Using J2EE Technologies for Implementation of ActorFrame Based UML2.0 Models

of

Geir Melby

Masters Thesis in
Information and Communication Technology

Agder University College
Grimstad, May 2003

Summary

Ericsson has developed a prototype of a Java framework called ActorFrame, for development and execution of services. The services will be deployed in networks where current Telecommunication and Internet has merged into an open service oriented network. The services are modeled using UML 2.0 concepts for concurrent state machines communicating asynchronously through message passing. ActorFrame has been used in development of prototype services deployed in real networks as part of the AVANTEL research project. The background for this thesis study is that Ericsson wants to move ActorFrame to a J2EE technology.

This report gives an introduction to the core technologies of interest: J2EE, ActorFrame and UML2.0. J2EE is a set of API's where the component technology EJB, name server JNDI and the message system JMS are the most central technologies. A new version of UML, called UML2.0, has been worked out by a consortium where Ericsson has been an active partner. The basic concepts of UML2.0 that support the concept-oriented approach in ActorFrame based models, is presented.

There exist different conceptual approaches to the problem of mapping platform independent models to concrete implementations. The approach taken in this thesis is a concept-oriented approach where domain specific concepts are used in the modeling of services. The concepts are defined as stereotypes in UML and a specific UML profile for ServiceFrame are proposed as part of this work.

The principle mapping of these concepts to the J2EE technologies are discussed, which forms a background for the design of a prototype of a framework for EJB. The framework is used in implementation of services by extending the classes defined in the framework. The solution was validated through implementation of an example that was run on a J2EE application server.

The main conclusion from this thesis work is that ActorFrame models can be implemented on J2EE application servers in a beneficial way. The platform supports asynchronous communication and it is possible to combine this communication style with implementation of persistent state machines.

The J2EE platform gives additional benefits compared to the current implementation of ActorFrame such as integration of Web services, increased management support, flexible distributions of actors, scalable application platforms, and transaction services.

The thesis work has also shown that there exists many mapping solutions and not all mapping issues have been elaborated. However this work should form a basis for further work with frameworks for service development and execution.

Preface

This thesis was written for Ericsson NorARC (Norwegian Applied Research Center) and it is a part of a Norwegian master degree. The work has been carried out in the period January 2003 and May 2003. The thesis is a part of the "mobile student" and AVANTEL research projects.

Ericsson has evaluated the thesis regarding patent ability and has decided to file three patent applications.

I would like to thank my supervisors, Knut Eilif Husa at Ericsson NorARC, and Jan P. Nytun at Agder Univeristy College for valuable help and inspiration. I would also like to thank Øystein Haugen, Birger Møller-Pedersen, Stein Bergsmark, Egil Ofstad, Jacqueline Floch, Kristen Hestir and Rolv Bræk for their valuable comments during my work.

Grimstad, May 2003

Geir Melby

Table of content

Summary		**I**
Preface		**III**
Table of content		**V**
Chapter 1	**Thesis background and problem definition**	**1**
1.1	Services	1
1.2	ServiceFrame - a service creation and execution environment	3
1.3	UML2.0 and MDA – a modeling driven approach	4
1.4	Java 2 Enterprise Edition, Technologies and distributed services	4
1.5	Characterisation of services	5
1.6	The definition of the problem	6
1.7	The thesis work	7
1.8	Reader's guide	7
Chapter 2	**UML2.0**	**9**
2.1	Introduction	9
2.2	Major improvements in UML2.0	9
2.3	Architectural concepts in UML	10
2.3.1	*Parts*	*10*
2.3.2	*Ports*	*12*
2.3.3	*Connectors*	*12*
2.4	State machines in UML2.0	13
2.4.1	*Composite state*	*14*
2.4.2	*Generalization of behavior*	*15*
2.5	UML profiles	16
2.6	Summary	17
Chapter 3	**Ericsson's service creation architectures**	**19**
3.1	ServiceFrame	19
3.2	ActorFrame	20
3.2.1	*Actor*	*20*
3.2.2	*ActorFrame protocol*	*21*
3.3	JavaFrame	23
3.3.1	*Composite*	*23*
3.3.2	*StateMachine*	*24*
3.3.3	*Mediators*	*24*
3.3.4	*Runtime systems*	*25*
3.4	Summary	25
Chapter 4	**J2EE – Java 2 Enterprise Edition**	**27**
4.1	Introduction to the different J2EE technologies	27
4.2	Enterprise Java Beans (EJB)	28
4.2.1	*Introduction to EJB*	*28*

4.2.2	*EJB Architecture*	*28*
4.2.3	*Enterprise beans*	*30*
4.3	Enterprise bean environment	33
4.3.1	*Environment entries*	*33*
4.3.2	*EJB references*	*33*
4.3.3	*Web service references*	*34*
4.3.4	*Message destination references*	*34*
4.3.5	*Deployment descriptors*	*34*
4.4	Java Messaging System	35
4.4.1	*Message Oriented Middleware*	*35*
4.4.2	*Java Message Service*	*35*
4.4.3	*Integration of JMS into J2EE*	*36*
4.5	Java Naming and Directory Interface	37
4.5.1	*Naming service*	*37*
4.5.2	*Integration of JNDI and J2EE*	*37*
4.6	Web-services	37
4.6.1	*Introduction*	*37*
4.6.2	*Web service technology stack*	*38*
4.6.3	*Web services in J2EE*	*39*
4.7	Summary	39
Chapter 5	**Conceptual approach to mapping of models**	**41**
5.1	Model oriented approach	41
5.2	Translation of models	43
5.3	UML profile for Entreprise Java Beans	45
5.3.1	*Introduction*	*45*
5.3.2	*UML profile for EJB*	*45*
5.3.3	*Limitation of the EJB profile*	*47*
5.4	Use of UML profile for EJB for ActorFrame modeling	48
5.5	Summary	49
Chapter 6	**Mapping ActorFrame to J2EE**	**51**
6.1	Mapping strategies	51
6.2	Mapping issues	51
6.3	Strategies for mapping of state machines	52
6.3.1	*Multiple state machines in one bean*	*52*
6.3.2	*One state machine in one bean*	*54*
6.3.3	*Group of state machines in one bean*	*54*
6.4	Kind of bean to be used	55
6.4.1	*Session beans*	*55*
6.4.2	*Entity bean*	*56*
6.4.3	*Message bean*	*56*
6.5	Asynchronous communication	56
6.5.1	*All state machines share the same queue*	*57*
6.5.2	*One queue for each state machine*	*58*
6.5.3	*Combination of alternative 1 and 2*	*58*
6.6	Structural relations	59

6.6.1	*Parts*	*59*
6.6.2	*Associations*	*60*
6.6.3	*Connectors*	*60*
6.6.4	*Ports*	*60*
6.7	Naming of state machines	61
6.8	Summary	61
Chapter 7	**Framework for implementation of actors on J2EE**	**63**
7.1	The mapping solution	63
7.2	UML profile for ActorFrame	65
7.2.1	*Actor*	*66*
7.2.2	*ActorMsg*	*67*
7.2.3	*ActorAddress*	*67*
7.3	Principle behavior of EJBActorFrame	67
7.4	Description of EJBActorFrame	68
7.5	The implementation of state machine as a message bean	70
7.6	Persistence of state data	71
7.7	Use of JMS	72
7.8	Use of JNDI	73
7.9	Summary	74
Chapter 8	**Implementation of a service using EJBActorFrame**	**75**
8.1	Traffic news – a context aware service	75
8.2	Design of CAS	76
8.3	Implementation of CAS	80
8.3.1	*Mapping of behavior of state machines*	*81*
8.3.2	*Mapping of state data*	*83*
8.3.3	*Use of JMS and JNDI*	*85*
8.3.4	*Implementation of parts*	*86*
8.4	Deployment of CAS	87
8.5	Summary	88
Chapter 9	**Discussion and conclusion**	**89**
9.1	Concurrency and asynchronous communication	89
9.2	Aggregation of actors	90
9.3	Persistent state machines	90
9.4	Actor modeling using UML profiles for EJB	91
9.5	Other issues	93
9.6	Future work	94
9.7	Conclusion	94
Chapter 10	**Abbreviations**	**97**
Chapter 11	**References**	**99**
Chapter 12	**List of figures**	**103**
Appendix A	**Source code for EJBActorFrame (confidential)**	**CDROM**
Appendix B	**Source code for CAS example**	**CDROM**

Chapter 1 Thesis background and problem definition

1.1 Services

The combination of mobility and Internet is creating a new and powerful industry that will deliver attractive, content-rich services to users on the move. All over the world, companies are preparing for the Mobile Internet. Mobile data networks (UMTS, WLAN and Bluetooth) with increasing bandwidth, advanced phones and handheld computers are available bringing a new generation of services into use. For instance, the introduction of I-mode in Japan has been a tremendous success with millions of subscribers and thousands of service providers that have created market demand for services available through mobile phones.

The expectation of strong growth in the mobile area is one reason for the establishment of the Open Mobile Alliance (OMA), which states on its home page [41] "The mobile industry has experienced a period of very exceptional growth during the past ten years. The next wave of growth is expected to come from mobile services".

Until now telecom operators have dominated as service providers for the telecom market. But deregulation of the telecom sector and requirements from application providers have gradually opened the telecom networks for 3rd parties. 3GPP [42] has specified a set of API's called Open Service Access (OSA), which give 3rd party application providers access to the resources and services of the telecom networks. OSA enables telecom services to be integrated with existing Internet applications. This integration has formed an industry, called Information and Communication Technology (ICT), which includes network operators, service providers, equipment vendors, application developers and content providers.

Telecom operators as part of the ICT industry have specified and partly developed the next generation network as shown in Figure 1-1. The network architecture consists logically of an access, a control and a service layer. These layers are connected to each other through a high capacity backbone network that is based on the IP protocol.

The access layer consists of different types of access systems such as mobile networks (GSM, UMTS), wireless networks (WLAN) and ADSL that are connected to the same backbone network. Terminals as phones, PCs and faxes are connected to the different access networks. The control layer consists of different servers as controllers and databases for network resources. These servers provide telecom services such as basic call set up and traffic control.

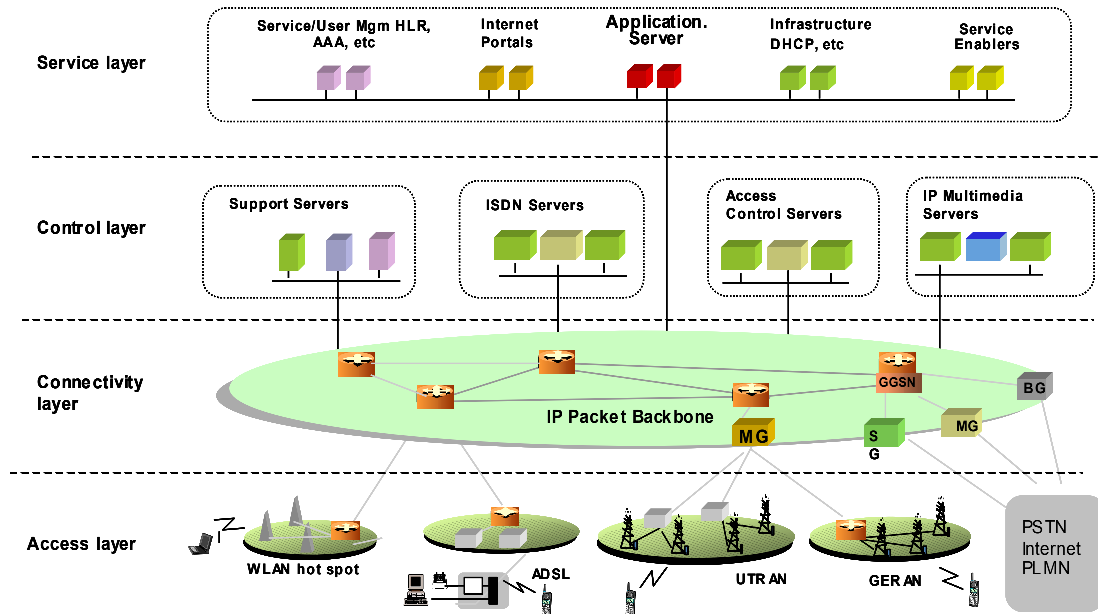

Figure 1-1 Next generation network

The service layer consists of the resources needed to perform additional control and the service logic, which provides value added services to the end user. Service Capability Servers (SCS) provide access to services in the control layer such as basic call setup. Other components are user and service databases (HLR, AAA), Internet portals and application content servers. These components also communicate with resources in the controller layer through the IP backbone network. This layer constitutes the service network and it will normally be connected to the Internet.

The Application Server (AS) contains the applications that provide services for end users or clients. Application servers normally consist of tools for creation, deployment and management of services. Web-logic [30] and JAMBALA [31] are examples of application servers and they also support software standards as Web services [32] and J2EE [33].

Application Servers may also provide service access to the Internet through open Internet APIs such as Web-services. Application servers may also have open access to Internet enabling applications for access and utilization of other services on the Internet. In this way the service network can be integrated with Internet services.

Due to lack of profitable business cases for the introduction of UMTS, it has become urgent for operators to provide content that users are willing to pay for. Companies in the ICT industry have recognized this, and Lars Borman from Ericsson said in [26] "They must quickly be able to implement new services and make them available to the mass-market".

To meet this requirement Ericsson, Telenor Research, and NTNU are cooperating in the research projects AVANTEL [34] and ARTS [35], on how to make new services for the service network. A service network with a platform for execution of services has

been established, and 3rd party application developers have been invited to experiment with the service network.

As part of the AVANTEL project, Ericsson in Norway has developed a prototype of a application server, called ServiceFrame [1]. ServiceFrame has been used by the participants in the research project and students at NTNU to develop and execute prototypes of services.

1.2 ServiceFrame - a service creation and execution environment

ServiceFrame is an application server in the service network. It provides functionality for communication with users connected through different types of terminals such as phones, PC's or PDA's. It also provides access to network resources through the OSA API, which enable services to set up phone calls between users.

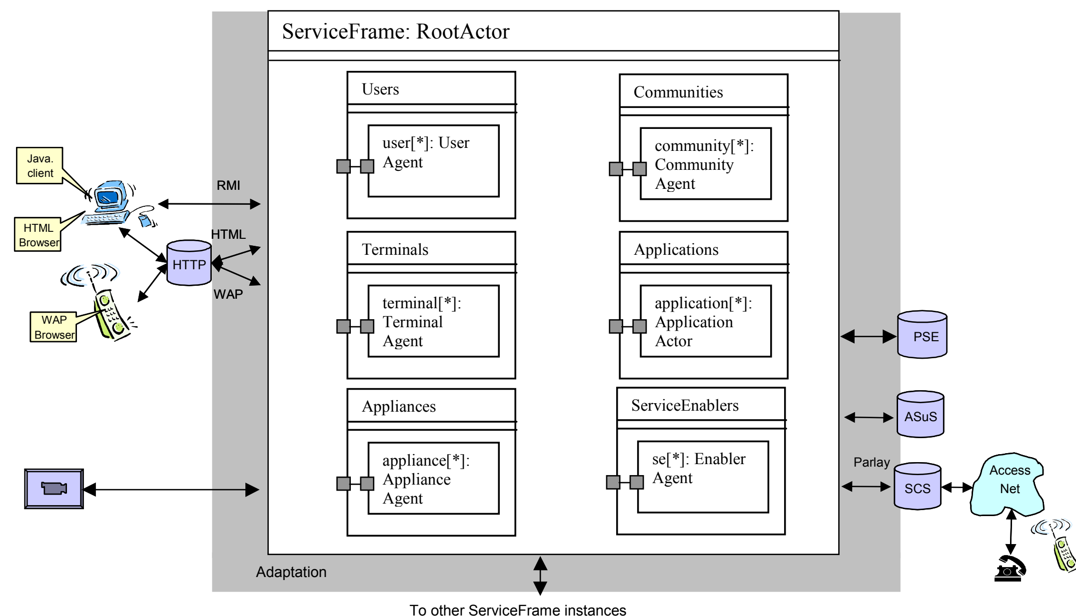

Figure 1-2 ServiceFrame [1]

ServiceFrame itself provides architectural support for service creation, service deployment and service execution, but it does not provide any end user services. Services are realized by ServiceFrame applications that are defined by specializing and instantiating framework classes. In addition it has mechanisms that support incremental development and deployment of services.

ServiceFrame is layered on top of ActorFrame and JavaFrame. ActorFrame is a generic application framework that supports the concept of actors and roles [22]. With ActorFrame actors play roles and involve other actors to play other roles using a role request protocol. Actors may contain other actors. JavaFrame is both an execution environment and a library of classes used to implement concurrent state machines and asynchronous communication between state machines.

ServiceFrame can be used as an application server in the service network as described in the previous chapter, but it does not offer all functionality that a commercial application servers usually have such as management functionality, data base storage etc. ServiceFrame interacts with the different resources in the service network such as Parlay Service Capability Server as described in Figure 1-2. ServiceFrame also communicates with clients and other servers through the Internet. ServiceFrame has been used in the AVANTEL project to make services that are deployed and run in the service network.

1.3 UML2.0 and MDA – a modeling driven approach

Ericsson and other telecom companies like Motorola and Nokia have used SDL [20] in design of telecom products. It has been possible to make a functional model of the system, which could be used for formal analysis, verification and validation, and for automatic generation of code. This model-oriented approach has proven to be successful in development of complex real time systems like telecom systems. UML has until now lacked the concepts and formalism that have made SDL successful in the telecom industry. UML has come from a background of enterprise applications where database modeling has been important but with little demand and support for formal behavior modeling. Today UML has become the defacto standard modeling language used in the software community.

With an active push and participation from leading telecom companies like Motorola, Siemens and Ericsson, the next version of UML has taken a long step in the direction of SDL. The proposal for UML2.0 [18], which will be voted on in the OMG [36] meeting in June 2003, has the language concepts and formalism needed to support a model driven approach to development of telecom systems.

OMG has also recently changed the focus from CORBA [37] that was intended to integrate heterogeneous systems, to UML as the core language to be used in a modeling approach called Model Driven Architecture (MDA). MDA defines two different UML models.

1. PIM is a platform independent UML model, which can be reused in different implementations.
2. PSM is a platform specific model that is tailored to an actual middleware platform.

The idea is to first make a PIM model and then transform it to a PSM model. A PSM model can then be automatically or manually transformed to an implementation using a specific platform like J2EE or .NET [38].

