PatientRecordsClient behaviour

The behaviour of the PatientRecordsClient is described in Figure 6-10. The events causing transitions between states are mainly of the type ApplicationMessage, but the createInterface method is inherited from the MobilityApplicationActor class and is used to create the interface for the actor when it has been moved. The actor move procedure is described in Figure 4-17. In this case the Role Figure can only be moved to a new terminal when the state is stLoggedOn, stWaitLogon or stWaitSelectPatient which is described in the state diagram. When receiving an actorPlugIn request, it will be checked if this actor is moved. If it is moved the state will remain stInitial until the createInterface method is called by the old actor. When createInterface is called, the Role Figure will be recreated at the new location.

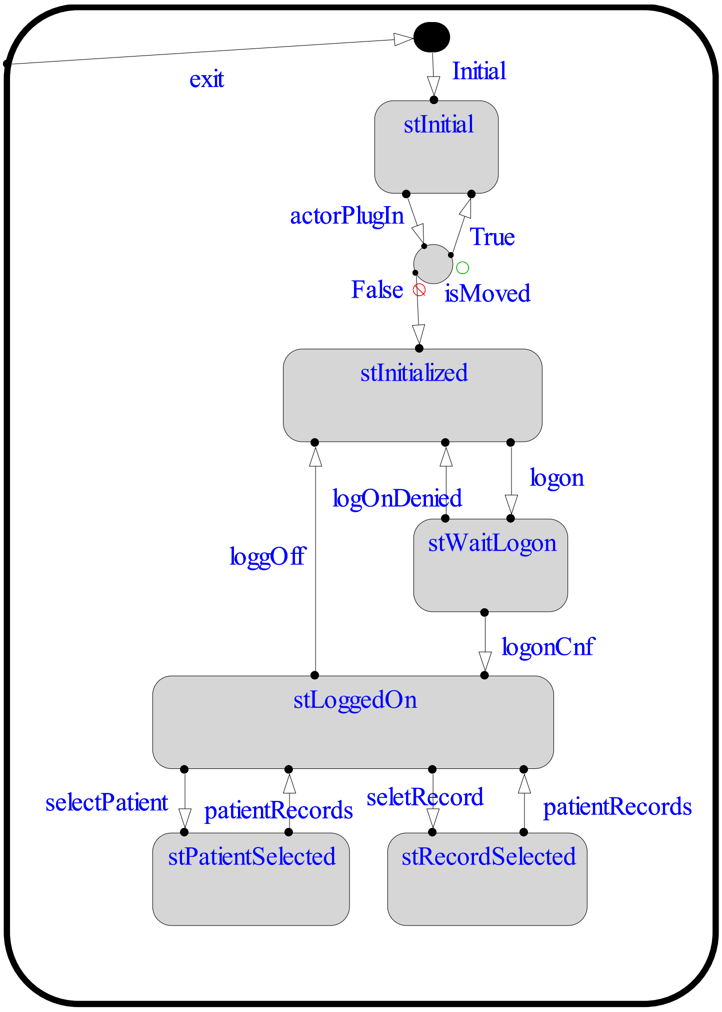


Figure 6-10 : State diagram for the PatientRecordsClient

6.1.4.2 PatientRecordsServer behaviour

The behaviour of the PatientRecordServer is described in Figure 6-11. Once the user is logged on the state stServerActive is entered.

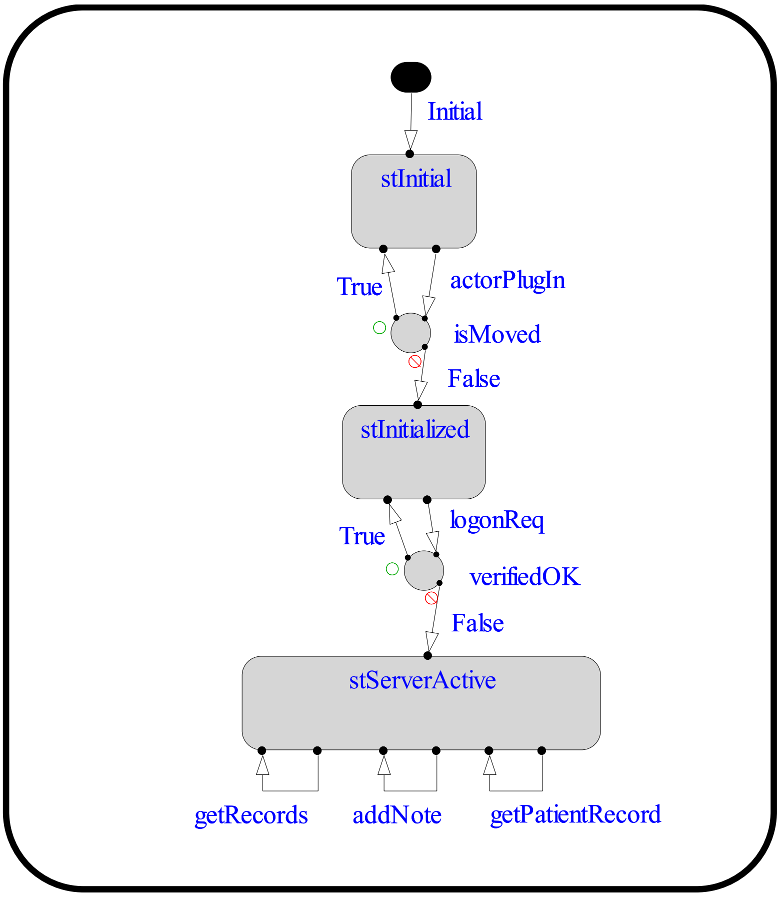


Figure 6-11 : State diagram for the PatientRecordsServer class

6.1.5 PatientRecordsUI behaviour

The behaviour of PatientRecordsUI is described in Figure 6-12. As for the PatientRecordsClient this class must be able to be moved to a new location. The behaviour of the PatientRecordsClient is fairly simple and it basically does what is required in the messages from the PatientRecordsClient regardless of the status on the graphical user interface.

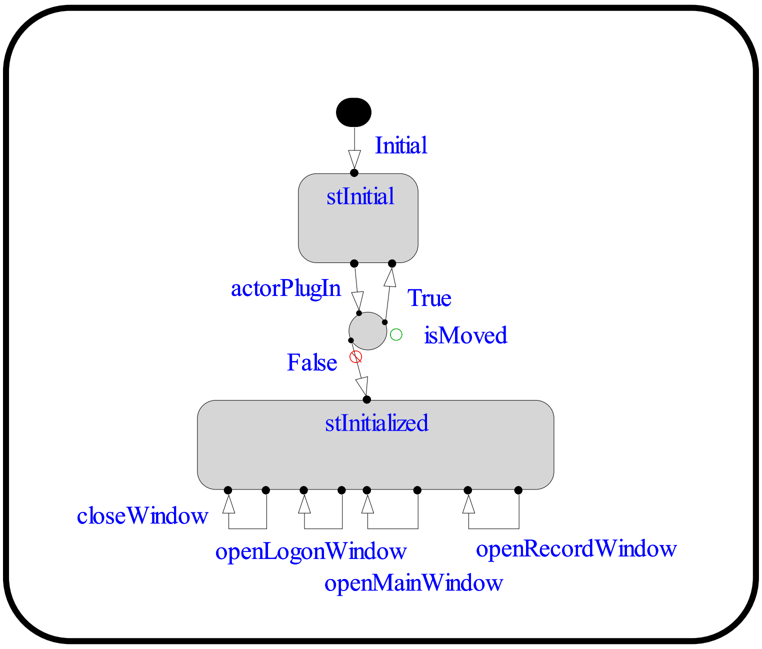


Figure 6-12 : State diagram for PatientRecordsUI

6.1.6 Implementation of the application

The behaviour of the Role Figures is mapped to Java code following the procedure described in section 5.4.1.1. The following sections describe the classes introduced in section 6.1.2 and the supporting classes in more details. The complete generated code can be found in Appendix A. The testing of the application is described in chapter 7.