1.4 Java 2 Enterprise Edition, Technologies and distributed services

J2EE technologies are the defacto standard used for development and execution of enterprise applications in the Java community. J2EE provides today a rather complete set of technologies for development of server side applications. That includes enterprise java beans (EJB) for development of distributable components, Java Naming

Directory Interface (JNDI) for lookup of references to components, Remote Methods Invocation (RMI) for transparent communication between distributed objects or components, Java Transaction API (JTA) for transactional support including roll-back and persistent storage and XML based technologies. The latest J2EE standard includes also a Java Messaging Service (JMS) for support of asynchronous communications and support for applying web-services technologies. Together with support for integration of client technologies as Java Server Pages (JSP) and Servlets, J2EE can be used for implementation of software applications that are distributed, scalable, transactional and persistent. However J2EE is originally a typical client server technology. Clients make request for information and servers respond with the requested information (hopefully).

As described earlier application servers are used for execution of services in the service network. 3GPP is now working with standardization of access to network resources via an API based on Web-services. Web services are supported by the latest version of J2EE 1.4. Application servers that support J2EE contain technologies that are necessary for building large and scaleable applications with high availability. Ericsson supports J2EE technologies for instance in their JAMBALA application server.

1.5 Characterisation of services

Services provided by applications that are available through the Internet, are mainly based on single initiatives from clients. A service is requested by a user via a client to a server, which responses to the request by accessing databases and sending information back to the client. The client server technology as provided by J2EE and http servers are typically used to implement such services. The tremendous success of the Internet is proof that this request and response approach works.

Future services will be built on an integration of services provided by various, remote computers. These will lead to loosely coupled computers physically spread around the world with variable and considerable latency in communication. This will probably cause increasing scaling and capacity problems in the applications servers if current client server technology is used. The use of synchronized communication between loosely coupled components that are physically and globally distributed does not function very well.

The service network including Internet may consist of active components, meaning that the components can act on their own. For instance, multiple sensors can cause events to be sent simultaneously causing conflicting requests to the services. This will cause concurrency problems if the services are implemented with technologies that are based on synchronized communication between the active components. Higher latency will increase the "window" for concurrency conflicts. Slow and non functioning services may be the result. It is therefore important that application platforms can handle conflicting initiatives to the same applications from many sources simultaneously.

These problems have been addressed by research projects. Khare [27] argues for use of asynchronous messages (events) for communication between what he calls network services. Network services are characteristically decentralized, which means that they

are crossing organizational boundaries. Network services are also loosely coupled and are typically implemented with different technologies.

Different services may therefore require different implementation techniques. Client server technology works well for implementation of many Internet services today, but client server technology fails when it is used for implementation of services that are characterized by decentralization, loose coupling and conflicting initiatives.

1.6 The definition of the problem

The main objective for this thesis is to investigate how the UML2.0 concepts that ActorFrame are based upon can be implemented and deployed by using middleware platforms that support J2EE technologies.

The most important feature of ActorFrame is the support of state machines that behave concurrently and communicate asynchronously. As presented earlier in this chapter network based services should be implemented with asynchronous messaging. Currently J2EE based application servers support the client server paradigm for service development. This paradigm is not sufficient for implementation of network services.

Problem 1: How to achieve concurrency between state machines and asynchronous messaging between state machines?

ActorFrame supports the "actor play roles" metaphor. A role is itself an actor, which recursively again can contain roles. UML2.0 also supports an aggregation concept called part, which can contain other parts. J2EE does not support these aggregation concepts.

Problem 2: How to map aggregation of actors?

Persistency of data is an import aspect of commercial services. ActorFrame does not provide any special support for persistency of data for state machines. J2EE supports persistent storage of data.

Problem 3: How can an implementation of state machines take advantage of the support of persistency in the J2EE platform?

When services are specified the intention is to use a UML profile that supports ActorFrame concepts to make ActorFrame models. There is ongoing work in the Java community for definition of a UML profile for Enterprise Java Beans (EJB), which is part of the J2EE technology. ServiceFrame has already been used for development of services. These services are described as ActorFrame models. These ActorFrame models should also be used when new implementation platforms are selected.

Problem 4: Can existing UML profiles for EJB be used for making ActorFrame models that survive changes in the implementation platforms?

It is also important that other concepts in ActorFrame can be mapped to the J2EE platform. It is also an important aspect of this study to identify additional benefits that may be gained by introducing ActorFrame to J2EE platform.

1.7 The thesis work

The approach taken in this work is to first study the different technologies, UML2.0, ActorFrame and J2EE. The study is then used as a basis for a discussion on possible solutions for mapping ActorFrame to J2EE. An existing UML profile for EJB will also be evaluated to find out if this profile could be used for ActorFrame modelling. Then different mapping solutions will be elaborated and one concrete mapping solution will be proposed.

To verify the mapping solutions for the problem statements 1, 2 and 3 a prototype version of a framework for implementation of ActorFrame models will be specified and implemented. The problem statement 1 and 3 will be verified by using the framework to make an example of a service. The service will be run on a J2EE application server. The result of the verification will be discussed, summarized and used to make conclusions.

1.8 Reader's guide

The J2EE platform constitutes of a large set of complex technologies. The specification of the EJB standard is alone about 600 pages. The UML standard is comparable in size. Therefore this report will focus only on those parts that are important for understanding the content of this report. The reader is encouraged to use the referenced literature if some of the technologies are new or the descriptions of the technologies are too brief or incomplete.

Chapter 2, 3 and 4 present the core technologies UML2.0, ActorFrame and J2EE that are used in this report. If the reader is familiar with these technologies, these chapters can be skipped.

Chapter 5 gives an introduction to the modeling and mapping approach taken in this report and it discusses the approach in relation to the Model Driven Architecture (MDA) promoted by OMG.

In chapter 6 different mapping solutions of the core concepts of ActorFrame to the J2EE platform are discussed, while in chapter 7 the proposed framework for implementation of ActorFrame models, is presented. Chapter 8 gives an example of a service and how this service is implemented using the framework.

In chapter 9 the solution is discussed in relation to the problem statements described in chapter 1, and this discussion is summarized in a conclusion at the end of the chapter.

Chapter 2 UML2.0

2.1 Introduction

OMG is now at the near end of standardization of the version of UML called UML2.0. The most important proposals to UML2.0, with respect to this report, have come from a consortium, named U2-partners [19]. Ericsson is one of the partners and researchers from NorARC have actively proposed new improvements to UML that are based on earlier work from standardization of SDL. A nearly finished version of the proposal from the U2 consortium was released in January 2003 and a final release to OMG RTF is planned for June 2003.

This proposal defines *"new user level constructs that will improve UML support for component-based development, architectural specifications, and advanced behavioral modeling techniques using interactions, state machines and activity diagrams"*. It contains many concepts, as it will be seen in chapter 3 are according to the concepts provided by JavaFrame and ActorFrame.

This chapter briefly describes the major improvements in UML2.0. Those parts of the proposal that are important to model ActorFrame based applications are described in more detail. Focus is on those parts of UML2.0 that are important for modeling active classes with complex behavior using state machines.

The release ad/2003-01-02 [18] from January 2003 is used in this report. Figures in this chapter are copied from the standard.

2.2 Major improvements in UML2.0

The major improvements in UML2.0 are:

- New concepts for describing the internal architectural structure of Classes, Components and Collaborations by means of Part, Connector and Port.
- Introduction of inheritance of behavior 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 the specification, realization and "wiring" aspects of the components.
- Integration 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 and an improved overview with Interaction Overview Diagrams.

As compared with earlier versions, UML2.0 seems to have matured into a more complete language, with improved integration of the various parts.

2.3 Architectural concepts in UML

Perhaps the most important improvement in UML2.0 with respect to modeling complex systems is the increased architectural support. Therefore a class may describe its behavior as a collaboration of behavior of instances of other classes contained in the class. The core concepts for describing this internal structure are Part, Connector and Port. These concepts are described in the sub chapters below.

2.3.1 Parts

The introduction of the Part concept in UML2.0 makes it possible to describe the internal structure of a class. A Part is a property of the containing class meaning that the Part lives and dies as part of the lifetime of an object of the containing class. Parts are set of instances of other classes.

The composition association in UML can also be used to describe composition as illustrated in Figure 2-1. However, this model does not express completely what should be described. The model describes a *Car* that consists of *Axle*, *Wheel* and an *Engine*. Each *Axle* is connected to at least two and at most four *Wheels* (may be three Wheels). Each *Axle* is also connected to an Engine. The *Boat* consists of an *Engine* and a *Propeller*. From this model it is possible that the same instance of *Engine* is connected to an *Axle* in a *Car* and to *Propeller* in a boat at the same time.

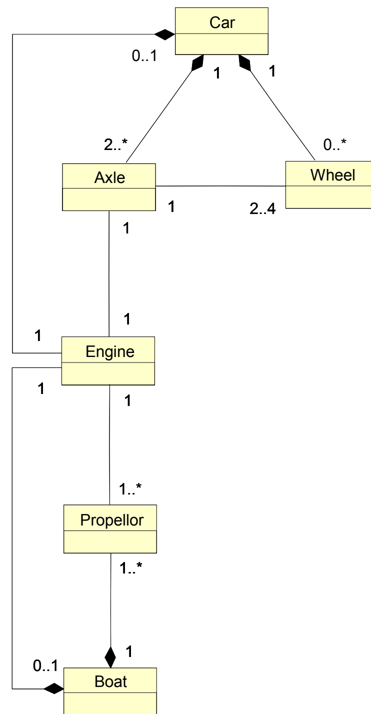

Figure 2-1 Composition versus parts [18]

That is obviously not what the model should express. The model should describe the class *Engine* independent of its use (encapsulation) and describe more precisely that an instance of class *Engine* is part of class *Car* and is connected to an *Axle* of that *Car*. Another instance of *Engine* is part of *Boat* and connected to a *Propeller* of that *Boat*. This is the main motivation for introducing the Part and Connector concepts.

In Figure 2-2 the class *Car* has got an internal structure of instances of other classes (parts). These internal parts and connectors will only exist as a part of an instance of the class Car. So when an Car is created, *e:Engine* is connected to *drive:Axle* that has two or more instances of *w:Wheel*. The car has also one or more sets of *run:Axle* where each instance has exactly one pair of *nw:Wheel* connected. The figure also describes that the same instance of class *Engine* cannot be part of both a car and a boat, which is possible according to the model shown in Figure 2-1.

Parts may have multiplicities in the format [n..m], where the n specifies how many instances are created when an instances of the containing class is created. The upper level m specifies the maximum instances that can be created.

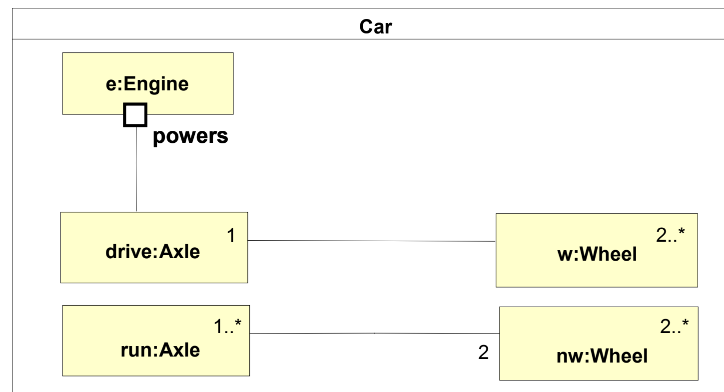

Figure 2-2 Class with internal structure [18]

2.3.2 Ports

When instances shall be connected together, the connection point should be described formally. A Port describes an interaction point for a class as described in Figure 2-3. *Port* is addressable, which means that signals can be sent to it. A *Port* may have a provided interface that specifies operations and signals offered by the class and a required interface that describes operations the class expects from its environment. In the figure below has the port *p* has a required interface named *power* and a provided interface named *powertrain*.

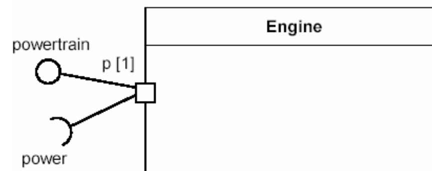

Figure 2-3 Ports connected to classes [18]

Use of ports enables specification of a class without knowing anything about the environment where the class may be used. Classes can send and receive signals via ports, and a class can expose operations through a port.

A port has an attribute *isBehavior* that specifies whether signal requests arriving at this port are received by the state machine of the object, rather than by any parts that this object contains. Such ports are referred to as behavior ports. The state machine of the class will handle signals that are sent to the behavior port.

2.3.3 Connectors

A Connector specifies a link (an instance of an association) that enables communication between two or more parts. In contrast to associations, which specify links between instances of the associated classifiers, connectors specify links between parts only. A Connector may be attached to a port or directly to a part as described in Figure 2-4. For example, an engine *e:Engine* in class *Car* is connected by the *axle* connector to the instances in the set *rear:Wheel*.

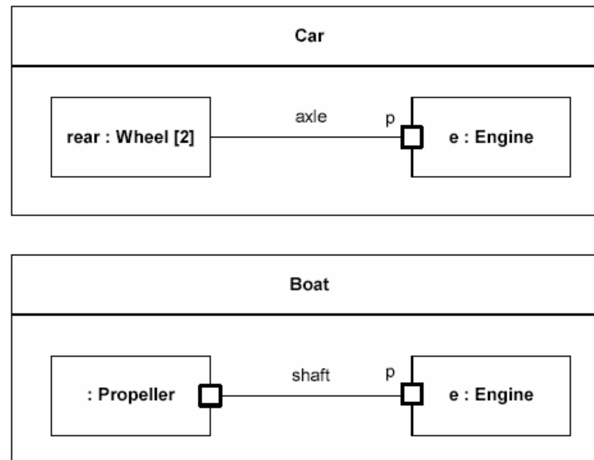

Figure 2-4 Connectors and ports [18]

The figure also shows how connectors are used to connect instances of a class to instances of different classes through ports. In the class Car *rear:Wheel* is connected to the port *P* of *e:Engine* and in the class *Boat* the *:Propeller* is also connected to the port *P* of *e:Engine*. Although the part *e:Engine* has the same instance name in the two classes *Car* and *Boat* they are different instances where each of them belong to their containing class. Connectors and ports are excellent concepts that enable more effective reuse of types and thereby encourage a component-based approach.

2.4 State machines in UML2.0

The major changes of the state machine concept in UML2.0 compared to UML1.4 are:

- Composite state with entry/exit points that increase the scalability and independence of behavior specification.
- State machine generalization that enables inheritance and specialization of behavior.
- Protocol state machines that enable allowed sequences of signals and operation calls to be specified.
- State machines that have operations that enable calls on state machines.
- State groups that enable common behavior of events in different states.

A state machine is used to model discrete behavior triggered by events. In addition to expressing the behavior of a part of the system, state machines can also be used to express the protocol through a port. These two kinds of state machines are referred to as behavioral state machines and protocol state machines. The state machine formalism used in UML2.0 is an object-oriented variant of Harel statecharts [39].

A state machine is triggered by different kind of triggers: signals, timeouts, operation calls and change in values. A trigger causes a transition if the trigger is specified for the current state for that state machine. A transition is described by actions that may cause new trigger events to be generated.

The most important improvements of the state machine concept in UML2.0 are encapsulation of behavior in a composite state and inheritance of behavior in sub types. These are described in more detail in the following sub chapters.

2.4.1 Composite state

UML1.4 has no limitations on how to enter and exit composite states. It is legal to enter directly to internal states of composite states and it is therefore difficult to specify composite states that can be reused in other state machines. UML2.0 has introduced exit and entry points that control access to a composite state. Figure 2-5 shows an example of a state machine with entry and exit points. Exit and entry points are named points that are placed on the frame of the state machine.

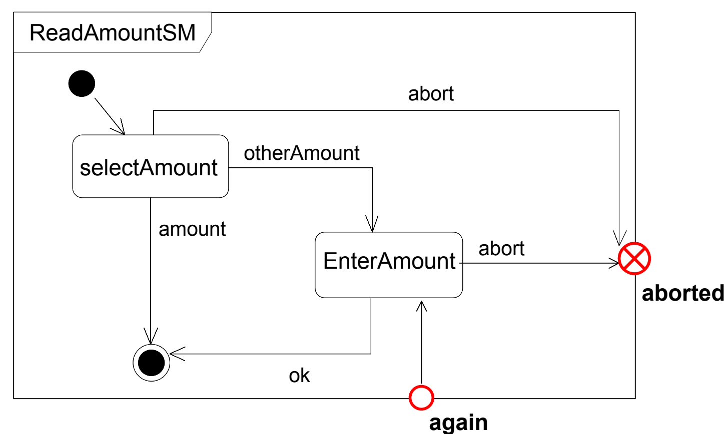

Figure 2-5 Definition of Exit / Entry points [18]

Figure 2-6 shows an example on how the ATM state machine uses the state machine *ReadAmountSM*. The transition *rejectTransition* ends in the entry point *again* of the state *ReadAmount*. If we look at Figure 2-5 again, we can see that transitions through this entry point ends in the state *EnterAmount.* The same semantic applies on transitions from states inside the composite state. The transition triggered by the signal *abort* exits the composite state through the exit point *aborted* and ends in the state *ReleaseCard*.

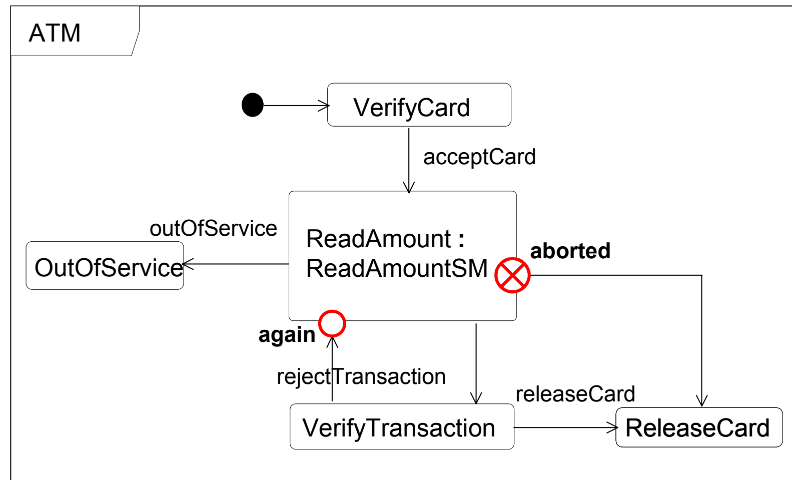

Figure 2-6 Use of Exit / Entry points [18]

Use of exit and entry points enables the design of components of behavior that can be more easily used in different places. This is analogous to use of procedures to obtain reusable operations.

An unnamed entry or exit point represents default behavior. A composite state may have several named entry and exit points.