6.1.6.1 The PatientRecordsClient class

The PatientRecordsClient class is described in Figure 6-13 below. This class represents the client behaviour. It controls the user interface and is responsible for the communication towards the server. As can be seen from the figure the data needed for recreation of the Role Figure is kept in the interface class, PatientRecordsClientInterface.

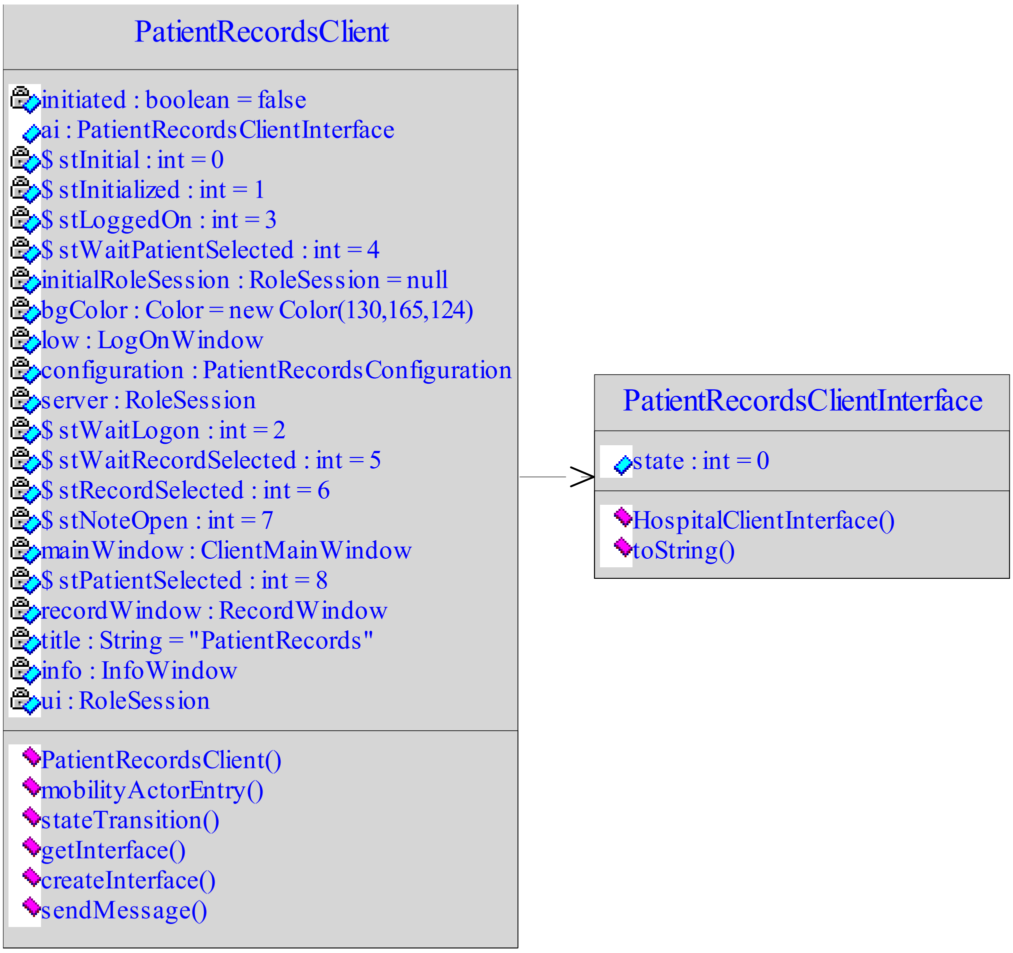

PatientRecordsClient
initiated : boolean = false
ai : PatientRecordsClientInterface
$ stInitial : int = 0
$ stInitialized : int = 1
$ stLoggedOn : int = 3
$ stWaitPatientSelected : int = 4
initialRoleSession : RoleSession = null
bgColor : Color = new Color(130,165,124)
low : LogOnWindow
configuration : PatientRecordsConfiguration
server : RoleSession
$ stWaitLogon : int = 2
$ stWaitRecordSelected : int = 5
$ stRecordSelected : int = 6
$ stNoteOpen : int = 7
mainWindow : ClientMainWindow
$ stPatientSelected : int = 8
recordWindow : RecordWindow
title : String = "PatientRecords"
info : InfoWindow
ui : RoleSession
PatientRecordsClient()
mobilityActorEntry()
stateTransition()
getInterface()
createInterface()
sendMessage()
PatientRecordsClientInterface
state : int = 0
HospitalClientInterface()
toString()

Figure 6-13 : Class diagram for PatientRecordsClient

6.1.6.2 The PatientRecordsUI class

The PatientRecordsUI class is described in Figure 6-14. This class is responsible for the graphical user interface. It controls the different windows used by the application. The interface class PatientRecordsUIInreface keeps the data needed for reconstruction of the Role Figure at a new location. This includes status on open windows and their content.

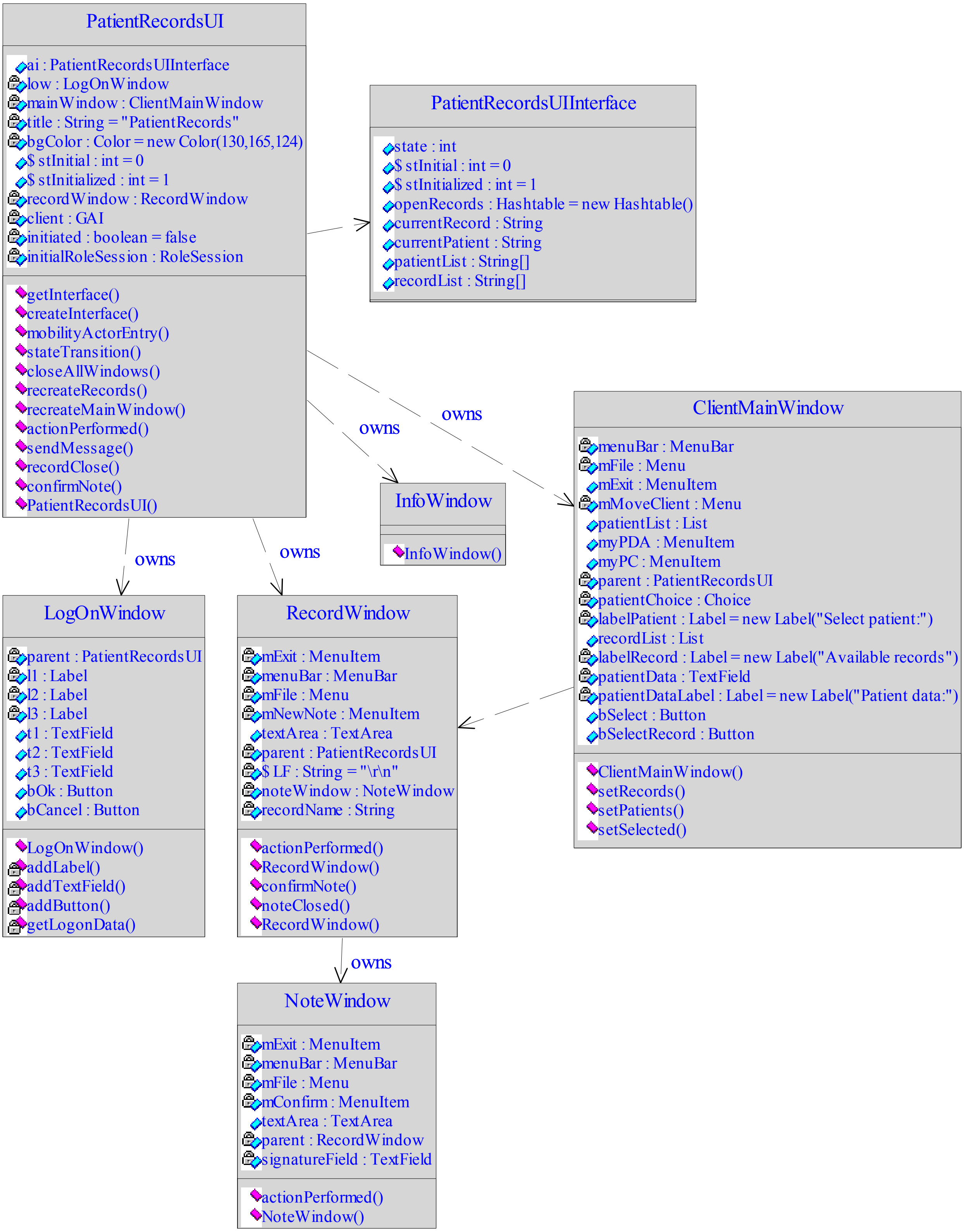


Figure 6-14 : The PatientRecordsUI classes

6.1.6.3 The PatientRecordsServer class

The PatientRecordServer class is described in Figure 6-15. This class is responsible for server functionality of the application. It accesses the data base and communicates with the client.

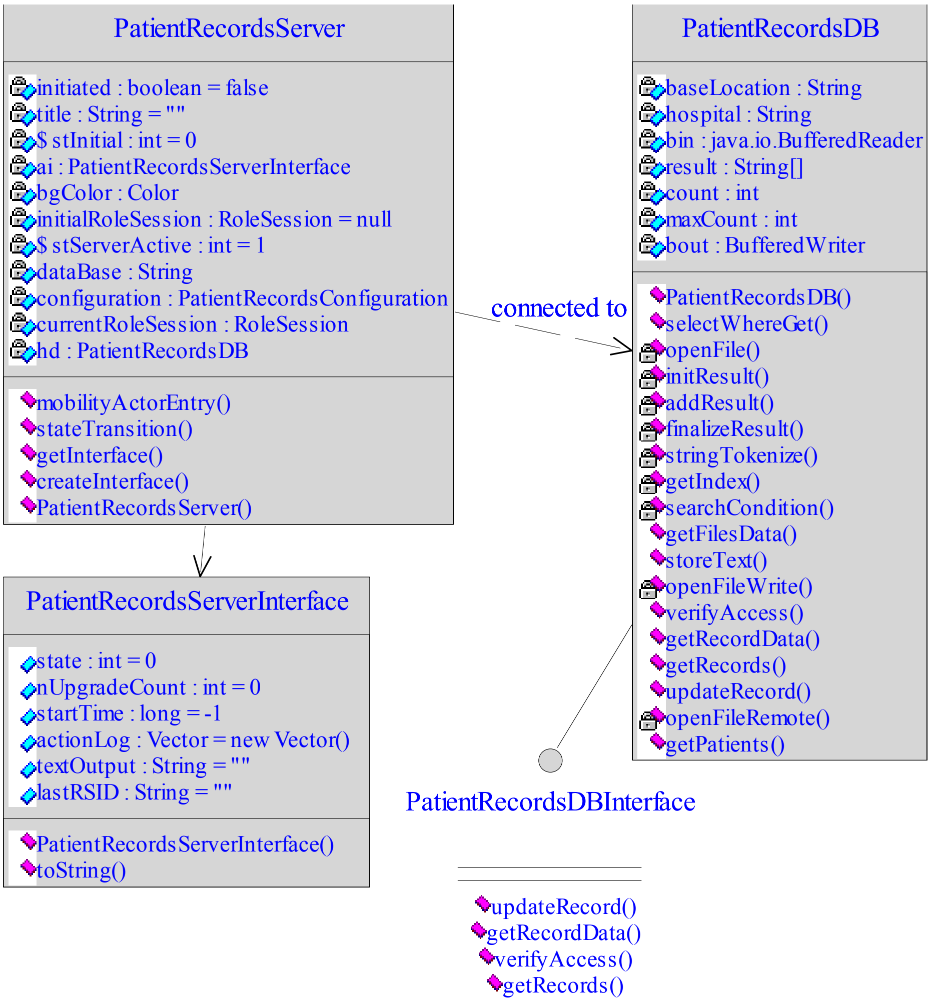


Figure 6-15 : The PatientRecordsServer class

6.1.7 Deployment of the software components

As described in section 5.5 UML provides diagrams to describe how software components are organized and how they are deployed on the hardware. According to the template provided a component diagram in Figure 6-16 describes the software components of the PatientRecords application.

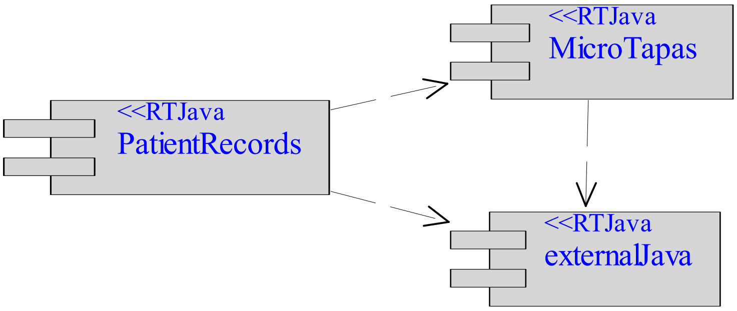


Figure 6-16 : Component diagram for the PatientRecords application

In Figure 6-17 a deployment diagram for the application is shown. The diagram describes what actors to be downloaded on the different types of TAPAS nodes. The PatientDB class is also included in this diagram, despite not being an actor, because it is a significant element of the application.