2.4.2 Generalization of behavior

The generalization and specialization concepts have been an important part of the UML language. It has not been possible until now to inherit the behavior of state machines. As shown in Figure 2-7 this has now been added to UML in the same way as ordinary inheritance of classes. New state machine types can also be specified using inheritance independent of classes. New behavior specification can be added and parts of existing behavior can be replaced as follows:

- States and transitions can be added
- States can be extended
- Transitions can be replaced or extended
- Targets of transitions can be replaced
- Submachine states can be replaced.
- Signals and operations may be added.

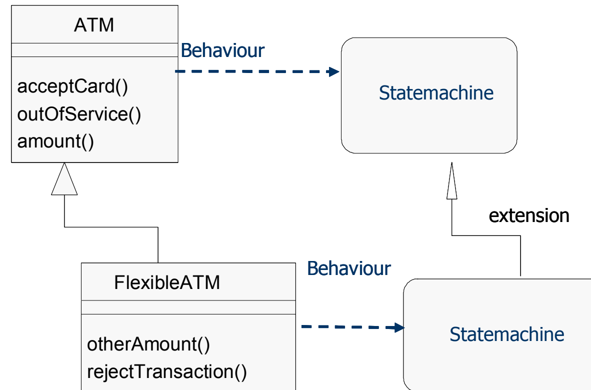

Figure 2-7 Specialization by extension [18]

Figure 2-8 shows how the composite state *ReadAmount* may be extended adding two new states (*SelectAmount*, *EnterAmount*), three transitions (*OtherAmount*, *ok rejectTransaction*), and one port. The figure also shows that it is allowed in UML2.0 to enter a composite state without going through an entry point. The transition *rejectTransaction* in state *VerifyTransaction* ends directly in the state *EnterAmount*.

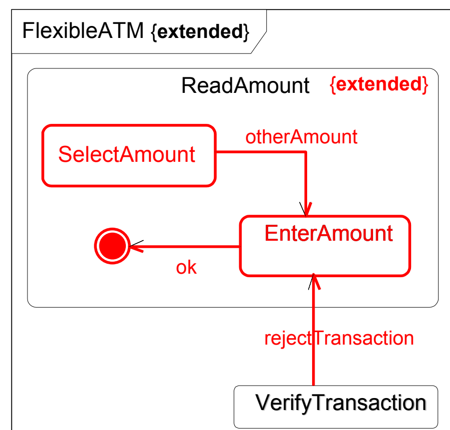

Figure 2-8 Specialization of state machines [18]

2.5 UML profiles

UML defines three standard extension mechanisms: stereotypes, constraints and tagged values. They are used to define extended metaclasses in packages that are called UML profiles. Profiles can be used to for instance, to tailor a UML metamodel to a specific platform such as EJB or .NET. It can also be used to make new concepts to be used in models. An UML profile for animals will for instance contain a metaclass *Cat* as shown in Figure 2-9. The class *Cat* is tagged with the keyword stereotype. *Cat* is an extension of the metaclass *Class* and *Cat* can therefore only be applied on instances of the metaclass *Class*. The stereotype can extend an existing metaclass or a stereotype.

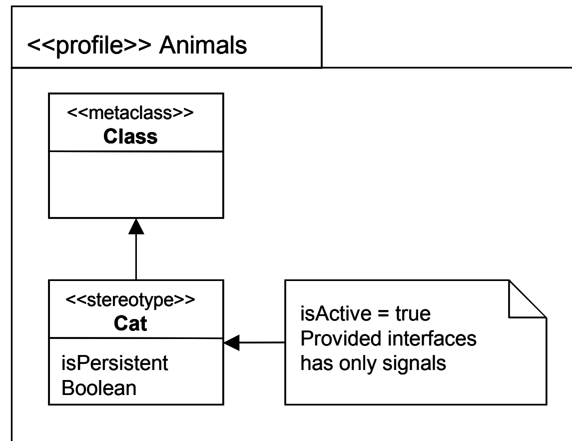

Figure 2-9 Defining UML extensions

A stereotype can have constraints. Constraints are expressed either by using OCL [40] or as informal text. In the example above there are two constraints defined. The stereotype *cat* must have the attribute *isActive* equal true and its interfaces can only contain signals. Stereotypes are defined in a UML Package tagged with <<profile>>.

A stereotype is used to tag instances of the metaclass as shown in

Figure 2-10. Attributes can be given values when the stereotype is applied.

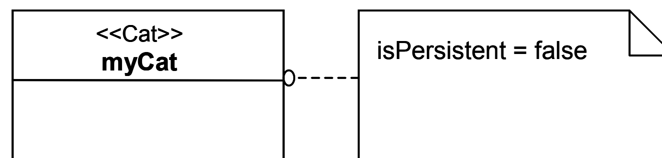

Figure 2-10 Use of stereotypes

2.6 Summary

This chapter has presented the major features in UML2.0 as it is specified in the proposal from the U2 consortium. It has shown that UML2.0 has been improved with respect to modeling of behavior and composition. Behavior can now be extended, which is important when more complex behavior is modeled. It is also possible to make fully encapsulated behavior components, state machines, which may enable reuse of behavior. It seems that UML2.0 now has concepts that better support modeling of systems that have to cope with conflicting initiatives and communication between loosely coupled components.

UML also has an extension concept called profiles. Profiles are used to define metaclasses in UML to tailor a UML model to a specific platform such as EJB.

In the following chapter ServiceFrame is presented, which is built on concepts that are very much inline with UML2.0.

Chapter 3 Ericsson's service creation architectures

3.1 ServiceFrame

ServiceFrame [1] provides architectural support for service creation and service execution. Services are realized by ServiceFrame applications that are defined by specializing and instantiating framework classes. The idea is that service developers shall be able to concentrate on modeling the service functionality and be relieved from considering technicalities that are not service specific.

ServiceFrame is intended to be packaged with method guidelines and tools to form a model driven service development kit (MDK) see Figure 3-1.

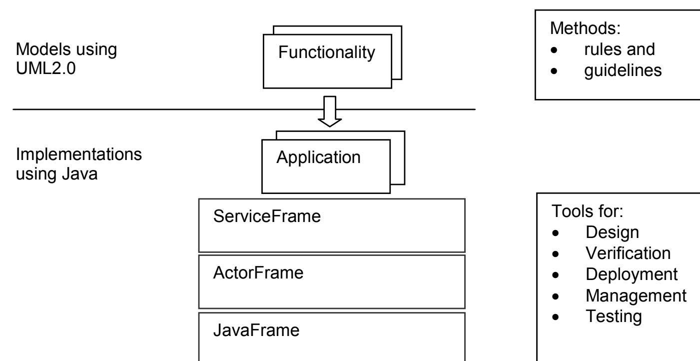

Figure 3-1 ServiceFrame - a model driven service development kit [1]

ServiceFrame provides architectural support for modeling and for implementation in terms of domain concepts. In addition it has mechanisms that support incremental development and deployment of services.

The architectural support is provided in three layers, as illustrated in Figure 3-2. ServiceFrame itself is an application of ActorFrame, which is a generic application framework supporting the concepts of actors and roles that are described in next chapter. Both are implemented in Java using JavaFrame [2], which provide support for state machines and asynchronous communication according to UML2.0 running on a Java Virtual Machine.

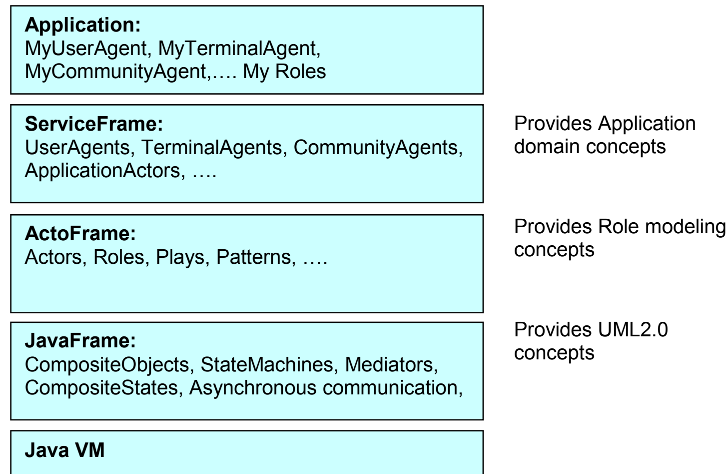

Figure 3-2 ServiceFrame layers [1]

A description of the ServiceFrame can be found in [1] and [3].

3.2 ActorFrame

In general a service is consider as an identified functionality serving a stated need or purpose. Since a system normally will provide many services, a service may be seen as a partial functionality of the system.

In general, services entail collaboration among objects that provide some specific functionality that form the part of the provided services. ActorFrame uses the well-known metaphor that "*actors play roles*" [22], [23]. Actors are objects that play different roles. Hence, a service may be defined in terms of collaborating service roles where a service role is the part an actor plays in a service [24]. Models that use the ActorFrame concepts are called ActorFrame models.

3.2.1 Actor

Actor is the core concept of ActorFrame. An actor, illustrated in Figure 3-3, is an object having a state machine and an optional inner structure of actors. Some of these inner actors are static, having the same lifetime as the enclosing actor, and others are dynamically created and deleted during the lifetime of the enclosing actor. The state machine of an actor will behave according to the generic actor behaviour that is common to all actors. If the actor shall play several roles, this is accomplished by creating several inner actors each playing one of the desired roles.

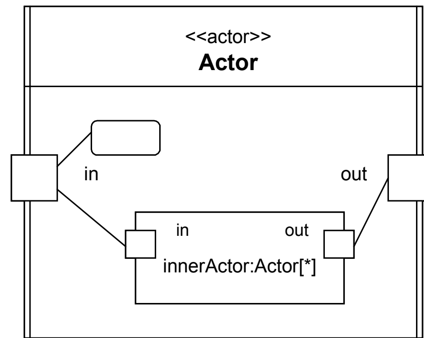

Figure 3-3 Class Actor

Communication between an actor and its environment takes place via an *Inport* and an *Outport*. Internal communication among the inner actors is also routed via these ports.

An actor has a generic behavior, inherited from the base class Actor, that provides management functionality. It manages the inner structure of actors and the roles they play. It knows the available roles and the rules for role invocation.

The generic behavior handles role requests as described in Figure 3-4. It will either deny the request or invoke an actor to play the requested role or an acceptable alternative role. The generic behavior also has the capability to add and remove roles, and to perform other actor management functions.

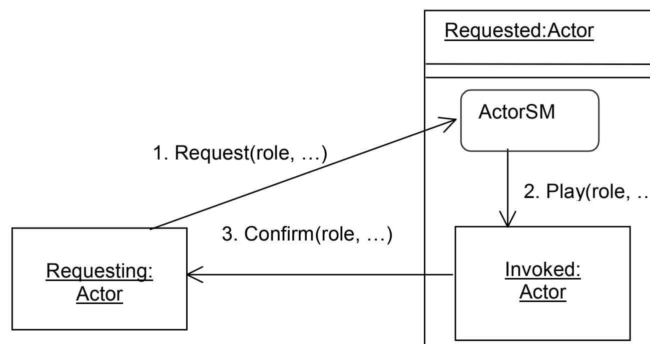

Figure 3-4 RoleRequest protocol [1]

An actor has an actor address that consists of an actor name and an actor type. The name has to be unique among the actor instances that are in the name scope of a requested actor, ref Figure 3-4.

3.2.2 ActorFrame protocol

ActorFrame has protocols for role requests and role releases. New roles can be created dynamically and initiated on requests. The idea is that an actor can request another actor to initiate new roles (actors) to do a requested service.

As shown in Figure 3-5 an actor may request several other actors and several other actors may request one actor. All actors are running in parallel. An actor may play

several roles in parallel. If a requested role is released from all requestors, the requested actor will delete the role. If a requested actor or role is defined but it does not exist, it will be created, if it is allowed to be involved

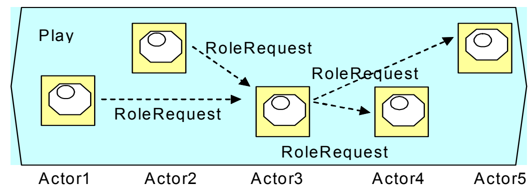

Figure 3-5 Multiple roles and actors

The basic feature of the protocol is to allow an actor (requestor) to request another actor to play a specific role and to allow the actors to interact to perform a service or a play. The protocol includes also a protocol to release a requested role. Figure 3-6 shows a typical pattern of how *RoleRequest* and *RoleRelease* are used to invoke other actors to play services. One *RoleRequest* may lead to another *RoleRequest* as shown in the figure below. Release of roles may lead to deleting of actors if they play no more roles. It is also possible to define that an actor may exist although it does not play any roles.

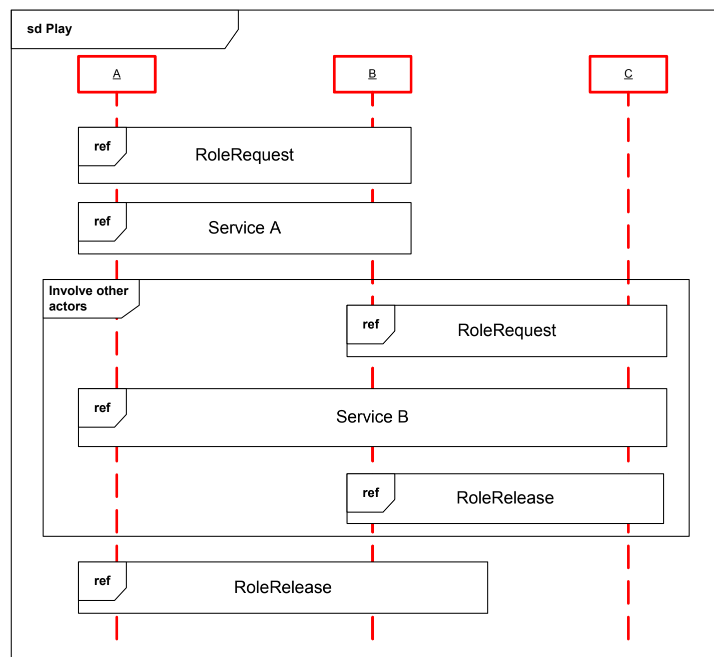

Figure 3-6 A simple service.

The term play is used for a dynamic structure of interacting actors performing according to role types, see Figure 3-7. A play may perform a single service or a combination of services. Exactly which services a play will perform is normally not determined at creation time, but the involved actors and roles determine it dynamically.

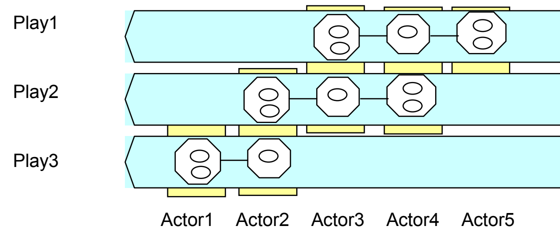

Figure 3-7 Play

A play has a lifetime that begins when its first role is created and ends when its last role is ended. The difference between plays lies in the role types that are active and the features that are selected in each. Different actors play the roles in a play concurrently. All roles in a play may be deleted by one of the participating actors.

3.3 JavaFrame

JavaFrame is a *Modeling Development Kit* for development and execution of state machines in Java. It provides a layer between the Java language and concepts used in state machine modeling. According to Haugen [25] "W*ith JavaFrame it is possible to apply modeling techniques and still work in Java. The Java source and the model have one-to-one relationship. The framework provides classes of well-proven modeling concepts, and by using these, instead of just programming in plain Java, the abstraction level is raised".*

The concepts of JavaFrame are, as will be shown later, closely corresponding to concepts defined in UML2.0. JavaFrame is both a library of classes to be used for implementation of state machines and a set of basic mechanisms for execution of state machines in terms of classes. Applications on top of JavaFrame typically define subclasses of the provided classes and create objects from these.

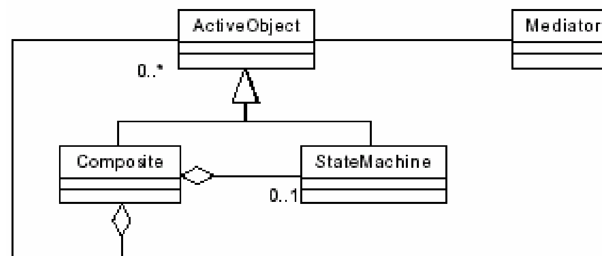

Figure 3-8 JavaFrame classes [2]

The main classes in JavaFrame that are used by the designers are *StateMachine*, *Composite* and *Mediator* as shown in Figure 3-8 and these are described in the following subchapters.

3.3.1 Composite

Composite and *StateMachine* are active objects meaning that they have their own behavior and are run in their own thread of control. In UML2.0 this is the same as

setting the *isActive* property of UML classes to true. An active object communicates by sending or receiving messages through *Mediators*.

A *Composite* may contain other active objects, which means it may contain other *Composites*. A Composite may have one *StateMachine* that represents the behavior of the composite itself. Classes in UML2.0 may also have an own state machine in addition to all parts they may contain. The actor concept described in chapter 3.2.1 always has an own state machine.

Specialization may be applied to classes of *ActiveObject*. The structure of the superclass including mediators is inherited.

3.3.2 StateMachine

StateMachine in JavaFrame is a subset of the state machine concept in UML2.0. It does not provide support for regions, which are used to split the state machine into orthogonal behavior areas. JavaFrame supports the submachine concept in UML and it is called *CompositeState* in JavaFrame. This is not exactly the same as the state machine concept in UML that allows entering and leaving of composite states without going through entry and exit points. This is contrary to *CompositeState* in JavaFrame that imposes the use of entry and exit points.

StateMachine may contain both composite states and simple states as defined in UML2.0. As in UML state machines may be specialized by replacing transitions and adding states and composite states.

Triggering of transitions can only be done by receiving signals through mediators. It does not support triggering of transitions based on changed values of properties and invoking of a state machine via operations or methods calls.

Post and pre conditions are supported in *StateMachine* through entry and exit methods that are called from a super class at the beginning and end of transitions. These methods may be specialized.

As also defined in UML2.0 JavaFrame state machines support the history and deep history concept. These are very useful concepts that allow specification of transitions to a composite state that are common for all its sub states. A transition with a deep history ends in the original state again, while a transition with history enter the composite state through its default entry point.

State machines in JavaFrame also support the save concept from the SDL language [20]. *Save* allows received signals to be stored temporary in a save queue. When the state machine enters another state, signals from the save queue are resent to the state machine.

3.3.3 Mediators

A mediator is addressable object meaning that it may be referenced and sent signals to. A mediator is an object that belongs to a state machine and it enables an active control of the communication flow between state machines. Such a mediator provides an encapsulation of the state machine, which makes it easier to reuse a state machine in different places. Simple mediators are connected to each other via references to

another mediators. Signals sent to a mediator are forwarded to the referenced mediator and so on.