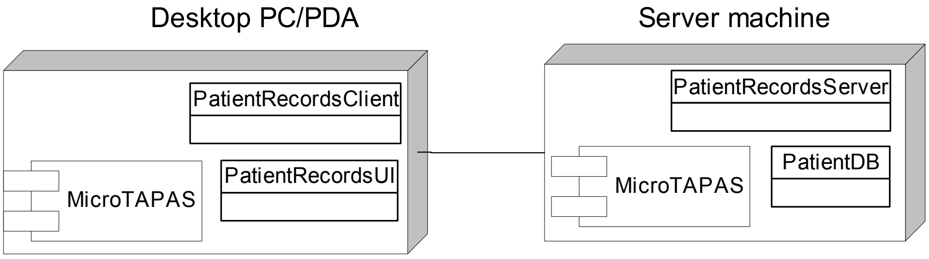
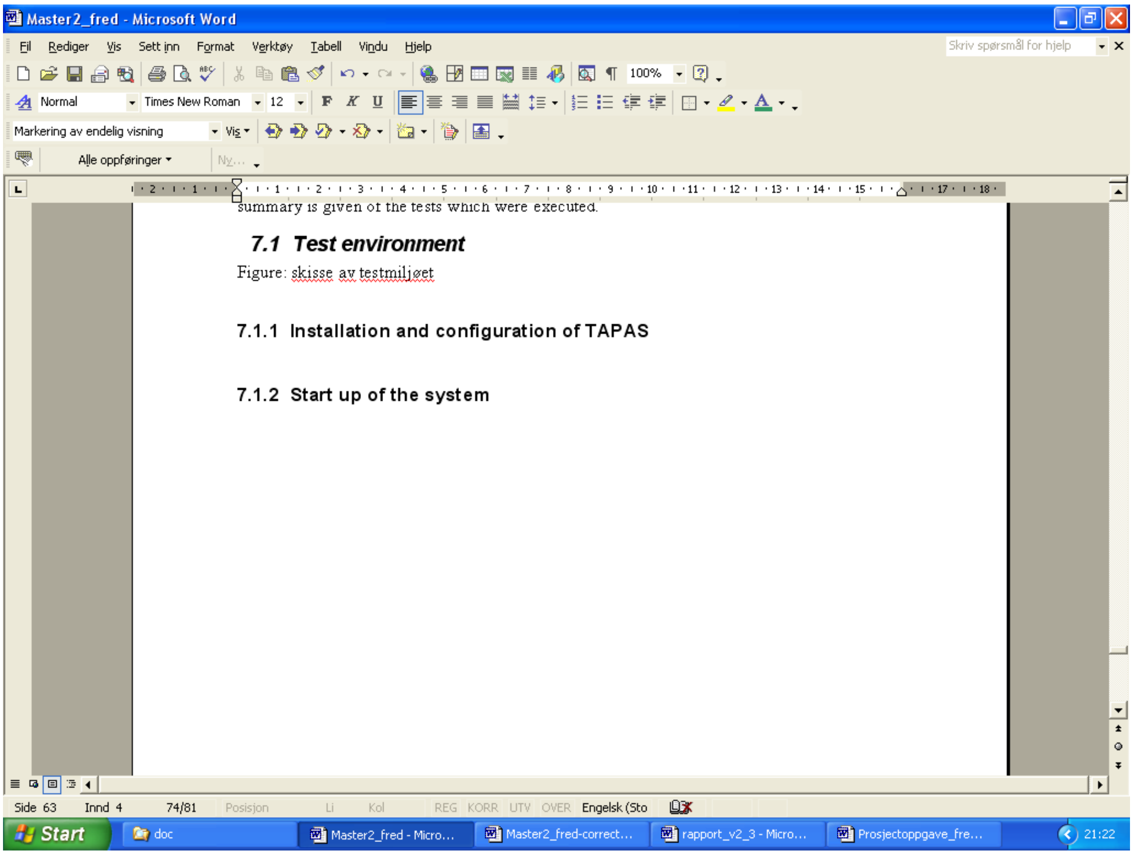


Figure 6-17 : Deployment of the PatientRecords application

6.1.8 Screen shots

<still missing : Here some screen shots will be shown>



Master2_fred - Microsoft Word
Fil Rediger Vis Sett inn Format Verktøy Tabell Vindu Hjelp
Skriv spørsmål for hjelp
Normal
Times New Roman
12
100%
Markering av endelig visning
Vis
Alle oppføringer
Side 63 Innd 4 74/81 Posisjon Li Kol REG KORR UTV OVER Engelsk (Sto
Start
doc
Master2_fred - Micro...
Master2_fred-correct...
rapport_v2_3 - Micro...
Prosjectoppgave_fre...
21:22

summary is given of the tests which were executed.

7.1 Test environment

Figure: skisse av testmiljøet

7.1.1 Installation and configuration of TAPAS

7.1.2 Start up of the system

Figure 6-18 : Screen shot from the PatientRecord application

6.1.9 Summary

In this chapter an example application is developed using the process introduce in chapter 5. The model is based on the role figure mobility feature provided with the MicroTAPAS platform. The application is a version of an electronic patient journal used in health care service, where the scenario is that the electronic patient journal client is moved seamlessly between terminals, depending on what situation the user of the application currently is in. The application is running on theMicroTAPAS platform, which also allows the role figures to be moved to PDAs.

7 Test and verification

To verify that the models and the generated code work some tests are carried out. This chapter describes the test environment used and the test cases executed.

7.1 Test environment

The network setup used for testing of the code generated from the models is shown in Figure 7-1. The test environment consists of a desktop computer and a PDA which are connected to a LAN. The PDA is connected through a wireless LAN access point.

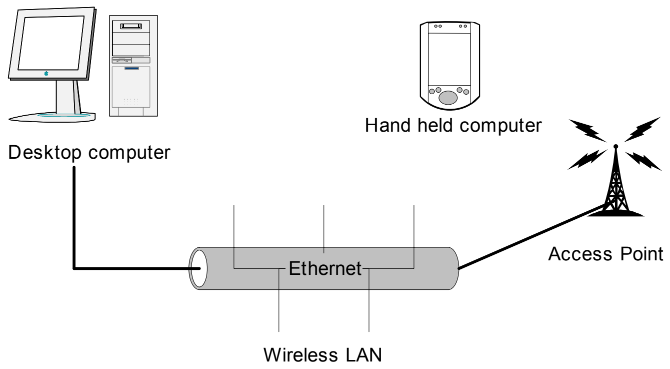


Figure 7-1 : Test environment

The desktop computer is a PC running a Java Virtual Machine (JVM) in the Windows XP operative system. The PDA runs J9, which is a virtual machine from IBM, on the Windows CE operative system.

7.1.1 Installation and configuration

As described in chapter 2 the TAPAS plays and support platform are stored on a web server and downloaded by the TAPAS nodes which are going to participate in a play when needed. The deployment of the TAPAS code is also described in section 5.5. The TAPAS node requires some boot software which will start up the TAPAS support platform on the node.

Installation and configuration of the TAPAS system is described on the TAPAS home page [17]. MicroTAPAS is also documented in [7].

7.2 Tests and results

To verify that the models of the support platforms work probably the code generated from them is tested using some of the existing and well tested applications. But first a visual inspection of the generated code is done, comparing the code with the code that the models are reverse engineered from. The following sections summarize the test cases executed for the models in the test environment described in previous chapter.

7.2.1 The Tele School application on the basic TAPAS support platform

Table 7-1 below describes the test cases executed for the Tele School application running on top of the basic support platform generated from the UML model. The test cases are selected to test the use cases for the Tele School application in section 4.3.1.

Number	Test case	Desired result	Result
T-1.1	Log on	Successful logon and opening of the work to do window	OK
T-1.2	Log off	Successful logoff and closing of all open windows	OK
T-1.3	Select ‘Courses and Lectures’	A new window is opened with a list of services.	OK
T-1.4	Select Real-Time Lecture service	A new window is opened for the RLT service	OK

Table 7-1 : Test Cases for the TAPAS basic support platform and the Tele School application

7.2.2 The PatientRecords application on MicroTAPAS platform

The test cases executed for the PatientRecords application is listed in Table 7-2. These tests also serve as regression test for the generated MicroTAPAS support platform. These test cases are selected to test the use cases presented in section 6.1.1.1.

Number	Test case	Desired result	Result
T-1.1	Log on	Successful logon results in opening of the main window.	OK
T-1.2	Log off	All windows are closed and the actors are plugged out.	OK
T-1.3	Select patient	The records available for this patient are displayed	OK
T-1.4	Select record	A new window is opened for the selected record	OK
T-1.5	Add new note	A new window is opened for this record where a new note can be added	OK
T-1.6	Confirm a note	The note window closes and the text in the record window is updated. The record is updated on the server.	OK
T-1.7	Add a new record	A new record is added to the patient’s list of records	OK
T-1.8	Add a new patient	A new patient is registered and displayed in the list of patients	OK
T-1.9	Move the client from a PC to a PDA	All the active windows are closed. The actors are moved and the active windows are reopened at the new location. The text in the window shall be exactly the same as at the old location.	OK
T-1.10	Move the client from a PDA to a PC.	The same result as the previous test case.	OK

Table 7-2 : Test Cases for the PatientRecords application

7.3 Test methodology

Testing of software systems is a wide area and by some considered an own science. The tests executed in this section are simple and not very extensive, yet not to be considered ineffective for applications of this size and complexity. If the applications grow in complexity and the number of different actors increase, a new approach to quality assurance probably have to be made. Testing of large systems on target nodes tends to be expensive, especially if the code is of poor quality at this stage. A new test strategy needs to be introduced to for the future applications.