Several different types of mediators exist:

- ProtocolMediators that hide lower level protocols.
- MulticastMediator that multicasts the signals to a set of referenced mediators.
- RouterMediator that routes signals based on information carried by the signal, as for instance the signal name, or the specified sender or receiver of the signal.
- EdgeMediator that connects the edge of the outmost Composite with different protocols such as RMI, CORBA, HTPP that are used to exchange information with the environment.

A mediator is partly equal to ports in UML. It does however only support reception and sending of signals and not operation calls. It may also have a state machine. UML also supports a concept called complex port, which has a special kind of state machine used to control the protocol of signals.

3.3.4 Runtime systems

In JavaFrame each state machine has its own signal queue. The queues are managed by a scheduler, which controls the scheduling of state machines based on a Round-Robin scheduling principle. One scheduler is run in one Java thread, so all its state machines are run in quasi parallelism, within a single Java thread.

3.4 Summary

This chapter describes ServiceFrame as a framework for creation and execution of services especially for the service network. It is built on ActorFrame, which supports a role-based approach to modeling. Models that use these concepts are called ActorFrame models. ActorFrame is built on JavaFrame that consists of state machine classes that can be extended, and an execution environment for state machines. This chapter has also shown that there are many similarities between ActorFrame and UML2.0.

Chapter 4 J2EE – Java 2 Enterprise Edition

4.1 Introduction to the different J2EE technologies

Sun has, through the Java community, defined a collection of Java based technologies for server side application development and execution named the Java 2 Platform, Enterprise Edition (J2EE) [11]. The current official version is named J2EE version 1.3, but version 1.4 is soon to be announced. A beta version of J2EE1.4 is used in this report.

J2EE consists of APIs for each of the following technologies assuring that applications that are using these API's may be run on J2EE compliant platforms. The specification of these APIs can be found at [11]. The most interesting APIs for this thesis work are:

1. Enterprise JavaBeans (EJB), which define software architecture for a server side components called enterprise java beans.
2. Java Messaging Service (JMS), which provides asynchronous messaging services for point-to-point communication and broadcasting of messages.
3. Java Naming and Directory Interface (JNDI), which provides an API for accessing names and directories services. It is mostly used to lookup references to EJBs by using names.
4. Remote Method Invocation (RMI/IIOP) [4] that enables a Java application to invoke methods on objects that reside on different Java Virtual Machines (JVM). IIOP is the protocol used in the CORBA [37] standard and J2EE provides a common interface for both CORBA and EJB components securing interoperability between these technologies.
5. Java Transaction API (JTA) provides mechanism for handling commit and the rollback of transactions as well as ensuring ACID properties of a transaction. ACID is an abbreviation for the key features Atomicity, Consistency, Isolation and Durability.
6. Java API for XML-based RPC (JAX-RPC) that enable clients to use the SOAP standard and HTTP to make XML-based remote procedure calls (RPCs) over the Internet. JAX-RPC also supports WSDL and it is now integrated with EJB in such a way that EJB components can seamlessly interact with web-services.

J2EE defines other APIs such as Java Server Pages (JSP), Java servlets, XML Processing (JAXP), Java transaction Services (JTS), JavaMail, Java Database Connectivity and Interface Definition Language (IDL) [43]. The J2EE standard includes complete specifications and compliance tests to ensure portability of applications across the wide range of existing enterprise systems capable of supporting J2EE.

4.2 Enterprise Java Beans (EJB)

4.2.1 Introduction to EJB

The Enterprise Java Beans Specification [12] defines architecture for building distributed object-oriented applications in the Java programming language. EJB is a component technology that supports a distributed paradigm and opens for interaction with other important technologies such as web-services. It provides and hides many troublesome mechanisms for an application developer such as transaction control, persistency of data, thread control, and directory services. It supports a "Write Once, Run Anywhere" philosophy.

The current version of the EJB specification is 2.1, which is also used in this report. The latest improvements are the seamless integration with web services, a coarse-grained timer function and extending of the Java Message Interface to include message interaction with the JAX-RPC protocol.

Some of the most essential characteristics of an enterprise bean are listed in the EJB specification:

- An enterprise bean typically contains business logic that operates on the enterprise's data.
- An enterprise bean's instances are created and managed at runtime by a Container.
- An enterprise bean can be customized at deployment time by editing its environment entries (see 4.3.1).
- The Container mediates client access to the enterprise beans.

4.2.2 EJB Architecture

Enterprise beans are living inside an EJB container, which is hosted by a J2EE Application Server as described in Figure 4-1. A J2EE application may also contain other containers such as a web container than handle requests from web clients.

An EJB container is an abstract entity that manages the lifetime of various instances of enterprise bean classes. The container provides the set of interfaces defined in the J2EE specification: transaction services, persistency of data, network transparency through use of RMI/IIOP or SOAP, thread handling, and JDBC connections to databases.

An enterprise bean is specified through a set of interfaces and bean class. These are marked grey in Figure 4-1. An exception is the message bean that does not have any interfaces. During deployment of a bean the container generates classes for creation of a *HomeObject* and *EJBObjects*. There is one *HomeObject* for each enterprise bean type and it is a factory object for management of *EJBObjects*. An *EJBObject* is a proxy object for interaction with a client. There is one *EJBObject* for each client.

A client does not interact directly with the enterprise bean. It must access the enterprise beans through its interfaces. An enterprise bean component has a home interface and a remote interface. The latest specification EJB 2.1 specifies local

interfaces for beans that are residing in the same Java Virtual Machine (JVM). The differences between local and remote interfaces will be described later in this chapter.

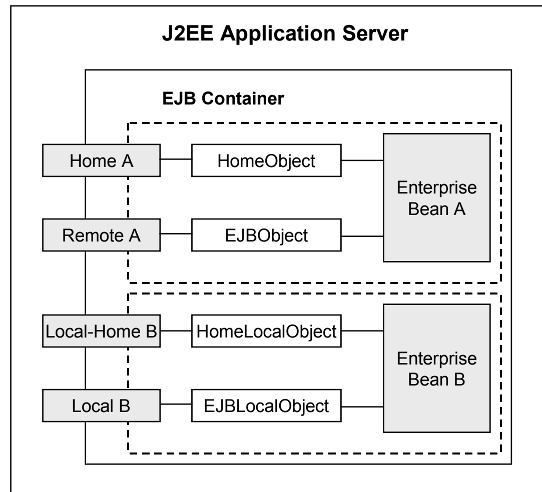

Figure 4-1 EJB Architecture

The *Home* interface specifies the methods for creating or finding beans, while the *Remote* interface contains business methods that are specific for the enterprise bean. The *Enterprise Bean* class implements the business methods defined in *Remote* interface and eventually those creation methods that are defined in the *Home* interface.

The container is responsible for making the home interfaces of its deployed enterprise beans available for clients through JNDI. Thus, the client can look up the home interface for a specific enterprise bean using JNDI.

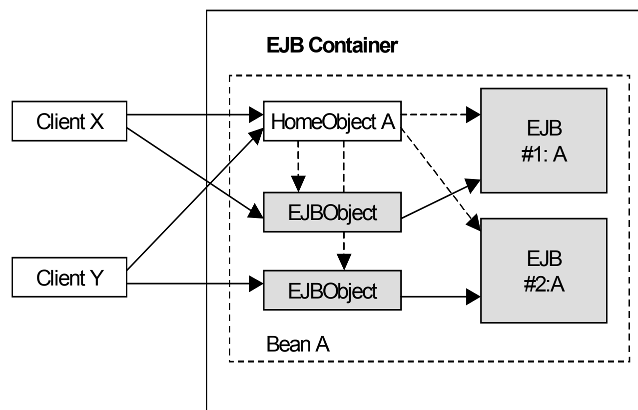

Figure 4-2 Client view of enterprise beans

AAAWhen a client accesses a specific *HomeObject*, the *HomeObject* will create an *EJBObject* instance and allocate or eventually create an *EnterpriseBean*. This is illustrated in Figure 4-2 where the creation of enterprise beans is illustrated with dotted lines. How this is implemented, is not specified by the standard. A unique

reference to the *EJBObject* is returned to the client. The client can then access the business methods through the *EJBObject*, which is responsible for forwarding the method call to the allocated enterprise bean.

As mentioned earlier, there are local interfaces for use by enterprise beans that reside in the same JVM. The reason for this is to make calls between enterprise beans much more efficient. When the *Local-Home* interface is accessed, the reference to *LocalObject* is returned. Calls to the *LocalObject* are then made by reference. This is obviously more efficient than use of the remote interface, where RMI/IIOP is used as the communication protocol. But this design solution also has an impact on the possibility to freely deploy enterprise beans to different containers. RMI/IIOP passes parameters as values, which has a different semantic than passing parameters as references used in the call by reference mechanism.

4.2.3 Enterprise beans

The enterprise bean class contains the implementation of the methods that are specified in interfaces. There is a specific name convention that has to be followed. Three different kinds of enterprise beans are defined:

- Session beans that are representing both stateless and stateful services for clients inclusive web-services.
- Entity beans that represent business data to be shared among multiple clients in a persistent storage medium.
- Message beans that represent stateless services invoked through messages received asynchronously.

Each of these will be further described in the following subchapters.

4.2.3.1 Session bean

A session bean represents a single client inside the J2EE Application server. It is created by the client by invoking a create method in the home interface and it exists only for the duration of a single session. The session bean normally represents some business logic.

Although session beans can be transactional, the container does not guarantee recovery after a system crash. Session beans must therefore manage their own persistency of data.

There are two types of session beans:

- Stateless beans where the conversational state is not kept between invocation of the beans methods.
- Stateful session beans where the container manages the conversational state of the session bean between the different method invocations.

A session bean does not have identification, but a globally unique reference can be obtained, which can be stored and later used for a re-invocation of the session bean.

A client accesses a session bean through its remote or local interfaces. A stateless session bean can also provide a web service endpoint, which can be used by web service clients.

The characteristics of session beans are:

- Session beans model some business logic.
- Session beans are normally used by a single client and they are normally relatively short-lived.
- Session beans can either be stateless or stateful, but they do not survive between sessions.
- A stateless session bean may also provide a web service endpoint.
- Session beans do not provide persistency of data.

4.2.3.2 Entity bean

Normally entity beans represent persistent data of an application. The enterprise bean interface has methods for accessing the data. An entity bean is identified with a unique primary key and the home interface must have a method that is used to find an entity bean based on the primary key. The data is normally stored in an underlying database and J2EE recommends that JDBC is used. Entity beans are often associated with database transactions and may handle concurrent access from multiple clients.

Entity beans are transactional and there are two different types of beans, which reflect how the beans handle the persistency of data. In container-managed persistence (CMP) the container guarantees persistency of data, while in bean-managed persistency (BMP) the bean is responsible for storing and restoring the data.

The specification recommends the use of CMP entity beans. The persistent data is specified by abstract set and get methods, which are used by the container to generate the necessary classes needed to obtain persistency service. The container also automatically generates methods for finding entity beans based on the method signature in the home interface.

Because most of the databases are SQL based, the EJB standard has provided a specification of an EJB SQL language that can be used to find a specific entity bean or a collection of entity beans.

The main characteristics of entity beans are:

- Entity beans represent business data that are persistent and are living as long as the data exist in the database.
- Entity beans can participate in transactions and the container supports the principles behind the acronym ACID.
- Entity beans are referenced by unique primary keys that are defined as Java classes.
- Entity beans handle concurrent access from multiple users.

- The container supports CMP beans with persistency of data. But the specification allows custom designed persistence control through use of BMP beans.

4.2.3.3 Message beans

Message-driven beans or, in short message beans, are asynchronous message consumers. The latest specification has extended the message beans to be able to receive messages from other sources in addition to JMS. A message bean is defined for a single message type, in accordance with the message listener interface it implements. This extension is motivated by the popularity of Web-services and it has opened the J2EE architecture to include communication with web services.

Message beans are listening on specific JMS destination or a web-service endpoints. A JMS destination is either a message queue or a topic. A web-service endpoint represents a service interface for a client that calls a web-service. These concepts are described in more detail in chapter 4.4 which describes the JMS system and chapter 4.6 which describes web-services. When a message is put into a JMS destination or a client calls a web-service endpoint, the container invokes the actual message bean. Which web-service endpoints or JMS destinations the message bean shall receive messages from are specified at deployment time.

Message beans do not have home and remote interfaces. They are anonymous and can therefore only be accessed via messages. The container may create several instances of a message bean type, which enables concurrent handling of messages. The container does not guarantee that messages are consumed by the message beans in the same order as they arrive into the destination or endpoint.

Message bean instances are like stateless session beans in the sense that they have no conversional state. When more than one message bean instance is deployed, they must serve all received messages in a equal way. Although message beans contain no state for specific clients, they may contain data forhandling of messages. tThis is for data that is common for all client messages such as references to other enterprise beans.

Message beans are transactional and integrated with the JMS transaction service. If a message bean uses container managed transaction, the message acknowledgement is handled automatically as part of a transaction commit. If bean managed transaction is used, the reception of messages cannot be part of a transaction. The container is then responsible for acknowledgement of messages.

It is only message beans that are allowed to implement the *MessageListner* interface. Other bean types can use JMS, but without the support of the container.

The main characteristics of message beans are:

- Message beans are asynchronous message consumers receiving messages from destinations or endpoints.
- Message beans have neither home nor remote interfaces and are therefore anonymous for clients.

- Message beans are stateless beans, but they may contain information such as references to other enterprise beans.
- The destination or service endpoint is specified at deployment time.

4.3 Enterprise bean environment

To be able to reuse an EJB component, the EJB component needs to be customized to fit into different environments that exist for the application. The EJB specification specifies an environment mechanism to allow customization of the enterprise bean's business logic, use of external resources or references to other components without accessing the enterprise bean's source code. All environment variables are defined in deployment descriptors, which are XML files. The container is responsible at deployment to read the deployment descriptors and put the environment variables into JNDI. The environment variables are available to the bean at runtime through the JNDI interfaces.

Several types of environment variables are defined and targeted for different purposes. The most interesting ones in this study are described in the following subchapters.

4.3.1 Environment entries

The enterprise bean's environment entries allow the enterprise bean to be customized without the need to access or change the enterprise bean's source code.

Each enterprise bean defines its own set of environment entries. All instances of an enterprise bean share the same environment entries. The environment entries are not accessible to other enterprise beans. Enterprise beans are not allowed to modify the bean's environment at runtime.

At deployment time all environment entries that are accessed in the beans source code have to be declared. An environment entry is declared using *env-entry* elements in the deployment descriptor for the bean. An *env-entry* element consists of an optional description of the environment entry, the environment entry name, the expected Java type of the environment entry value, and an optional environment entry value. The environment entry value must be of type String, Character, Integer, Boolean, Double, Byte, Short, Long and Float.

4.3.2 EJB references

EJB references are used to refer to other enterprise beans by "logical" names. The EJB references are special entries in the enterprise bean's environment. The EJB references are bound to the different enterprise beans that are deployed in the target runtime environment. EJB references are declared using *ejb-ref* and *ejb-local*-ref elements of the deployment descriptor. The difference between these two elements is that *ejb-ref* refers to the remote interface of an enterprise bean, while *ejb-local-ref* refers to local interfaces. All referenced elements are available through JNDI at runtime.

EJB references describe the home/local-home and remote/local interfaces of the referenced enterprise. The *ejb-ref* element consists of elements for an optional

description name of the bean, the type of the referenced bean, and the expected Java types of the referenced enterprise bean's home and remote interfaces. EJB references can only refer to session and entity beans.

4.3.3 Web service references

Service references are references to external web-services using "logical" names. The web service references are specials entries of the environment of a bean and are used to bind logical names to web services endpoints in the target operational environment. A web service interface is of remote procedure call (RPC) type as defined in JAX-RPC.

A *service-ref* element describes the interface requirements for the referenced web service that an enterprise bean needs for accessing the web service. It contains an optional description, a mandatory name of the web service, and an interface name that qualifies the JAX-RPC service.

4.3.4 Message destination references

Message destination references are references to JMS destinations that allow an enterprise bean to send messages to other objects through use of "logical" names. The are destinations are looked up in JNDI by using the name of the message destination reference. The deployment element *message-destination* contains an optional description element, a mandatory name to be used in the enterprise bean's code, a specification of the type of destination (queue or topic), and an element that defines if the destination is used to consume or produce messages.

4.3.5 Deployment descriptors

The environment variables described above (and others) are declared in XML deployment descriptor files. The deployment descriptor files are read at deployment time by the container where most of the information is put into JNDI enabling the enterprise beans to read the different variables. The container uses part of the environment information in the deployment descriptors to configure different properties of the enterprise beans. For instance, the container uses the specification of JMS destinations when it creates message beans.

The deployment descriptor file consists of two types of information:

- Enterprise structural information that describes the structure of an enterprise bean and declares the enterprise beans external dependencies such as specification of home and remote interfaces. This information is mandatory and cannot be changed later in the deployment process.
- Application assembly information that describes how the enterprise beans may be composed into larger parts or applications such as environment entries. Provision of assembly information is optional.

4.4 Java Messaging System

4.4.1 Message Oriented Middleware

Message Oriented Middleware (MOM) provides messaging facilities for distributed components to be able to asynchronously communicate with each other. A component, which is a message client, can send and receive messages to any other component. Although MOM is used to enable distributed communication between components that are loosely coupled, asynchronous message passing is also fundamental for the actor paradigm described in chapter 3.2. It supports active components with own behavior, like an actor in ActorFrame, to communicate asynchronously with other actors. For example, MOM enables sending messages to several components at the same time requesting some information. This reduces the response time to arrival of the latest received message.

There has been MOM based products available for some time, but they lacked a common specification, which caused problems with interoperability between different products. Java Message Service (JMS) is Java based MOM middleware that is now part of the J2EE specification and J2EE servers must support it.

In MOM a component sends a message to a destination and the receiving component can retrieve the message from the destination. The sender and the receiver do not need to know about each other's methods. They only have to agree on the format of the message and which destination to use. In this respect, messaging differs from tightly coupled technologies, such as RMI, which require a component to know the remote methods of other components.

4.4.2 Java Message Service

The Java Message Service (JMS) is a Java API that supports MOM functionality. It is a common specification that enables JMS clients to communicate with other messaging implementations and which provides interoperability between the different MOM products.

The JMS API supports an asynchronous and reliable communication between loosely coupled JMS clients. JMS supports both point-to-point messaging and publish/subscribe messaging. The main differences between these concepts are:

- In point-to-point messaging there is only one receiver of each message, while in publish/subscribe messaging, the same message is received by all receivers that have subscribed to the topic.
- In point-to-point messaging a message is stored in a queue until it is delivered to a receiver, while in subscribe/publish the message sent to a topic is immediately sent to all subscribers of the topic. There is no queuing of the messages.

Although in messaging there is timing dependency between sending and receiving of messaging, JMS supports two different modes of message reception.