One common approach is to test the application on a unit level, which means that complete tests are done for each class. Execution of tests at an early stage will reveal errors that would be more costly to discover later on in the development process. To execute the tests, own code can be developed or one of the test frame works available can be applied. One example of a test framework for Java is JUnit which is open source [18].

7.4 Summary

This chapter has described the test environment and test cases executed for code generated from the UML models developed. The use cases for the Tele School application are tested on target nodes to verify that the code generated from the UML model is correct. These tests also verify that the code generated from the basic support platform UML model is correct. The same approach is taken for the PatientRecords application. This application is tested on the code generated from the MicroTAPAS UML model.

8 Discussion

In this chapter the work carried out and the solutions that are chosen in this thesis will be discussed related to the goals. Further, some proposals for further work are given.

8.1 Experiences with the modelling tool

One of the objectives with this thesis work was to increase the speed of developing TAPAS applications through creation of a modelling environment. By introducing an UML modelling tool, clearly some major improvements in the modelling process are gained. The modelling tool used in this thesis work, Rational Rose Real-Time, has most of the features expected from a sophisticated and well known tool. One of the major benefits is that the design, implementation and deployment tasks of the applications can all be done in one tool. The modelling tool links all the parts of the development together, making it easier to keep model and implemented code updated as development proceeds. The code and the relationships in the model will always be synchronized, and the designer is able to brows the model and have a look at the detailed code by clicking the elements of interest. Further, building of the system components is integrated in the tool and code is generated.

However, some code always needs to be written and when this phase is reached, the user is reminded that the tool is made with focus on modelling. The editor has no extra features and that may be a drawback compared to advanced editors and integrated development environments (IDE).

8.2 Evaluation of the template and development process

One of the major benefits of the template provided is that a complete design model can be developed, which is independent of the implementation. The idea is that much effort shall be made to develop a complete and detailed design model. UML does as discussed earlier not provide the possibilities for formal description of the functionality, but it is possible to develop detailed and standalone design models. This is also in principal the case for the models of the TAPAS support platforms. It should be stated that for the design model to be independent it has to be detailed and consistent, not leaving any issues to be misinterpreted.

By developing a good design model, the implementation part of the development process is straight forward using the mapping procedure to Java or XML, which are used in this thesis. The model is flexible for the choice of implementation and new mapping schemas to other languages will be easy to introduce. However, the solution which is most promising for the future is mapping to XML, while this will provide completely platform independent manuscripts.

The benefits of a good process and design rules can first be seen when developing systems which are quite complex. The example application is due to time constrains not a very advanced application, thus not illustrating the benefits of a modeling environment to the full extent.

8.3 Enhancements and further work

8.3.1 Application messages

In the existing TAPAS support platforms, the application message data type consists of message type and message which are represented as String and String[] respectively. This format is not flexible from the application developer's point of view. Conversion of data types will often be required and some data types will not be possible to send as application messages. A more flexible way of implementing the application messages will be to make a stereotype for the application message format which can be extended for the various types of messages. This will also be more descriptive in the model of the application, because the message types are implemented as own classes.

8.3.2 Towards UML2.0

As described in section 3.1.3, UML2.0 introduces new concepts for behaviour modelling that will be useful for modelling of complex behaviour. In future development of TAPAS applications these concepts will probably be useful and when more modelling tools will support new versions of UML it is natural to emerge TAPAS modelling into the new standards.

8.3.3 Code generation from state diagrams

This thesis proposes a mapping procedure from the UML state diagrams to Java code. The mapping is simple and easy to perform. However, the developer which is familiar with modelling in SDL may have other expectations to automatic code generation from the state diagrams. This may also become possible in new versions of UML, but it could also be an idea to develop a simple code generator for the behaviour of the TAPAS Role Figures.

8.3.4 XML and behaviour specification

XML has quickly become a universal standard of storing and distributing information in the software industry. Using XML to represent the behaviour of the Role Figures will provide platform independent manuscripts, which means that the actors executing on various platforms may download and use the same manuscript files. This will enhance the ability to handle platform compatibility issues. In this thesis mapping from the Role Figure Model to XML is described.

9 Conclusion

The main objective of this thesis was to find out if UML is suitable for modelling of TAPAS applications. UML models of the existing TAPAS platforms, and a template for further application development should be made. To demonstrate the use of the template an example application should be developed with focus on utilization of the Role Figure Mobility concept.

The models are developed using the UML modelling tool Rational Rose Real-Time. Models are made for the basic support platform and for MicroTAPAS. A model is also made for the Tele School application to investigate how the application uses the TAPAS platform. The new template consists of a set of UML diagrams, template classes and design patterns to use for application development. A process for the modelling and implementation phase is also described. The UML models are completed with code by reverse engineering the code from the existing prototypes. The result is that the support platforms and the new applications developed to run on top of them can be completely developed and deployed using the modelling tool.

An example application is developed by using the template, which demonstrates use of the Role Figure Mobility concept in TAPAS. The PatientRecords application is a simplified system for electronic patient journals, where the user client can be moved seamlessly between terminals.

This report shows that UML is a well suited for modelling of the TAPAS support platforms, basically because the platform prototypes is developed in Java which contain many of the same concepts as UML. However, UML has until now been lacking formal support for behaviour modelling which is essential in modelling of the TAPAS Role Figures. The new version of UML, called UML2.0, has major improvements in this area of modelling. At the moment few modelling tools are supporting UML2.0 and it is hard to find a tool supporting 2.0 and generation of Java code which is an important building block in the TAPAS development. Therefore, a tool based on UML 1.4 is used for modelling in this thesis and Java code is generated. The conclusion is that this is sufficient for modelling of behaviour which is of moderate complexity. In the future, when applications become more and more advanced and behaviour more complex, new versions of UML will be appropriate for modelling of the Role Figures.

References

[1] Finn Arve Aagesen, Bjarne E. Helvik, Chutiporn Anutariya, and Mazen Malek Shiaa: *On Adaptable Networking*. The 2003 International Conference on Information and Communication Technologies (ICT 2003), Bangkok-Thailand,april 2003.

[2] Mazen Malek and Finn Arve Aagesen: *Mobility management in a Plug and Play Architecture*, IFIP TC6 Seventh International Conference on Intelligence in Networks (SmartNet2002), Saariselka - Finland, April 2002. Published by Kluwer Academic Publishers.

[3] Finn Arve Aagesen, Bjarne E. Helvik, Ulrik Johansen and Hein Meling: *Plug and Play for telecommunication functionality – architecture and demonstration issues*. The International Conference on Information Technology for the New Millennium (IConIT2001), Thammasat University, Bangkok – Thailand, May 2001.

[4] K.O Chow, Weijia Jia, Vito C.P Chan, Jiannong Chao: *Model based generation of Java code.* International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). Las Vegas 2000.

[5] UML - Object Management Group – Specification found at http://www.omg.org [Accessed Mai 2004]

[6] Eirik Luhr: *Mobility support for wireless devices- within the TAPAS platform*. Master Thesis, Deparment of telematics, NTNU 2004

[7] Eirik Luhr: *TAPAS for Wireless PDA*. Project Report, Department of Telematics, NTNU, 2003

[8] Hein Meling: Complete System Overview, http://tapas.item.ntnu.no/documentation/SystemDoc/Main/Main.pdf [Accessed Mai 2004]

[9] Shanshan Jiang and Finn Arve Aagesen: *XML-based Dynamic Service Behaviour Representation*. NIK'2003. Oslo, Norway, November 2003

[10] SDL – Specification and Description Language, CCITT recommendation Z100

[11] Geir Melby: *Using J2EE Technologies for implementation of ActorFrame based UML2.0 models.* Master Thesis, Agder University College, Grimstad 2003.