- Synchronously, in the sense that the receiver explicit fetches the message from the queue or topic by calling a receive method. It may be blocked until a message is received or it may time out.
- Asynchronously, in the sense that a receiver may register a listener interface. A message is received when the JMS calls the listener's *onMessage* method at the time the message arrives at the queue or topic.

A message consists of a header, properties and a body. Properties may carry user specified properties. A receiver can use these properties to filter the messages it receives. JMS support different types of messages, which differ in the format of the message body. The different types are listed in Table 1.

Table 1 JMS message types [10]

Message Type	Body Contains
`TextMessage`	A `java.lang.String` object (for example, the contents of an Extensible Markup Language file).
`MapMessage`	A set of name/value pairs, with names as `String` objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
`BytesMessage`	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
`StreamMessage`	A stream of primitive values in the Java programming language, filled and read sequentially.
`ObjectMessage`	A `Serializable` object in the Java programming language.
`Message`	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

Queues and topics are what J2EE calls administered objects. Administrated objects are bound by administrators of the application servers. Administered objects are configured in the JNDI namespace, and JMS clients can look them up by using the JNDI API.

4.4.3 Integration of JMS into J2EE

The JMS API has the following features:

- Application clients, Enterprise bean components, and Web components can send or synchronously receive a JMS message. Application clients can in addition receive JMS messages asynchronously.
- Message-driven beans use asynchronous consumption of messages.
- Includes messaging within distributed transactions.
- Concurrent consumption of messages by allowing multiple message bean instances.

4.5 Java Naming and Directory Interface

The Java Naming and Directory Interface (JNDI provides an interface for accessing name and directory services such as LDAP directory services and Domain Name Service (DNS). JNDI enables Java programs to use name servers and directory servers to look up Java objects by name. This feature enables an application to locate distributed objects, which is essential in distributed programming. JNDI is a generic API that works with any name or directory server and as such it provides a common interface against existing directory and naming servers.

A directory service typically provides access to data structured in hierarchies, such as directories in a file system. It is also used to categorize data into hierarchies such as yellow pages. A naming service allows access to objects by name for instance, looking up an IP-address to a computer based on a name as in DNS. In the thesis work the naming service is the most interesting.

4.5.1 Naming service

A naming service provides a method for mapping unique identifiers or names to a specific value or a reference to an object. It is important that the name is unique in the actual namespace. Compound names are names that consist of a sequence of names that conform to the naming convention. For instance, "melby/ola" is a different name than "husa/ola" although the person has the same name "ola". This convention is often used for scaling the number of unique names in a namespace. It is also possible to define composite names that consist of several namespaces. An URL consists usually of three parts: protocol name, name of server, and name of the file to be accessed.

A name must be bound to a value that may be an object, a Java class or a reference to an object or class. Remote Method Invocation (RMI) is an example of a distributed object management system that provides a name service for mapping names to objects residing on different computers. The object may be looked up by using the name of the object and gets in return the object or a reference to the object.

A context is the initial starting point when searching for objects. JNDI provides a listing facility that returns a list of items in the defined context. A lookup facility is used to look up a single object based on an object name in the context.

4.5.2 Integration of JNDI and J2EE

JNDI is a core service in J2EE. It is used to locate Enterprise beans and to let Enterprise beans get access to different environment variables. Other components in the J2EE framework are also available through JNDI as for instance JMS destinations. An EJB container is responsible for putting names specified in the deployment descriptor into JNDI.

4.6 Web-services

4.6.1 Introduction

Web services are applications that are defined, published and accessed across the Web. It consists of a set of technologies enabling loosely coupled applications to expose

service interfaces (WSDL), a protocol that enables communication between applications (SOAP) and a registry (UDDI) for publishing the services. Together these technologies provide a Service Oriented Architecture (SOA). Web services are an answer to the need to integrate business applications, enabling exchange of business information in real time. To that purpose an open and implementation independent standard is needed, which emphasizes services rather than interconnection of computers. Although Java RMI and CORBA are middleware for connecting distributed applications together, they are limited in integrating a wide variety of systems that exist over a rather unstable and firewall protected Internet. The SOAP protocol can run on different communication protocols such as HTTP, and it supports asynchronous communication of messages. Current web-service standards support only synchronized communication between service endpoints. The proposal for SOAP1.2 [47] defines an asynchronous communication style between service endpoints. This enables networks to route service requests as SOAP messages to the correct service endpoint like computers are addressed on the Internet today.

4.6.2 Web service technology stack

Web services are based on the three main technologies SOAP, WSDL and UDDI. In addition there is XML [50], which is a text-based language used to describe data, a core technology for describing service interfaces and the content of SOAP messages.

SOAP or Simple Object Access Protocol supports the encoding of arbitrary data, usually described in XML, and the transfer of data over a channel between communication endpoints. SOAP is wire protocol neutral and therefore can be used over different protocols like for instance HTTP, FTP and SMTP. But because of the ubiquity of the HTTP protocol, most implementations today use HTTP. SOAP is a lightweight mechanism for exchange of data between heterogeneous applications that is independent of operating systems, programming languages and network protocols. SOAP supports a language independent Remote Procedure Call (RPC) protocol mechanism. Specification of different encoding rules enables exchange of application-specific data types. SOAP supports also definition of schemas for data types. This makes SOAP a good protocol for integrating of heterogeneous applications.

WSDL or Web Service Description Language is an XML based language that describes the functionality of Web services. It has similarities with the Interface Description Language IDL used for instance to define CORBA interfaces. WSDL describes the operations a Web service offers including the parameters of each operation and the return value. But WDSL does not describe the semantics of the operations. WSDL enables method calls to a web-service regardless of the selected RPC mechanism.

UDDI or Universal Description Discovery and Integration provides global directory services for Web services. It supports also a protocol for publishing and discovering of services. The Web services are described with WSDL and are stored in the UDDI as WSDL files. The WSDL files are retrieved from the UDDI by an application developer making it possible to make calls to Web services.

4.6.3 Web services in J2EE

In the recent J2EE specification Web services are a part of the J2EE API framework. Web services and EJB are complementary technologies. EJB is a way of making an application and Web-service is a way to offer services to other applications or clients. The foundation for this integration was the inclusion of the XML based technologies in J2EE. The specification J2EE1.4 require "To support web service interoperability, the EJB 2.1 specification requires compliant implementations to support XML-based web service invocations using WSDL 1.1 and SOAP 1.1 over HTTP 1.1 in conformance with the requirements of JAX-RPC "

An EJB application exposes a Web service interface through a stateless session bean. The Web service client view may be initially defined by a WSDL document and then mapped to a web service endpoint interface that conforms to this. The WSDL description may be stored in UDDI.

J2EE supports the JAXM protocol, which enables Java applications to send and receive document oriented XML messages. A message bean may receive SOAP messages instead of JMS messages. The message bean must then implement a specific interface for receiving a SOAP message with attachment.

An Enterprise bean may also access a Web service as an ordinary Web service client, but invoking Web services will introduce an unpredictable delay since the calls are synchronous. A Web service endpoint is accessible for a bean through the JNDI API.

4.7 Summary

In this chapter different J2EE technologies have been presented. It has been shown that together they comprise a complete set of API's that can be used to develop distributed applications, and with its recent integration with web services, J2EE applications can be part of a globally Open Service Network. With the integration of MOM based messaging into J2EE, J2EE now supports an asynchronous message paradigm, which is important for the ActorFrame approach. However this approach is currently limited by Web services that still support only synchronous communication with the RPC mechanism.

Chapter 5 Conceptual approach to mapping of models

There are different approaches to software engineering, which focus on how to specify, design, implement, test and deploy software systems. The different approaches can be divided into two categories.

- Implementation oriented where the focus is on the implementation of a system. Although there may exist design and modeling descriptions they are not formal and are often incomplete. The system can usually only be understood in terms of the implementation.
- Model oriented where the focus is on the descriptions of the functionality or the properties of the system. These descriptions constitute more or less formal models of the system and they can be verified, analyzed and understood independently of the implementation of the system. The models can be used for many different implementations.

ActorFrame is based on a model-oriented approach to software engineering. This chapter will give a description of the modeling oriented approach and the impacts this approach has on mapping of ActorFrame models to middleware platforms.

5.1 Model oriented approach

There is seldom an easy solution to the problem of describing a complex reality. It is not possible to have all things in mind at the same time. *Separation of concern* and *conceptual abstraction* are two techniques that in combination can be used to manage the complexity.

Separation of concern in this context means [28] *"to factor out aspects that can be understood one at a time and combined into an understanding of the whole"*. Independent parts of the problem should be identified and described separately. In development of services the functionality of the service can be understood and described independently of how to implement the service. A service like "redraw cash" can be described and understood independent of how the system should be implemented for instance, as an automatic teller machine or handled by a bank clerk. The mapping and deployment of the functionality of the system to the actual realization need to be described. These three main aspects are shown in Figure 5-1 [28].

Deployment defines a mapping between the functionality and the realization of the system, by identifying the technology to be used and where the functionality is to be realized. Deployment description focuses on aspects of the system that come in addition to the core functionality such as distribution, use of middleware and class libraries.

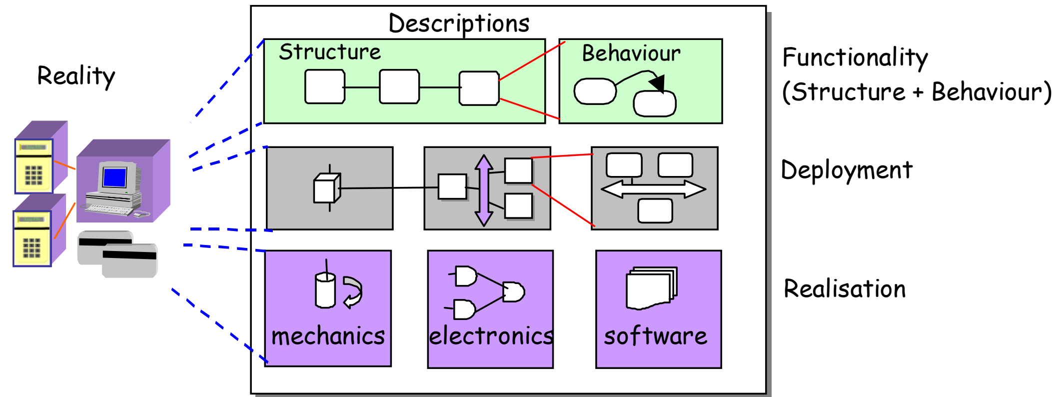

Figure 5-1 Three main aspects [28]

Abstraction is built on the idea to remove irrelevant details in order to focus on the essential from the actual context. Conceptual abstraction means [28] *"to replace low level concepts representing technical detail by more abstract concepts that are better suited to describe and study some aspects"*. The essential part of a service, like most software applications, is the ability to perform logically and to handle information related to the service. For instance it is better to describe a "depositing cash" service using the UML language than the C programming language.

The reference model for Open Distributed Processing (RM-ODP), has defined a different set of viewpoints: Enterprise, Information, Computation, Engineering, and Technology, where the three first viewpoints cover the functionality aspects, and Computational and Engineering cover Realization.

The choice of concepts for the description of the functionality is dependent on the characteristic of the system to be described. Applications to be deployed in Service Network as described in chapter 1.1, which are characterized by communication between concurrently operating and physically distributed objects, are best described in terms of communicating state machines that encapsulate data. As shown in chapter 2, UML2.0 has those concepts needed to describe this type of functionality.

There are different approaches [28] to the process of transforming the description of the functionality to a realization:

- *The elaboration approach* where the functionality is described with informal or semi informal languages, and therefore the models are incomplete and not precise. Additional details have to be added by elaboration, during deployment and during realization. This is the most common approach in software engineering today also among those that use UML to describe the functionality.
- *The translation approach* where the functionality is described completely and formally. The deployment description is kept orthogonally to the functional description to obtain transparency. There is also a trend against leaving out the deployment information for the deployment of a component. These two descriptions define the input to the realization, as such can the functionality

survive technology and platform changes. The translation can be both automatic or manually.

Model Driven Architecture (MDA) defined by OMG supports the translation approach. The PIM model is the functional description and the PSM describes both the implementation and the functionality aspects of the system.

5.2 Translation of models

A common way to specify the deployment is to tag the different elements of a model. These tags are used by the translator to select the mapping rules that should be applied for each model element. There are two principle types of tags.

- *Platform specific tags* where the different tags represent concepts in the realization platform. For instance if a J2EE platform is used, the tags can identify the different types of Enterprise java beans. This approach can be used with MDA. The PIM is defined with UML and it should be independent of the realization. If a J2EE platform shall be used in the realization, the different elements of an UML model should be tagged by stereotypes defined in a UML profile for this platform. The PIM model is then tailored with deployment information, which transforms the PIM model to PSM model.
- *Concepts specific tags* where the tags represent concepts in the domain of interest. They normally do not tell anything about the realization, only which phenomenon it represents. How to realize these concepts, is part of the definition of the translator. So the translator has to be changed if the realization platform is changed. This approach is often called concept modeling. These concepts can also be defined as stereotypes. The stereotypes are used in the platform independent model as modeling elements. This is different from the use of realization-oriented concepts that are added to the PIM model.

The differences between these two approaches are significant as shown in Figure 5-2. The figure describes the different approaches to transforming of platform independent UML models to an EJB platform using the Java programming language.

When platform specific tags are used there are two complete models that each describe the functionality. The PIM is transformed to a PSM model by adding platform specific information to the PIM model. The PSM model is then transformed to an implementation using corresponding classes from the EJB java package.

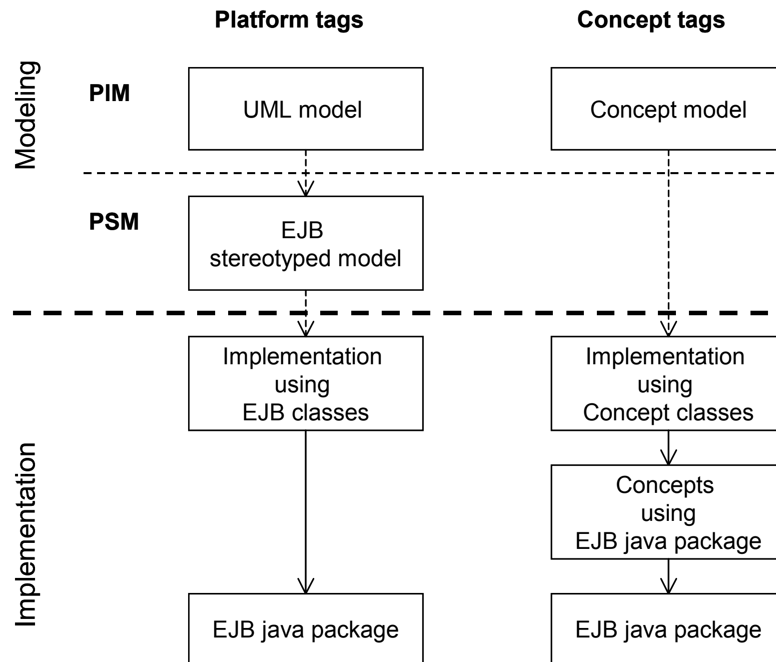

Figure 5-2 Realization versus concept tags

When concept tags are used there are also two models, but they are orthogonal descriptions. There is only one functional model tagged with concepts. The PSM information is placed in own descriptions, but these are independent of the concept model. They only tell how to map the different concepts to the EJB platform. The concept model and the deployment descriptions are used to implement the model by extending Java classes that represents the concepts. These concept classes are again implemented by extending classes from the EJB java package.

The figure is simplified to show the difference between these two approaches. In both cases the transformation will generate additional information to be used at deployment of the EJB components.

The main advantage of the concept-oriented approach is

1. There is only one platform independent model to maintain
2. Most of the implementation may be reused either when the platform changes or when the model changes.

The main disadvantage is that if tools are used for transformation of the models, they have to be customized or made for each concept.

There is a proposal for an UML profile for Enterprise Java Beans developed by the Java community. The next subchapter will describe this profile and answer the question: can it be used to describe ActorFrame models?

5.3 UML profile for Entreprise Java Beans

5.3.1 Introduction

There are two EJB profiles defined in the proposal from the Java community [13].

- The EJB Design profile defines how to model EJB applications by using stereotyped concepts for the different EJB concepts types of beans, interfaces etc.
- The EJB Deployment profile is used to define artifacts in EJB terms such as ejb-jar files.

The Design profile supports two different views.

- The code-centric view uses stereotypes that closely match the underlying Java files. All needed EJB classes and interfaces are supported. This leads to a very implementation oriented view. EJB requires some redundant information to be specified for instance, create methods are defined in the implementation class and in the local / remote interfaces.
- The ejb-centric view is more abstract and avoids exposing implementation details. This approach is more like the approach that the TIMe methodology [49] recommends. The ejb-centric view defines all information needed in an Enterprise Bean class. For example is all methods that are exposed on the different interfaces are shown as stereotyped methods on this class.

5.3.2 UML profile for EJB

This profile supports EJB version 2.0. The UML profile builds on the UML 1.4 with some minor extensions that are included in the draft version of UML2.0.

The most important stereotypes defined in EJB Design Profile are listed in Table 2. The different elements of PIM model can be tagged with these stereotypes telling how the model can be mapped to the EJB platform.

Table 2 EJB Design stereotypes (stripped version) [13]

Stereotype	**Applies To**	Definition
«EJBBusiness»	Method	Indicates that the Method represents an *instance-level* business method, a method that supports the “business logic” of the EJB.
«EJBCmpField»	Attribute, AssociationEnd	Indicates that the Attribute or Association End represents a container-managed field for an EJB Entity Bean with container-managed persistence
«EJBCreate»	Method	Indicates that the Method represents an EJB Create Method.
«EJBEnterpriseBean»	Class	Specializes the standard UML Stereotype «implementationClass». An abstract Class that represents an EJB Enterprise Bean..
«EJBEntityBean»	Class	Indicates that the Class represents an EJB Entity Bean. Specializes «EJBEnterpriseBean».
«EJBFinder»	Method	Indicates that the Method represents an EJB Finder Method.
«EJBHome»	Method	Indicates that the Method represents an EJB Home

		Method, *either* local or remote, which is a *class-level* "business" method, as opposed to a "create", "finder", *etc.* method.
«EJBLocalMethod»	Method	Indicates that the Method represents a method that is exposed on *either* the local or local-home interface.
«EJBLocalReference»	Dependency	A stereotyped Dependency representing an EJB Local Reference, where the client is an EJB-JAR and the supplier is an EJB Enterprise Bean.
«EJBMessageDrivenBean»	Class	Indicates that the Class represents an EJB Message Driven Bean. Specializes «EJBEnterpriseBean».
«EJBPrimaryKey»	Usage	Indicates that the supplier of the Usage represents the EJB Primary Key Class for the EJB Enterprise Bean represented by the client.
«EJBRealizeHome»	Abstraction	Indicates that the supplier of the Abstraction represents an EJB Remote Home Interface for the EJB Enterprise Bean Class represented by the client.
«EJBReference»	Dependency	A stereotyped Dependency representing an EJB (Remote) Reference, where the client is an EJB-JAR and the supplier is an EJB Enterprise Bean.
«EJBRelationship»	Association	Indicates that the EJB Entity Bean at the supplier Association End represents a container-managed relationship field for the client EJB Entity Bean. (EJB 2.0)
«EJBRemoteMethod»	Method	Indicates that the Method represents a method that is exposed on *either* the remote or remote-home interface. The former case is assumed if the EJB Business stereotype is also present.
«EJBSessionBean»	Class	Indicates that the Class represents an EJB Session Bean. Specializes «EJBEnterpriseBean».

To support the ejb-centric approach a generic stereotype *EJBEnterpriseBean* captures all common properties of the different bean types. This is shown in Figure 5-3 [13]. Note that *Core::Actor* which is a metaclass in UML, is not the same as the actor concept defined in ActorFrame.

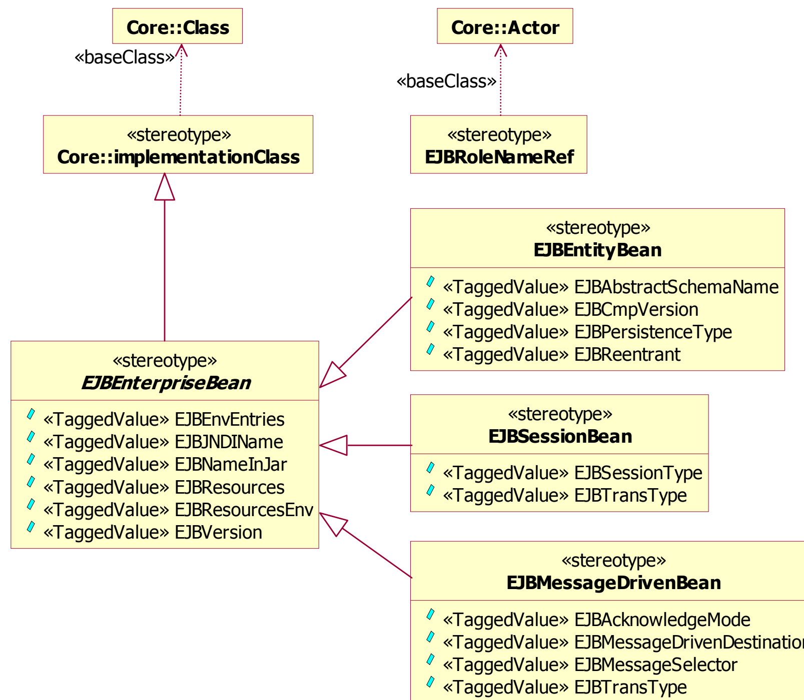

Figure 5-3 Stereotypes and tagged values for EnterpriseBean [44]

The tagged values are mapped to deployment descriptor elements. Not all of these have to be defined. One of the advantages of using the stereotype *EJBEnterpriseBean* in the model is that a single lifeline in a sequence diagram can be used to represent the bean. In the code-centric view the methods calls must be represented with one lifeline for the home object, one for the remote object and one for the implementation class. This makes the sequence diagram unnecessarily detailed.

5.3.3 Limitation of the EJB profile

This profile presented here is based on UML 1.4. UML2.0 introduces new structural concepts as parts, ports and connectors. UML2.0 also supports a more asynchronous communication style and the use of state machines to model complex behavior. How shall these concepts be mapped to EJB?

For each concept the EJB profile has defined, a set of constraints that should be applied to UML classes that represent the concept. The stereotype *EJBEnterpriseBean*, which is the super class for the different beans as shown in Figure 5-3, has set a constraint on UML classes that cannot be active (*isActive=false)*. This prohibits use of state machines in the classes, which is the core concept in ActorFrame. It has also restricted use of the concurrency property of operations in UML to be sequential. In UML2.0 the behavior of operations may be described by use of state machines. This profile allows use of message beans, which support asynchronous communication, but it is not clear how this restricts the UML model from using asynchronous communication when sending signals between UML classes.

5.4 Use of UML profile for EJB for ActorFrame modeling

How the actor concepts should be expressed in the PIM model is important. If the UML profile for EJB is used to represent the actor concept, the actor concept will be a platform specific concept telling how the actor concept can be implemented on the EJB platform. Platform specific information should be avoided in the PIM model. The actor concept must therefore be represented by general UML classes or by predefined classes that represents the actor concept. It is not, in the general case, possible for a translator to know that these specific classes shall be mapped to a specific pattern in the PSM model unless this information is present in the PIM model. If the actor concept shall be mapped to the classes X and Y in the PSM model, the PIM model must also reflect the same structure. This is illustrated in Figure 5-4 where classes X and Y together represent the actor concept in the PIM model. Class Y represents the state machine and class X represents the state data for the actor.

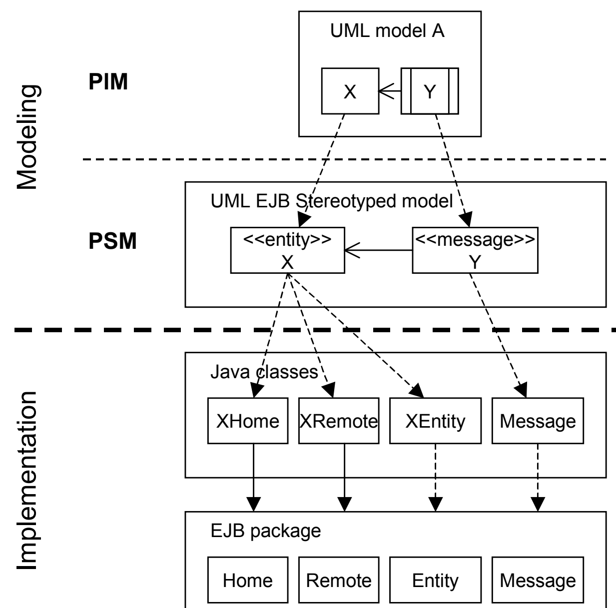

Figure 5-4 Use of UML profile for EJB for ActorFrame modeling

The UML profile for EJB can then be used to make a PSM, where class X is stereotyped with the <<entity>> tag and class Y with the <<message>> tag. The translator can then map objects of these classes to the corresponding Java classes as shown in the figure above.

This solution has the following weaknesses:

- The actor concept is invisible in PIM model, which reduces the effect of using the actor concept.
- The PIM model consists of more classes than necessary because two classes represent each actor.
- The PIM model is inspired by the underlying concepts such as the state data class is related to the to entity bean. Changes in the platform may then affect the PIM model.

A single UML class can also represent the actor concept, but then the problem remains of how to mark this class such as it is recognized as an actor. A better solution is to make an UML profile for ActorFrame and then use this profile in the PIM model. This profile will then express the actor concept independent of the realization. An UML profile for ActorFrame is proposed in chapter 7.2.

5.5 Summary

This chapter has presented a model-oriented approach to software engineering. It has also presented different approaches to transformation of models. If the models are described with formal languages, they can also be used for a translation approach to model transformation. Models are then tagged with specific tags representing either platform or domain specific concepts. An UML profile for the EJB platform was presented. Concepts and approaches defined in this chapter will be used later as a reference for discussion.

Chapter 6 Mapping of ActorFrame to J2EE

There are many possible ways of mapping ActorFrame to the J2EE platform. Each alternative has different characteristics regarding performance, resource usage, persistence etc. In this chapter the different mapping possibilities are discussed separately although they are dependent of each other. However, these dependencies will be taken into consideration when a concrete proposal shall be made as it is done in chapter 8.

6.1 Mapping strategies

In chapter 5, different approaches to implementation of models were presented. The choice of approach depends on the methodologies in software engineering that are used. The intention behind ActorFrame is to support a model oriented approach to software engineering. Formal models of the functionality should be made with ActorFrame concepts described as stereotyped UML models. These ActorFrame models can then be verified and validated through simulation, and code can be made for the target platform. This approach has proven to be beneficial especially if good tools are available, and thereby enabling automatic code generation [45].

A model-oriented approach enables a translational approach to be applied. In chapter 5.3 a UML profile for the EJB platform was presented. This UML profile used realization specific concepts to tag the UML model. But ActorFrame supports concept modeling, so the actor concept should be used in modeling. All elements of the model that represent an actor should be tagged with a concept specific tag for an actor. The translator will for all elements in the model tagged as an actor, do a translation of the element to the specific platform according to mapping rules or pattern.

The mapping of the ActorFrame concepts to the EJB platform and a UML profile for ActorFrame modeling need to be specified. A UML profile for ActorFrame is proposed in chapter 7.2. The rest of this chapter will describe alternatives for mapping ActorFrame concepts to a J2EE platform. Chapter 7 describes a framework called EJBActorFrame, which is a class library used to implement actors on the EJB platform.

6.2 Mapping issues

The following requirements are important for selection of mapping strategies from ActorFrame to J2EE:

- Asynchronous communication between state machines
- Parallel or quasi parallel execution of the state machines (active classes)
- The persistency of state machines

- The scalability of mapping solution meaning the possibility to distribute state machines
- How efficiently the solution uses the resources managed by the container
- Performance achieved by the different solutions
- How the solution is "faithful" to the intention of the design of middleware. For instance: a solution can be possible, but the middleware was not intended to be used in that way.
- The size of the code that has to be implemented. If the implementation is done manually, it is important that the mapping is as trivial as possible. For example how many java classes have to be implemented for each state machine.
- Portability of the application to new platforms.

6.3 Strategies for mapping of state machines

An important question is how the set of state machines shall be mapped to the beans. There exist 3 possible solutions:

1 All state machines are implemented in one bean

2 One state machine is implemented in one bean

3 A combination of 1 or 2.

The choice of solution will influence heavily the characteristics of the implemented systems.

6.3.1 Multiple state machines in one bean

One possible solution is to map multiple state machines into one bean as shown in Figure 6-1. A bean is run in one Java thread and no other treads are allowed to be spawned from the bean. Because of this restriction, the bean itself has to manage the concurrent execution of the state machines. How does the bean execute a state machine and how does it select which state machine is to be started?

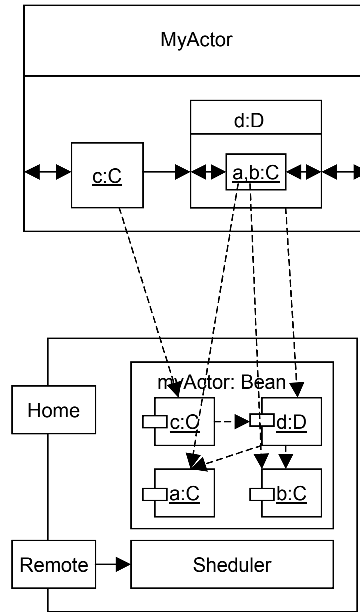

Figure 6-1 All state machines into one bean

A possible solution is to have one internal queue for each state machine. A state machine that has one or more messages in the input queue is ready to execute. If several state machines are ready for execution, the scheduler selects one of the state machines according to a scheduling principle like "round robin". When no state machines are ready, the container then suspends the bean until new messages are sent to the bean. The bean then puts the message into the input queue for the state machine according to the value of receiver address field of the message.

This solution is used by the JavaFrame system. It is very efficient with respect to message passing between the state machines that belong to the same bean. The messages are sent as references between the state machines.

On the other hand this solution has some drawbacks. It does not scale very well. As all state machines reside in one bean, they cannot be distributed without doing a redesign. It does not utilize the management functionality that the container supports for instance transaction control and scheduling of beans.

Another important restriction that has to be fulfilled is that the state machine cannot be allowed to wait during a transition. Ideally a transition should take no time. A state machine should for instance not perform RPC or RMI calls that entail blocking or waiting situations. No other state machines can be executed during the time the transition takes. All interactions that take time should therefore be realized by asynchronous communication, thus avoiding and allowing other state machines to be executed while awaiting a response.

In the next chapter another solution is presented that does not have these limitations.

6.3.2 One state machine in one bean

The other extreme mapping strategy is to use one bean to implement one state machine. Then there is no need for a special scheduler, because the container does the job. The bean, which is actually the state machine, receives messages addressed to it and executes the transition. Each state machine owns one thread and it may interact with other resources since blocking of the state machine is not a problem. The container is responsible for executing concurrent state machines that also receive messages. The state machines can be distributed freely on different machines thus achieving a more scaleable solution.

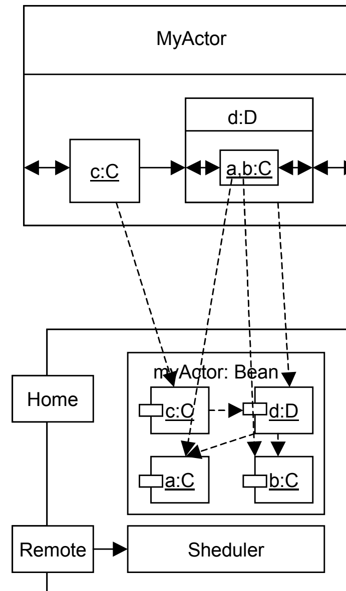

Figure 6-2 One State Machine into on bean

It is also easy to use the transaction control service that the container offers. For instance if a state machine sends several messages during a transition and one or more of these messages fail to be received by a state machine then transaction control will redraw the messages that have been sent. A precondition is that the messaging system is integrated with the transaction control system that the container has. As described in chapter 4.4, the Java Message System is integrated with EJB as defined in J2EE 1.4.

A major drawback with this solution is that it uses too many resources. A system may consist of thousands of state machines and requiring one JMS queue or message bean for each state machine will probably not scale very well.

6.3.3 Group of state machines in one bean

Between these two extremes described above there exists a solution that balances the number of beans required and the number of state machines in each bean. Normally a system consists of a smaller number of different *types* of state machines. The number of *instances* of each type of state machine may vary a lot from a single one to thousands. As illustrated in Figure 6-3 all instances of a state machine type can be implemented by one bean.

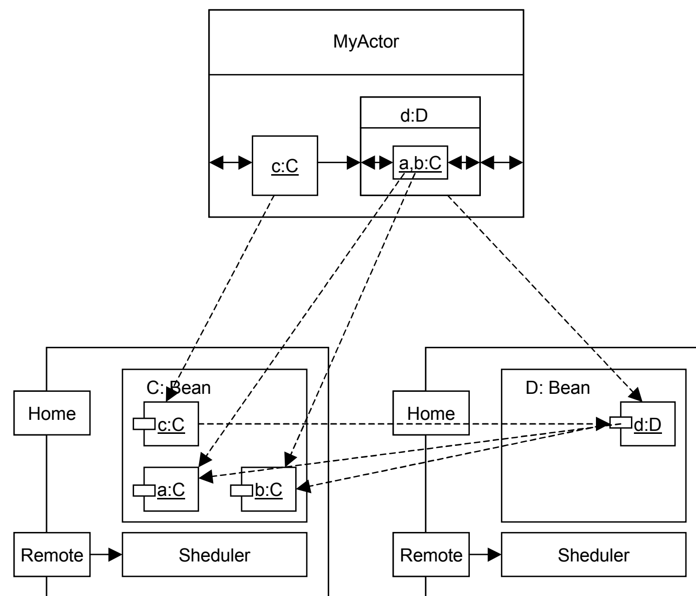

Figure 6-3 State machines of same type into one bean

This solution maintains the strengths of the above solution, and it reduces the weaknesses of the two other solutions. This solution has several advantages:

- The number of beans are reduced, which is a drawback of the "one2one" solution
- Transaction control between state machines is possible
- Enable an efficient handling of persistence of state data
- The resource control in the container is better utilized
- All instances of a state machine type share common behavior that can be implemented as the behavior of a bean.

6.4 Kind of bean to be used

In chapter 4.2 the different types of bean were presented. In the following subchapter it will be discussed what the differences between the beans will mean for the mapping.

6.4.1 Session beans

The intention of session beans is to serve typical client requests that last some time. Stateless beans are used for single requests and statefull beans are used for multiple requests from the same client. A statefull bean keeps state information during a session. It is possible to use both variants for implementation of state machines. The choice depends on how long time the state machines should last. In most cases a statefull session bean could be used for implementation of state machines that are relatively short lived and need not to be persistent.

A session bean can be used for implementation of both single and multiple state machines. It exists as long as a client has references to it. The EJB standard recommends that only a single client should keep a reference to a session bean.

Session beans are not allowed, according to the standard, to receive JMS messages asynchronously. However session bean can send and synchronously receive JMS messages.

6.4.2 Entity bean

Normally entity beans are used for storage of data. A CMP entity bean can be used to achieve persistence of state data. It also provides transaction control. An entity bean can also be used to implement instances of a state machine type. A client can access an entity bean by finding the methods that the entity bean provides. Then it calls the state machine with a message. The state data is accessed directly during a transition.

Access to entity beans should normally be relatively short-lived. An entity bean can use JMS to send messages and receive messages synchronously. However, entity bean should not wait for synchronized reception of messages because this will cause other beans that call the entity bean, to be blocked.

6.4.3 Message bean

A message bean is designed for receiving messages from JMS. It does not provide persistence of data and it is normally short-lived. The JMS Destination controls the execution of the bean. When a message arrives at the JMS Destination specified at deployment time, the container activates a random message bean from a pool of message beans.

A message bean can implement a state machine, but it cannot store data for that particular instance. It has to use another storage medium for that or it may use CMP entity beans which also provide persistence and transaction control. The behavior structure of the state machine, which is common for all instances of the same state machine type, can be stored in the message bean.

The container provides multiple instances of message beans in a pool created at deployment time. So more that one message bean may receive messages for the same state machine. This must be prohibited because the same state machine cannot execute simultaneously. A bean managed transaction control can be used to lock entity beans for access while state machine execute transitions. This mechanism is the Isolation part of the ACID mechanism that EJB supports.

A message bean is very simple to implement. It has no remote or home interfaces. This solution will scale well. Only few message beans are necessary for a specific state machine type. The state data for each state machine will be “loaded” and “stored” for each transition. This may on other hand lead to performance problems.

6.5 Asynchronous communication

As outlined in chapter 1, asynchronous communication between the state machines is essential. The first versions of J2EE did not provide asynchronous message passing.

Version 2.0 integrated JMS with EJB to achieve this. The message bean is integrated with JMS and this provides a good solution for asynchronous communication between state machines. The other types of beans can only send and synchronously receive JMS messages, which means that the bean can call a JMS destination and then wait for a message. This will block the bean to be called by other beans or clients.