[12] Ivar Jacobson: *Object-Oriented Software Engineering – a use case driven approach*. Addison-Wesley Professional 1992. ISBN: 0201544350

[13] J Krogstie: Evaluating UML using a Generic Quality Framework. 'UML and the Unified Process', IDEA group, published 2003.

[14] Kendall Scott: The Unified Process explained. Addison-Wesley Professional 2001. ISBN: 0201742047

[15] Extreme programming XP: http://www.extremeprogramming.org/
[Accessed June 2004]

[16] Rational Unified Process – best practices for software development teams. White Paper. Rational Software Cooperation 1998.

[17] TAPAS, website. Available online: http://tapas.item.ntnu.no
[Accessed Mai 2004]

[18] JUinit, website. Available online: http://www.junit.org
[Accessed Mai 2004]

Appendix A: Source code for the PatientRecords role figures

This appendix contains the generated source code for the Role Figures of PatientRecords application which is the PatientRecordsClient, PatientRecordsUI and the PatientRecordsServer.

```
package PatientRecords.v1_1;
import PatientRecords.v1_1.PatientRecordsClientInterface;
import PatientRecords.v1_1.NoteWindow;
// {{{RME classifier 'Logical View::PatientRecords::v1_1::PatientRecordsClient' tool 'RTJava' property 'ClassFileHeader'
import MicroTAPAS.*;
import MicroTAPAS.mobility.MobilityApplicationActor;
import MicroTAPAS.debug.DebugEvent;
import MicroTAPAS.util.Util;

import java.util.Hashtable;
import java.util.Vector;
import java.util.Enumeration;
import java.awt.Color;

// }}}RME classifier 'Logical View::PatientRecords::v1_1::PatientRecordsClient' tool 'RTJava' property 'ClassFileHeader'
// {{{RME classifier 'Logical View::PatientRecords::v1_1::PatientRecordsClient'
public class PatientRecordsClient
        extends MicroTAPAS.mobility.MobilityApplicationActor
        implements java.io.Serializable {
        // {{{RME classAttribute 'initiated'
        private boolean initiated = false;
        // }}}RME classAttribute 'initiated'
        // {{{RME classAttribute 'ai'
        public PatientRecordsClientInterface ai;
        // }}}RME classAttribute 'ai'
        // {{{RME classAttribute 'stInitial'
        private static final int stInitial = 0;
        // }}}RME classAttribute 'stInitial'
        // {{{RME classAttribute 'stInitialized'
        private static final int stInitialized = 1;
        // }}}RME classAttribute 'stInitialized'
        // {{{RME classAttribute 'stLoggedOn'
        private static final int stLoggedOn = 3;
        // }}}RME classAttribute 'stLoggedOn'
        // {{{RME classAttribute 'stWaitPatientSelected'
        private static final int stWaitPatientSelected = 4;
        // }}}RME classAttribute 'stWaitPatientSelected'
        // {{{RME classAttribute 'initialRoleSession'
        private RoleSession initialRoleSession = null;
        // }}}RME classAttribute 'initialRoleSession'
        // {{{RME classAttribute 'bgColor'
        private Color bgColor = new Color(130, 165, 124);
        // }}}RME classAttribute 'bgColor'
        // {{{RME classAttribute 'low'
        private LogOnWindow low;
        // }}}RME classAttribute 'low'
        // {{{RME classAttribute 'configuration'
        private PatientRecordsConfiguration configuration;
        // }}}RME classAttribute 'configuration'
        // {{{RME classAttribute 'server'
        private RoleSession server;
        // }}}RME classAttribute 'server'
        // {{{RME classAttribute 'stWaitLogon'
        private static final int stWaitLogon = 2;
        // }}}RME classAttribute 'stWaitLogon'
        // {{{RME classAttribute 'stWaitRecordSelected'
```

```
private static final int stWaitRecordSelected = 5;
// }}}RME classAttribute 'stWaitRecordSelected'
// {{{RME classAttribute 'stRecordSelected'
private static final int stRecordSelected = 6;
// }}}RME classAttribute 'stRecordSelected'
// {{{RME classAttribute 'stNoteOpen'
private static final int stNoteOpen = 7;
// }}}RME classAttribute 'stNoteOpen'
// {{{RME classAttribute 'mainWindow'
private ClientMainWindow mainWindow;
// }}}RME classAttribute 'mainWindow'
// {{{RME classAttribute 'stPatientSelected'
private static final int stPatientSelected = 8;
// }}}RME classAttribute 'stPatientSelected'
// {{{RME classAttribute 'recordWindow'
private RecordWindow recordWindow;
// }}}RME classAttribute 'recordWindow'
// {{{RME classAttribute 'title'
private String title = "PatientRecords";
// }}}RME classAttribute 'title'
// {{{RME classAttribute 'info'
private InfoWindow info;
// }}}RME classAttribute 'info'
// {{{RME classAttribute 'ui'
private RoleSession ui;
// }}}RME classAttribute 'ui'
// {{{RME operation 'PatientRecordsClient()'
public PatientRecordsClient() {
        ai = new PatientRecordsClientInterface();
        ai.state = stInitial;

}
// }}}RME operation 'PatientRecordsClient()'
// {{{RME operation 'mobilityActorEntry(RequestPars)'
public RequestResult mobilityActorEntry(RequestPars rp) throws
// {{{RME tool 'RTJava' property 'JavaThrows'
Exception
// }}}RME tool 'RTJava' property 'JavaThrows'
{
        if (DEBUG)
                debug.send(
                        new DebugEvent(
                                this,
                                DebugEvent.INFODEBUG,
                                "mobilityActorEntry()",
                                rp));

        // ActorPlugIn
        if (rp.requestType == RequestPars.ActorPlugIn) {
                if (!initiated) {
                        return stateTransition(rp);
                } else {
                        return new RequestResult(
                                RequestResult.OK,
                                "Actor already initiated",
                                rp.ID);
                }
        }
        // ActorPlugOut
        else if (rp.requestType == RequestPars.ActorPlugOut) {
                return stateTransition(rp);
        }
        // ActorChangeBehaviour
        else if (rp.requestType == RequestPars.ActorChangeBehaviour) {
                return stateTransition(rp);
        }
        // RoleSessionAction
        else if (rp.requestType == RequestPars.RoleSessionAction) {
                return stateTransition(rp);
        }
        // FAIL
        else {
            debug.send(
                new DebugEvent(
                    this,
                    DebugEvent.FAULT,
                    "mobilityActorEntry()",
                    "Invalid request: " + RequestPars.RT[rp.requestType]));
            return new RequestResult(
                RequestResult.FAIL,
                super.context.self
                    + "::[mobilityActorEntry()] Unknown request type: "
                    + RequestPars.RT[rp.requestType],
                rp.ID);
        }

    }
    // }}}RME operation 'mobilityActorEntry(RequestPars)'
    // {{{RME operation 'stateTransition(RequestPars)'
    public RequestResult stateTransition(RequestPars pRP) throws
    // {{{RME tool 'RTJava' property 'JavaThrows'
    Exception
    // }}}RME tool 'RTJava' property 'JavaThrows'
    {
        if (DEBUG)
            debug.send(
                new DebugEvent(
                    this,
                    DebugEvent.INFODEBUG,
                    "stateTransition()",
                    pRP));
        RequestResult rr =
            new RequestResult(
                RequestResult.OK,
                this.getClass().getName()
                    + "::stateTransition: state="
                    + ai.state,
                pRP.ID);
        ApplicationMessage appMsg = null;
        ActorPlugInReq request;

        switch (ai.state) {

            case stInitial :
                if (ActorContext.microPNES != null) {
                    // Define configuration to be used by the application
                    configuration =
                        new PatientRecordsConfiguration(
                            context.play.playLoc + context.play.playId,
                            context);
                    initialRoleSession = (context.rsc.initialRoleSession());
                    rr =
                        new RequestResult(
                            RequestResult.OK,
                            "ActorPlugIn was successful",
                            pRP.ID);
                    if (pRP.isMoved) {
                        // Stay in this state. When the interface is recreated, the actor will
be recreated in its

                        // original state
                        ai.state = stInitial;
                    } else {
                        // Plug in user interface
                        // User interface shall run at same Node and PNES, but with
another identifier

                        GAI uii =
                            new GAI(
                                context.self.getType(),
                                context.self.getNode(),
                                context.self.getPNES(),
                                "PatientRecordsUI");
                        request =
                            new ActorPlugInReq(
                                uii,
                                new Role("PatientRecordsUI"));

                        // Actor plug in
                        rr = actorPlugIn(request);
                        ui = rr.roleSession;
                        ai.state = stInitialized;
                    }

                } else {
                    rr =
                        new RequestResult(
                            RequestResult.FAIL,
                            "ParentPNES is null",
                            pRP.ID);
                }
                break;

            case stInitialized :
                if (pRP.requestType == RequestPars.RoleSessionAction) {
                    appMsg = pRP.applicationMessage;
                    if (appMsg.messageType.compareToIgnoreCase("exit") == 0) {
                        try {
                            rr = super.actorPlugOut(initialRoleSession, false);
                        } catch (Exception e) {
                            e.printStackTrace();
                            rr =