JMS provides different mechanism that can be used to make an infrastructure of state machines. In principle there are the following possibilities:

1. All state machines share the same queue.
2. One queue for each state machine
3. Combination of alternative 1 and 2

The advantages and drawbacks of each of the solutions are discussed below.

6.5.1 All state machines share the same queue

As illustrated in Figure 6-4 it is possible to let all state machines receive messages from the same queue. JMS supports a mechanism to select a message by specifying a pattern that the message must match. The pattern can be evaluated against fields in the header or property fields of a message. Using a property field that specifies the receiving address of the state machine can be used to do the message selection. This works also for beans that implements multiple state machines.

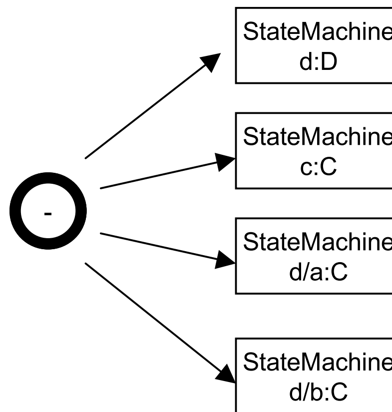

Figure 6-4 All state machines share same queue

This solution is simple both to implement and to use. All output messages are sent to the same queue. There is no need for addressing the right queue.

However this solution does not scale very well. A real application with many state machines waiting for messages from the same queue will probably cause poor performance. The container will spend time checking the selection patterns to find the right state machine to send the message to.

6.5.2 One queue for each state machine

It is also possible to create one queue for each state machine. This is shown in Figure 6-5. With this solution the message selection concept in JMS is not needed. The state machine that waits on a queue shall receive all messages that are sent to that queue.

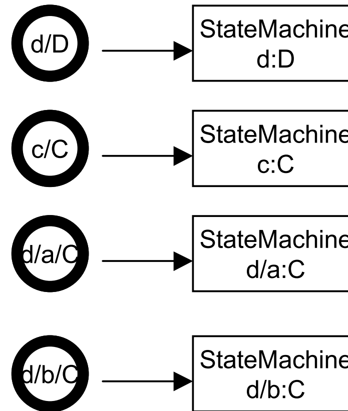

Figure 6-5 One queue for each state machine

An advantage of this solution is the simplicity of the process of receiving messages. On the other hand the state machine must select which queue it shall send messages to. So sending message is a little bit more complicated.

This solution does not scale very well either because of the number of queues that have to be created. This probably requires too much memory and processing resources on the application server.

6.5.3 Combination of alternative 1 and 2

A better solution is to balance the number of queues with the complexity of receiving the message. A solution is to group all state machines of the same type into one bean and let this bean wait for message from only one queue. This is the same principle solution as presented in chapter 6.3.3.

This solution will reduce the number of queues and increase the performance. The receiving bean needs only to “load” and “store” the state data for the state machine that is addressed in the message.

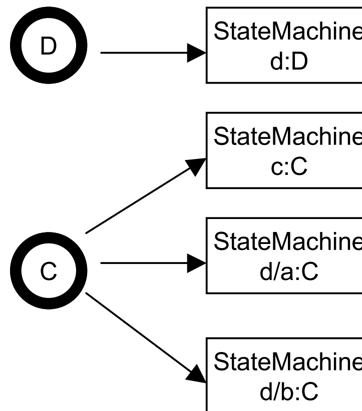

Figure 6-6 State machine of same type share one queue

6.6 Structural relations

Structural relations are concepts in UML that describe the structure of the model. It describes what a class consists of (parts), which classes are associated and how classes are connected to each other.

It is important to understand that this structural information serves different purposes.

- It describes or specifies what are legal associations, what a class may consist of and how ports are connected to each other. At runtime this information should be used to check the legality of operations.
- The structural information is also an important part of the behavior of the model. This information must be stored at runtime to take care of changes in the runtime model such as creating or deleting parts or associations. For example, to delete an instance of a class also means that all its inner instances must be deleted.

As long as a state machine is part of a class, the same structure also applies to the state machines. Even if state machines are implemented in a flat structure, the structural information in the implementation needs to be taken care of.

How to map this information for each of the concepts is described below.

6.6.1 Parts

The *part* concept in UML, as described in chapter 2.3, is used to make a structure of a class. It is used to construct a class that consists of other instances of other classes. These inner parts exist only as part of the surrounding object of the class. This is illustrated in the Figure 6-7. The structure is a part of the behavior of the model. For example, if the state machine *d:D* is deleted, all its inner state machines will also be deleted.

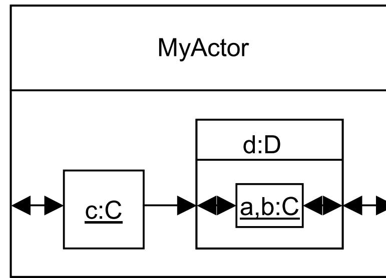

Figure 6-7 Parts in UML2.0

There is no counterpart in EJB to the *part* concept. The information about the structure has to be stored in the state machine. This is done in ActorFrame by extending the state data of the state machine with information about its inner actors and the containing actor.

Another possible solution is to use CMP entity beans for storing this information. the "SQL" statements in EJB can then be utilized to find those state machines that are part of its structure.

6.6.2 Associations

Mapping of *associations* can be done in a similar way as for *parts*. In this context *associations* are references to other state machines. But EJB also has a concept called *ejb-ref* and *ejb-local-ref* that is used to keep references to other beans. The only difference between these is that *ejb-local-ref* refers to the local home interface, which means that the referenced entity beans have to reside in the same JVM. This is guaranteed when the two beans are deployed in the same ejb-jar. These references are set in the deployment descriptors.

The entity bean also supports associations between locally deployed entity beans. This may be used to map UML association between instances of passive classes.

Ejb-ref and ejb-local-ref may be used to map one-to-one references to other ejbs that are not state machines, but they cannot refer to message beans. The most general solution is therefore to associations as part of the state data of the state machine.

6.6.3 Connectors

Connectors (see chapter 2.3.3) are used to connect ports together. A connector can convey messages. Connectors could be mapped to ejb-references, but should then be restricted to connect objects of passive UML classes. As will be presented in chapter 6.7, JNDI is used to find the addresses of other state machines, so connectors could be mapped to JNDI names set at deployment time.

6.6.4 Ports

Ports, see chapter 2.3.2, are used to specify connection points of a UML class. All communication with external objects can be done through ports. Ports can also pass messages. Ports can be mapped to JMS destinations, which may either be a queue or a topic. State machines are logically connected to JMS queues. A queue will then

represent a one-way port in UML. Messages sent to the queue, will be received by the corresponding state machine. This solution will be further described in chapter 7 where the EJBActorFrame is described.

6.7 Naming of state machines

Naming is a critical issue when it comes to addressing state machines. In UML objects may have references to other objects. Sending a message to a state machine is done by using the reference to the other object.

An ejb bean can be globally referenced. JNDI (see chapter 4.5) is used to store global names for the beans. These names can be set at deployment time, so one solution is to map names of *ports* to JNDI names that refer to a JMS destination. State machines will then be associated with the JNDI name of the JMS destination.

Another problem is to make names globally unique. The instance name of a state machine in UML has to be qualified with the instance names of the containing state machines. Instances of the same state machine type can be created with the same name, but only if they are created in different name scopes. This is illustrated in Figure 6-8, where two instances of class *C* are created, but they can be distinguished by adding the instance name of containing object. These two instances will therefore be named in the context of class *MyActor*, as *c/C* and *d/c:C*.

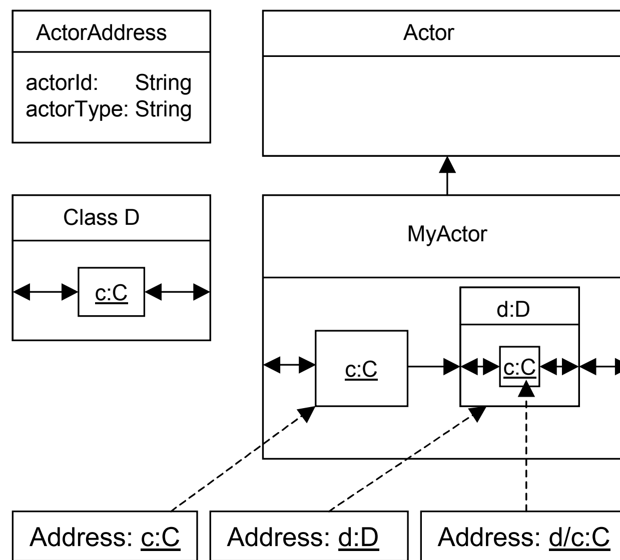

Figure 6-8 Name scope for actors

6.8 Summary

This chapter describes different solutions to mapping of ActorFrame concepts to the J2EE platform. Each concept has been discussed separately, but some of these mappings are dependent on each other. This has to be considered when a concrete mapping solution shall be made.

Chapter 7 Framework for implementation of actors on J2EE

7.1 The mapping solution

The mapping solution selected is according to the framework approach described in chapter 5.1. The UML model is modeled using Actors, which are described in chapter 3.2.1. In chapter 6 alternative mappings are discussed. There is not a single mapping solution that fits to all implementations. This mapping is chosen to solve the problem stated in chapter 1.7.

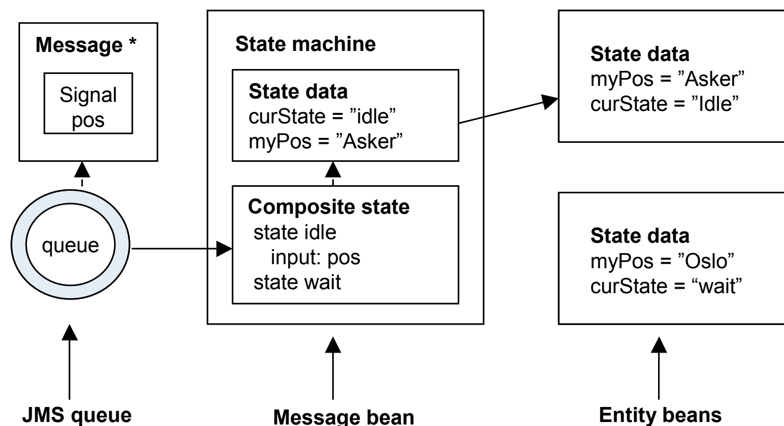

Figure 7-1 Implementation of state machines

The principle solution for mapping of state machines to an EJB middleware platform is shown in Figure 7-1. The state machine is implemented as part of a message driven bean that receives signals from a single JMS queue. The state data is stored as entity beans. The state data is copied to the state machine to obtain a backward compatibility of previous implemented state machines. The reasons for selecting this mapping solution are:

- Use of entity beans to achieve persistency and transactional support of state data.
- Use of message beans to get a seamless integration with JMS, to achieve asynchronous communication and to limit the amount of queues.
- Use of entity beans, which are optimized for large amounts of data, to achieve scalability and persistent storage of state data.
- To use EJB as it was intended.

The main drawback caused by splitting the state data from the state machine behavior is partly reduced by using local references between the message bean and the entity bean. Calls to state data are then done by reference avoiding the extra burden caused by invoking RMI.

The implementation is split into three layers as shown in Figure 7-2 EJBActorFrame.

- Java code that implements the application specific functions. It uses ActorFrame classes from the EJBActorFrame package.
- EJBActorFrame is the actual implementation of the ActorFrame concepts that the model is using.
- The EJB package provides classes for implementation of EJB applications.

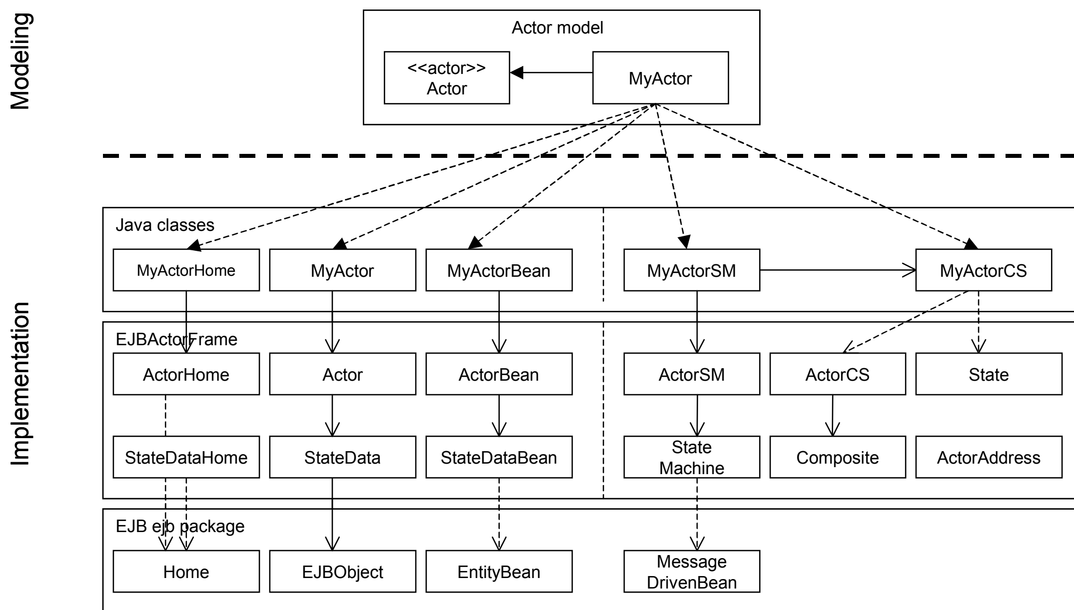

Figure 7-2 EJBActorFrame

It is the implementation of the classes in the EJBActorFrame package that is specific for the EJB platform. These classes have the same signature as the classes in the ActorFrame package, which ensure the portability of the implementation to other middleware platforms.

Figure 7-3 describes how the different parts of class MyActor are mapped to EJBActorFrame classes. Each of the numbered arrows represents a mapping. These are

1. The actor type is mapped to the name of the different interfaces of the entity bean and to the class name for the entity bean implementation class.
2. The instance name “actor” is mapped to a primary key representing the entity bean instance.
3. The state data is mapped to persistent data in the entity bean.
4. Methods for set and get values of the different state data are defined in the entity bean’s remote (local) interface such as *getMyPosition*.

5. The state data is also mapped to property definition of a variable of class *ActorSM*.
6. The behavior definition of the class *Actor* is mapped to the class *ActorCS*.
7. The port *Actor* is mapped to a JNDI name of the input queue for the state machine.

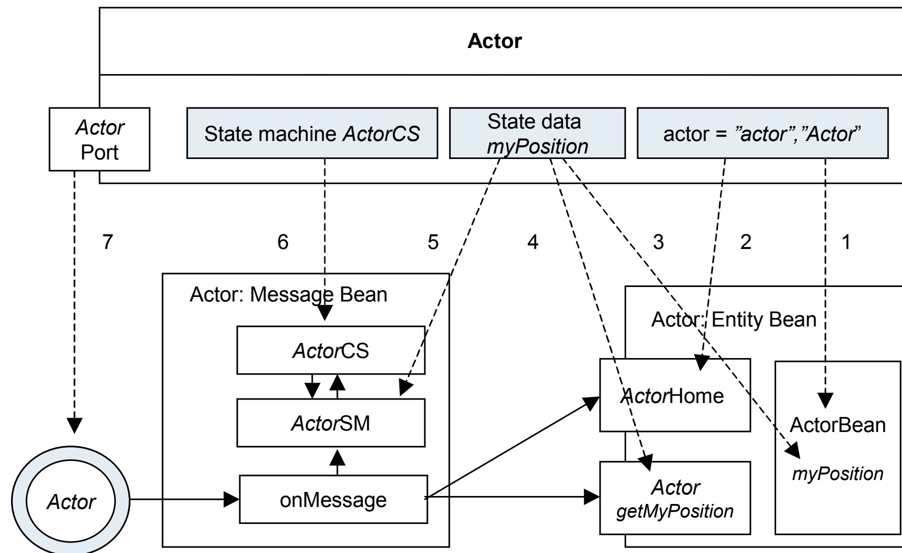

Figure 7-3 Mapping of Actor class to EJBActorFrame classes

The following subchapters will describe the different parts (classes) of EJBActorFrame. A mapping of an example is described in chapter 8.

7.2 UML profile for ActorFrame

The stereotypes needed to model ActorFrame are defined in the profile package ActorFrame as shown in Figure 7-4. The sterotypes are used to define constraints and to add tagged values to stereotype classes. The stereotype *actor* represents the actor concept in ActorFrame. Actor should not be mistaken with the actor concept that is used in use case diagrams in UML. The stereotype *ActorMsg* represents the only message type that actor may receive and send. The stereotype *ActorAddress* represents the name of an actor instance. All these stereotypes are described below.

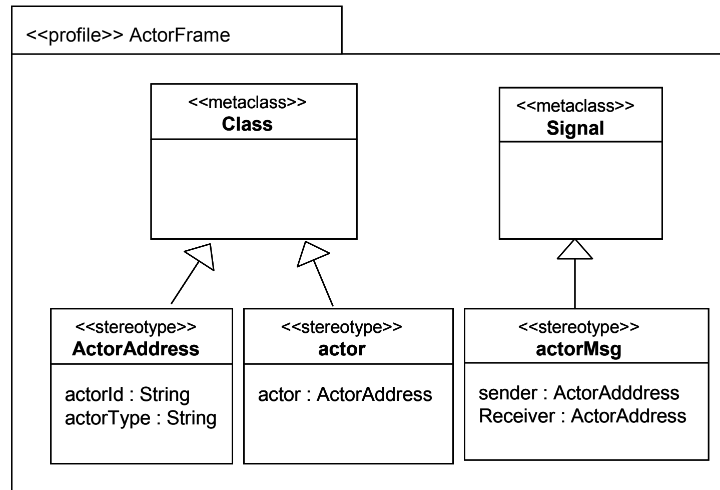

Figure 7-4 Profile for ActorFrame

7.2.1 Actor

The stereotype actor extends the metaclass *Class* in UML and it represents the actor concept defined in ActorFrame. The actor concept can also be described in UML as shown in Figure 7-5, which is stereotyped with actor. The *Actor* class has one in port and one out port. An actor has an "own behavior", which is illustrated with a rounded rectangle connected to the in port. The "own behavior" is the actor behavior such as the RoleRequest protocol described in chapter 3.2. The behavior is described with a state machine that is defined separately. The actor class can have inner actors as parts and their ports are connected to the ports of the containing actor.