                                new RequestResult(
                                    RequestResult.FAIL,
                                    "Exception: " + e.toString(),
                                    0);
                        }
                    } else if (
                        appMsg.messageType.compareToIgnoreCase("logon") == 0) {
                        // Build parameters for Server plug in
                            request =
                        // NOTE: Now only location and role are specified - other must be
completed

                        new ActorPlugInReq(configuration.locationServer(),
                            new Role("PatientRecordsServer"));

                        // Actor plug in
                        rr = actorPlugIn(request);
                        server = rr.roleSession;

                        // Send message
                        rr =
                            sendMessage(
                                server.cooperator,
                                "logonReq",
                                appMsg.message);
                        ai.state = stWaitLogon;
                    }

                }
                break;

            case stWaitLogon :
                if (pRP.requestType == RequestPars.RoleSessionAction) {
                    appMsg = pRP.applicationMessage;
                    if (appMsg.messageType.compareToIgnoreCase("logonDenied")
                        == 0) {
                        // Send message
                        rr =
                            sendMessage(
                                ui.cooperator,
                                "OpenInfoWindow",
                                new String[] { "Login failed. Wrong user name
or password." });
                        ai.state = stInitialized;
                    } else if (
                        appMsg.messageType.compareToIgnoreCase("logonCnf")
                            == 0) {
```

```
                        // Send message
                        rr =
                            sendMessage(
                                ui.cooperator,
                                "closeWindow",
                                new String[] { "logon" });
                        rr =
                            sendMessage(
                                ui.cooperator,
                                "openMainWindow",
                                appMsg.message);
                        ai.state = stLoggedOn;
                    }
                }
                break;

            case stLoggedOn :
                if (pRP.requestType == RequestPars.RoleSessionAction) {
                    appMsg = pRP.applicationMessage;
                    if (appMsg.messageType.compareToIgnoreCase("selectPatient")
                        == 0) {
                        rr =
                            sendMessage(
                                server.cooperator,
                                "getRecords",
                                appMsg.message);
                        ai.state = stWaitPatientSelected;

                    } else if (
                        appMsg.messageType.compareToIgnoreCase("exit") == 0) {
                        try {
                            rr = super.actorPlugOut(initialRoleSession, false);
                        } catch (Exception e) {
                            e.printStackTrace();
                            rr =
                                new RequestResult(
                                    RequestResult.FAIL,
                                    "Exception: " + e.toString(),
                                    0);
                        }

                    } else if (
                        appMsg.messageType.compareToIgnoreCase("newPatient")
                            == 0) {

                    } else if (
                        appMsg.messageType.compareToIgnoreCase("moveClient")
                            == 0) {
                        GAI newGAI =
                            New
GAI("Actor://10.2.1.8/MicroPNES/PatientRecordsClientM");
                        rr =
                            sendMessage(
                                ui.cooperator,
                                "moveUserInterface",
                                new String[] { newGAI.toString()});
                        actorMove(newGAI);

                    } else if (
                        appMsg.messageType.compareToIgnoreCase("selectRecord")
                            == 0) {
                        rr =
                            sendMessage(
                                server.cooperator,
                                "getPatientRecord",
                                appMsg.message);
                        ai.state = stWaitRecordSelected;

                    } else if (
                        appMsg.messageType.compareToIgnoreCase("addNote")
                            == 0) {
                        rr =
                            sendMessage(
                                server.cooperator,
                                "addNote",
                                appMsg.message);
                        ai.state = stRecordSelected;

                    } else if (
                        appMsg.messageType.compareToIgnoreCase("logOff")
                            == 0) {
                        rr =
                            sendMessage(ui.cooperator, "closeAllWindows", null);
                        rr =
                            sendMessage(ui.cooperator, "openLogonWindow", null);
                        ai.state = stInitialized;

                    }

                }
                break;

            case stWaitPatientSelected :
                if (pRP.requestType == RequestPars.RoleSessionAction) {
                    appMsg = pRP.applicationMessage;
                    if (appMsg
                        .messageType
                        .compareToIgnoreCase("patientRecords")
                        == 0) {
                        rr =
                            sendMessage(
                                ui.cooperator,
                                "displayPatientRecords",
                                appMsg.message);
                    } else if (
                        appMsg.messageType.compareToIgnoreCase("noRecord")
                            == 0) {
                        // Error message
                    }
                    ai.state = stLoggedOn;
                }
                break;

            case stWaitRecordSelected :
                if (pRP.requestType == RequestPars.RoleSessionAction) {
                    appMsg = pRP.applicationMessage;
                    if (appMsg
                        .messageType
                        .compareToIgnoreCase("recordContents")
                        == 0) {
                        rr =
                            sendMessage(
                                ui.cooperator,
                                "openRecordWindow",
                                appMsg.message);
                        ai.state = stLoggedOn;
                    } else if (
                        appMsg.messageType.compareToIgnoreCase("noContents")
                            == 0) {

                    }
                }
                break;

            default :
                debug.send(
                    new DebugEvent(
                        this,
                        DebugEvent.FAULT,
                        "applicationActorEntry()",
                        "Invalid state encountered in actor HospitalClient State:"
                            + ai.state
                            + "message:"
                            + appMsg));
                return (
                    new RequestResult(
                        RequestResult.FAIL,
                        "Invalid state: " + ai.state,
                        pRP.ID));
        }
        if (DEBUG)
            debug.send(
                new DebugEvent(
                    this,
                    DebugEvent.INFODEBUG,
                    "stateTransition()",
                    rr));
        return rr;

    }
    // }}}RME operation 'stateTransition(RequestPars)'
    // {{{RME operation 'getInterface()'
    public Object getInterface() {

        return (Object) ai;

    }
    // }}}RME operation 'getInterface()'
    // {{{RME operation 'createInterface(Object)'
    public void createInterface(Object actorInterface) {
        // Update the actors interface

        if (actorInterface != null) {
            PatientRecordsClientInterface tmp =
                (PatientRecordsClientInterface) actorInterface;
            System.out.println(
                context.self + "::createActorInterface(): ai=" + ai.toString());
            if (DEBUG)
                debug.send(
                    new DebugEvent(
                        this,
                        DebugEvent.INFODEBUG,
                        "createInterface()",
                        ai.toString()));
            ai.state = tmp.state;
        } else {
            debug.send(
                new DebugEvent(
                    this,
                    DebugEvent.WARNING,
                    "createInterface()",
                    "ActorInterface = null"));
            System.out.println(
                context.self + "::WARNING::ActorInterface = null");
        }

    }
    // }}}RME operation 'createInterface(Object)'
    // {{{RME operation 'sendMessage(GAI,String,String[])'
    public RequestResult sendMessage(
        GAI client,
        String messageType,
        String[] message) {
        RequestPars rp =
            new RequestPars(
                RequestPars.RoleSessionAction,
                context.self,
                client);
        rp.applicationMessage = new ApplicationMessage(messageType, message);
        return super.requestToActor(rp);

    }
    // }}}RME operation 'sendMessage(GAI,String,String[])'
}
// }}}RME classifier 'Logical View::PatientRecords::v1_1::PatientRecordsClient'
```

Appendix B: XML description for the PatientRecordsClient

This appendix contains a XML manuscript for the PatientRecordsClient which is transferred from the UML state diagram description of the Role Figure. The mapping is done according to the procedure described in section 5.4.1.2.

```
<manuscript>

  <!--descripton of the manuscript for the role PatientRecordsClient-->
  <fsm name="PatientRecordsClient">

    <data>
        <name> v_interface </name>
        <type> RoleSession </type>
    </data>
    <data>
        <name> v_server </name>
        <type> RoleSession </type>
    </data>

    <init_state> stInit </init_state>

    <!--Description of state stInit (initial state)-->
    <state name="stInit">

        <input msg="INITIAL_TRANSITION">

        <!--Configuration is now specified by play
       configuration rules. No setConfiguration
       action is therefore specified in the
       manuscript... -->

            <action>
                <meth_name> conditionalJump </meth_name>
                <param>
                    <name> variable </name>
                    <value> isMoved </value>
                </param>
                <param>
                    <name> value </name>
                    <value> true </value>
                </param>
            </action>

            <action>
                <meth_name> conditionalJump </meth_name>
                <param>
                    <name> variable </name>
                    <value> isMoved </value>
                </param>
                <param>
                    <name> value </name>
                    <value> false </value>
                </param>
                <param>
                    <name> gotoSubtrans </name>
                    <value> Continue </value>
                </param>
            </action>

        </input>
    </state>


    <state name="stInitialized">
```

```
        <input msg="Logon">

            <action>
                <meth_name> ActorPlugInReq </meth_name>
                <param>
                    <name> role </name>
                    <value> PatienRecordsServer </value>
                </param>
                <store_return> v_server </store_return>
            </action>

            <output>
                <msg type="logonReq">
                    <param>
                        <name> message </name>
                        <value> INPUT_MSG </value>
                    </param>
                    <dest> server </dest>
                </msg>
            </output>

            <next_state> stWaitLogOn </next_state>
        </input>
    </state>

    <state name="stWaitLogOn">
        <input msg="LogonCnf">
            <output>
                <msg type="CloseWindow">
                    <param>
                        <name> windowType </name>
                        <value> Logon </value>
                    </param>
                    <dest> v_user_interface </dest>
                </msg>
                <msg type="OpenMainWindow">
                    <dest> v_user_interface </dest>
                </msg>
            </output>
            <next_state> stLoggedOn </next_state>
    </input>

    <input msg="LogonDenied">
        <output>
            <msg type="OpenInfoWindow">
                <param>
                    <name> text </name>
                    <value> Login failed. Wrong user name or password </value>
                </param>
                <dest> v_user_interface </dest>
            </msg>
        </output>

        <next_state> stInitialized </next_state>
    </input>

    <input msg="UNDEFINED">
    </input>
</state>


<state name="stLoggedOn">
    <input msg="selectPatient">
        <output>
            <msg type="getRecords">
                <param>
                    <name> message</name>
                    <value> INPUT_MSG </value>
                </param>
                <dest> v_server</dest>
            </msg>
        </output>
        <next_state> stWaitPatienSelect </next_state>
    </input>

    <input msg="selectRecord">
        <output>
            <msg type="getPatientRecord">
                <param>
                    <name> message </name>
                    <value> INPUT_MSG </value>
                </param>
                <dest> v_server </dest>
            </msg>

            <next_state> stWaitPatientSelected </next_state>
        </output>
    </input>

      <input msg="addNote">
        <output>
            <msg type="addNote">
                <param>
                    <name> message </name>
                    <value> INPUT_MSG </value>
                </param>
                <dest> v_server </dest>
            </msg>

            <next_state> stWaitRecordSelected </next_state>
        </output>
    </input>

        <input msg="logOff">
        <output>
            <msg type="closeAllWindows">
                <param>
                    <name> message </name>
                    <value> INPUT_MSG </value>
                </param>
                <dest> v_user_interface </dest>
            </msg>
        </output>

        <output>
            <msg type="openLogonWindow">
                <param>
                    <name> message </name>
                    <value> INPUT_MSG </value>
                </param>
                <dest> v_user_interface </dest>
            </msg>

            <next_state> stInitialized </next_state>
        </output>
    </input>

    <input msg="UNDEFINED">
    </input>
</state>

<state name="stWaitPatientSelect">
    <input msg="patientRecords">
        <data>
            <name> v_temp </name>
            <type> boolean </type>
        </data>

        <output>
            <msg type="displayPatientRecords">
                <param>
                    <name> message </name>
                    <value> INPUT_MSG </value>
                </param>
                <dest> v_interface </dest>
            </msg>
        </output>
        <next_state> stLoggedOn </next_state>
    </input>

    <input msg="noRecord">
        <output>
            <msg type="openInfoWindow">
                <param>
                    <name> text </name>
                    <value> No records found for this patient </value>
                </param>
                <dest> v_user_interface </dest>
            </msg>
        </output>
        <next_state> stLoggedOn </next_state>
    </input>

    <input msg="UNDEFINED">
    </input>
</state>

<state name="stWaitRecordSelected">
    <input msg="recordContents">
        <output>
            <msg type="openRecordWindow">
                <param>
                    <name> message </name>
                    <value> INPUT_MSG </value>
                </param>
                <dest> v_user_interface </dest>
            </msg>
        </output>
    </input>
    <input msg="noContents">
        <output>
            <msg type="openInfoWindow">
                <param>
                    <name> text </name>
                    <value> Empty record </value>
                </param>
                <dest> v_user_interface </dest>
            </msg>
        </output>
    </input>

    <next_state> stLoggedOn </next_state>

</state>

 <subtransition name="Continue">
    <!--Asks for an actor to play the role
           PatientRecordsUI-->
            <action>
                <meth_name> ActorPlugInReq </meth_name>

                <param>
                    <name> role</name>
                    <value> PatientRecordsUI </value>
                </param>
                <param>
                    <name> location</name>
                    <!-- THIS refers to the same role as the node
                          where this  actor (PatientRecordsClient) is installed
                 -->
                    <value> THIS </value>
                </param>
                <store_return> v_interface </store_return>
            </action>
            <next_state> stInitUserInterface </next_state>
 </subtransition>

</fsm>
</manuscript>
```

Appendix C: PatienRecords sequence diagrams

This appendix contains sequence diagrams which describes alternative flows for the PatienRecords application.

Appendix D: CD

The CD provided with this thesis contains the following directories:

\model – this directory contains the UML model – Model.rtmdl

\model\PaP – generated source code for the basic TAPAS support platform.

\model\School – generated source code for the Tele School application.

\model\MicroTAPAS – generated source code for MicroTAPAS.

\model\PatientRecords – generated source code for the PatientRecords application.