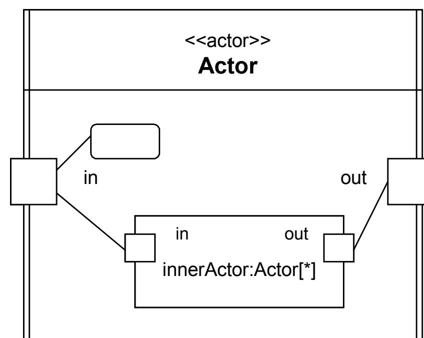

Figure 7-5 Class Actor

The actor concept in ActorFrame supports a subset of features in the class *Class* in UML. The following constraints are therefore placed on the stereotype:

- Provided interfaces have only signals stereotyped with *ActorMsg*
- A port has either a provided or a required interface: the in port
- The *Actor* class has only one port with provided interface: the out port
- The *Actor* class has one port with required interface
- The *Actor* class has no operations that are publicly visible

- The *output* statement in the actions language can only send to a port with a required interface
- An *Actor* supports the *RoleRequest* protocol
- All instances of class *Actor* must have a unique instance name
- The class is set to be active meaning that it has its own thread of control
- The in and out ports are protocol ports, which mean that they contain own behavior
- The class *Actor* has an own state machine that supports the *RoleRequest* protocol

7.2.2 ActorMsg

The stereotype *ActorMsg* is a signal with the properties sender and receiver, which are stereotyped with *ActorAddress*.

There is one constraint for *ActorMsg*. The receiver of the signal has to be set prior to sending of *ActorMsg* signals.

7.2.3 ActorAddress

The stereotype *ActorAddress* represents the global address of an actor instance. It has the properties *actorId* and *actorType,* which both are strings.

There is no constraint defined for ActorAddress.

7.3 Principle behavior of EJBActorFrame

In Figure 7-6 the conceptual solution of the implemented state machine is shown. An asynchronous message passing is obtained by using a JMS queue for reception of signals to the state machine. A message driven bean contains the behavior of the state machine and the state data is stored in entity beans.

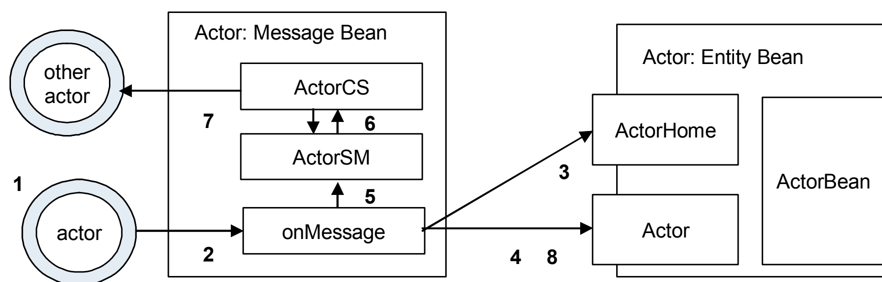

Figure 7-6 Operation of an actor state machine

Input signals trigger transition in the state machine. This is obtained through following the steps numbered according to the figure above.

1. A JMS message is sent to the queue named "actor".

2. JMS calls the *onMessage* method in the message bean with the message as parameter. The *onMessage* method checks if the message contains an actor message (type ActorMsg). If not, the message is skipped.

3. The reference to the entity bean's home object was obtained through lookup in JNDI name server when the message bean was created. This reference is now used to call the *findByPrimary()* method, which finds the entity bean that contains the state data of this actor instance. The instance identification is obtained through the receiver address of the received message. The *findByPrimary()* method returns a reference to the actor object of the entity bean.

4. The *onMethod()* calls the method *getCurrentState()* in the *Actor* interface of the entity bean, which returns a string *currentStateId* representing the current state. Eventually a transaction will be started.

5. The actual state object is found by searching through the state hierarchy to find the state that is equal to the *currentStateId*. A reference to this state is stored in the *currentState* variable.

6. The *execTrans()* method of the current state object is called with an instance of *ActorMsg* as a parameter. *ExecTrans()* checks if the current state contains this signal. If so, a transition is triggered.

7. New signals may be sent during the transition to other JMS destinations representing the input queues for other actors.

8. If the destination of the transition is a new state, the *currentStateId* is updated. The state data is stored in the entity bean by calling the *setCurrentState()* method of the entity bean. The transaction is eventually ended.

The steps above are similar for all sub types of class *Actor*. If sub types of class *Actor* are defined, the state date for the subtypes may be extended with new attributes. If so, the instances of these attributes have to be retrieved from the entity bean defined for that subtype. This will be explained in chapter 8, where an example of implementation of subtype of an actor is shown.

7.4 Description of EJBActorFrame

EJBActorFrame consists of two parts of classes, behavior and state data related classes. In Figure 7-7 the classes for implementation of the behavior of the state machine are shown. Most of these classes are equal to the classes in ActorFrame described in chapter 3. The main difference between EJBActorFrame and ActorFrame is implementation of the class *StateMachine*. This class implements the interface *MessageDrivenBean* from the EJB package. As described in chapter 4.2.3.3, the *MessageDrivenBean* interface contains only the *onMessage* method, which receives messages from specified JMS queues. Class *StateMachine* has references to the interfaces of the entity bean where the state data of the state machine is stored.

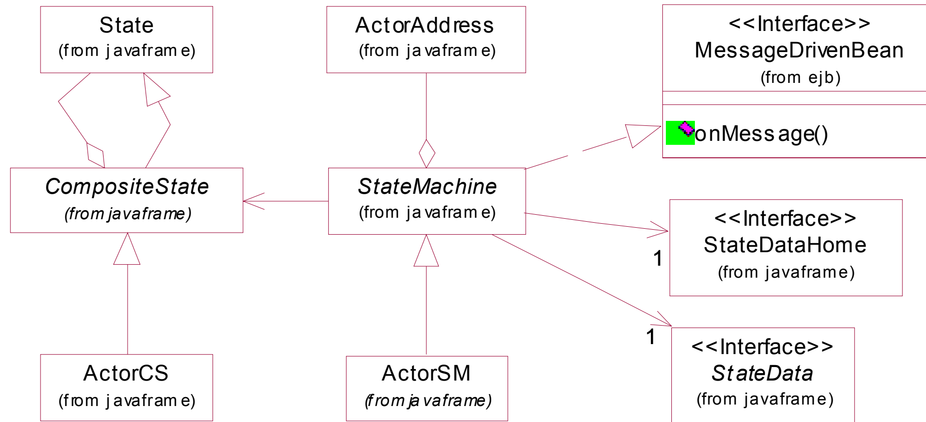

Figure 7-7 EJBActorFrame classes

In Figure 7-8 the classes for implementation of state data as an entity bean are shown. The class *StateDataBean* is the implementation class of the entity bean. It contains the definitions of each data element of state data. State data contains a string *currentState* that represents the current state of the state machine and an actor identity *myId*, which is also the primary field for the entity bean.

As presented in chapter 4.2.3.1 the properties of an entity bean are specified by defining abstract "get" and "set" methods for each of the properties. These are listed in the operation field of the class *StateDataBean* marked with cursive script. The other methods in class *StateDataBean* are the standard methods that an entity bean has to implement according to the EJB standard. These methods are empty, but they may be overridden in subclasses of the class *StateDataBean*.

Figure 7-8 State data classes

Each of the get and set methods are also specified in the interface called *StateData*. These methods define the business methods that are available for other beans, which in this case is the class *StateMachine*.

The interface *StateDataHome* defines the methods used to find or create entity beans. The method *findByPrimaryKey()* finds the state data for a specific state machine instance. The primary key *myId* represents a unique identification of a state machine instance.

The class *StateMachine* is the only client that accesses the *StateData* entity bean. To optimize access to state data the local home and remote interfaces are used[1]. A method call to the state data bean is then done by reference and not RMI. A prerequisite is that both the entity bean and the message bean are deployed in the same jar file.

7.5 The implementation of state machine as a message bean

The class *StateMachine* implements the message bean interface. This interface contains one method *onMessage* that is called by the container when the message bean receives a JMS message.

StateMachine is an abstract class. It contains abstract methods that must be defined in the concrete subclasses of class *StateMachine*. These methods are listed in Table 3. Common for all these abstract methods are that they are specific for the EJB bean that implements the subtype of class *StateMachine*. These methods are called from the class *StateMachine*. In the definition of the class *StateMachine* the goal has been to simplify what has to be implemented in the subtypes. In general the class *StateMachine* contains functionality needed to execute a general state machine including persistency of state dataAn example is given in chapter 8 to illustrate this.

Class *Actor* is a subtype of class *StateMachine* and it defines all the abstract methods listed in the table below. A new class that is a subtype of class Actor does not therefore define these methods.

Table 3 Abstract methods in class StateMachine

Abstract method	Description
getData	Reads the state data that is specific for the subtype into the corresponding attributes of the sybtype
storeData	Stores the attributes of the subtyped state machine that is specific to the subtype.
createInstance	Creates a new instance of the state machine, which means a new bean
findEntityBean	Finds an entity bean with the specified instance name and returns a reference to the local object
findAllInstances	Finds all instances of a state machine type and returns a collection of references to the local objects

[1] Because the application server JBOSS seems not to support use of local interfaces of beans, the ordinary remote interfaces were in the implementation of EJBActorFrame package. JBOSS supports optimization of calls between beans deployed in the same container.

The behavior of the state machine is defined by extending the class *CompositeState*. The implementation follows the pattern defined for JavaFrame.

7.6 Persistence of state data

The major advantage of using entity beans is that persistency of state data is achieved. State data is stored in underlying databases and the container synchronizes the state of data in memory with the database. In this system, which is based on read and write of state data at every transition, this may cause performance problems. It is therefore important that the size of data to be stored in entity beans is as small as possible.

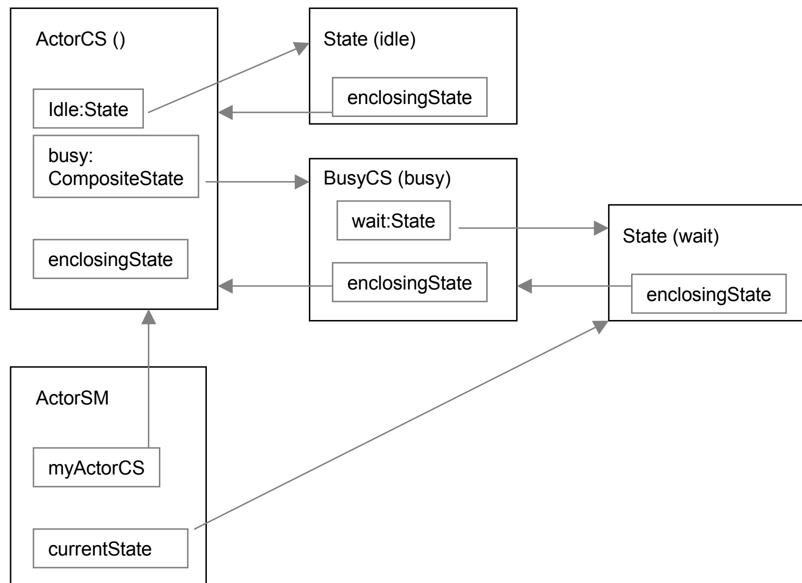

Figure 7-9 Dependency of the state hierarchy

The state data for an actor type is current state and the instance id of the state machine. In JavaFrame current state is a reference to the state object. That solution is appropriate when the current state is kept in memory during the lifetime of the actor. But not appropriate in the case where the state data is stored in a database, all variables in the state data are serialized before they are stored. When an object is serialized all referenced java objects are also included in the serialization. Figure 7-9 shows an example of structure of a state machine. As this figure shows *currentState* in class *ActorSM* refers to the state wait. As explained in chapter 3.3.2, a state also has a variable *enclosingState* that refers to its enclosing state, which in this case is the state busy. A *CompositeState* contains references to its sub states and when *currentState* is serialized the whole state hierarchy is included in the serialization. This is a poor solution regarding performance and use of memory resources.

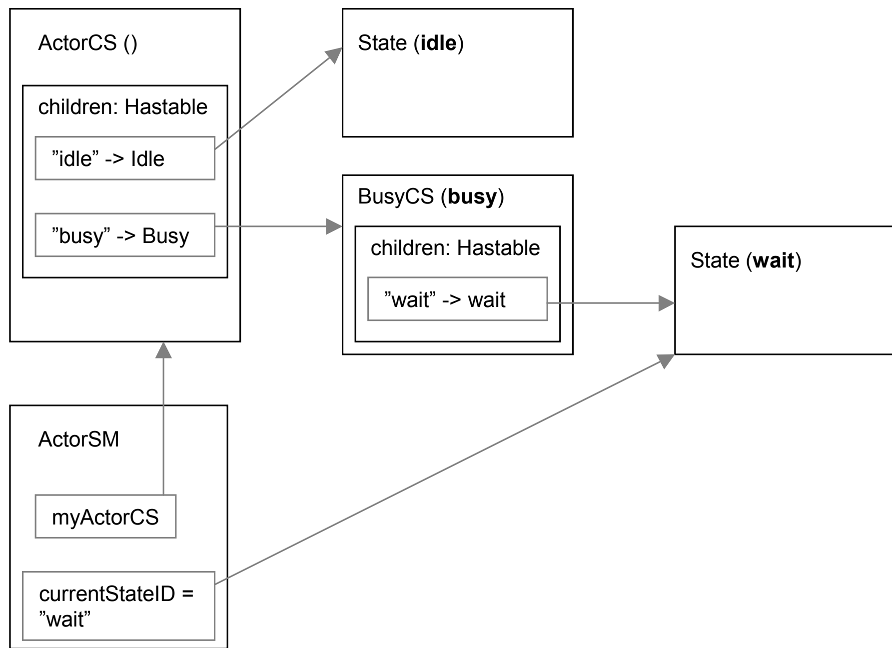

Figure 7-10 Current state represented as a state id

A string is used to refer to current state as illustrated in Figure 7-10. A new table is included in the class *CompositeState* that contains a reference to all its sub states. The table is indexed with the state identification. A new variable *currentStateId* is included in the class *StateMachine* that keeps the state identification as a reference to the current state. This variable is stored in the entity bean as part of the state data.

At the next transition the *currentStateId* is used for searching in the state hierarchy for current state object. A recursive method *findCurrentState()* searches first the top level composite state, which in this case is *ActorCS,* to see if this composite state contains a state equal to the value of *currentStateId*. If not, all sub states of the children table are search recursively. The figure above shows the current state “wait” is found in *BucyCS* and the variable *currentState* is set to refer to this state.

7.7 Use of JMS

JMS, see chapter 4.4, is used to achieve asynchronous communication between actors. The queue mechanism in JMS is design for architectures where multiple clients send messages to a single recipient. This is in line with actor concept where actors are sending signals between each other.

As described in chapter 6.5 there are alternative solutions on how to organize the structure of JMS queues. The solution shown in Figure 6-6, uses a single queue as input to all instances of the same actor type. A typical system will then consist of only a few queues. Addressing an actor is the same as sending a signal to the queue attached to the message bean that implements the actor type.

JMS queues and JMS topics are called managed objects in the J2EE standard. Managed objects are created and deleted independently of the deployed beans. The middleware vendors may do this differently. In the application server JBOSS a separate xml file

defines the JMS queues that are deployed in an application server. The queue names are stored in JNDI and the beans can use JNDI to find references to the names as described in chapter 7.8.

In EJBActorFrame the message type *Object* is used to convey actor messages. The class *ActorMsg* is serialized by JMS before it is delivered to the destination. Automatic acknowledge is used to guarantee that a message is delivered to the message consumer, which in this case is a message bean.

Other clients, like servlets and other java applications, can access JMS queues. This makes it easy to integrate loosely coupled components distributed on the net. Clients implemented in other languages can also communicate via JMS.

7.8 Use of JNDI

JNDI is as described in chapter 4.5, a naming directory containing references to deployed EJB components, JMS destinations and environment variables. JNDI in EJBActorFrame is used to read environment variables and to find references to other actors. As mentioned in the previous chapter, the JNDI name of the queue is equivalent to addressing a specific actor type.

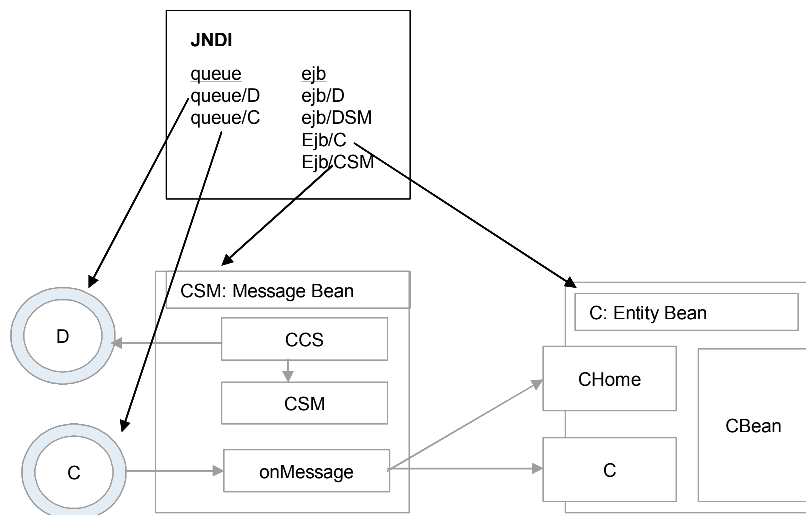

Figure 7-11 JNDI names of actors

In Figure 6-8 an example of a name scope for actors is shown. In JNDI these addresses will be shown as names for JMS queues and beans as shown in Figure 6-8. The EJB standard recommends that JNDI names for beans and JMS queues are qualified with the “/ejb” and “/queue” respectively. The figure shows that JNDI names for queues have the same names as the actor types C and D. These names are used when actors are sending signals to other actors. JNDI names for entity beans are only the name of actor types. The JNDI name for an entity bean that keeps the state data, is accessed only by the corresponding message bean. Names of message beans are not stored in the JNDI, because message beans are not allowed to be called from other beans, they can only receive JMS messages through queues or topics.

7.9 Summary

This chapter describes the EJBActorFrame Java package. It consists of a set of classes that are extended to form new subtypes of actors. A message bean receives messages from a JMS queue for all instances of an actor type. The state data for each instance is stored in a separate entity bean, which provides persistency.

JMS is used to obtain asynchronous communication between actors. JNDI provides a lookup service for beans, JMS queues and environment variables. An UML profile called EJBFrame, for actor concepts has been defined and this profile is used in ActorFrame modeling.

Chapter 8 Implementation of a service using EJBActorFrame

An example of how to map an application modeled with ActorFrame concepts, to the J2EE platform is presented in this chapter. The example is described using the EJBFrame profile package for UML2.0. The application is implemented using the EJBActorFrame Java package described in chapter 7.

The figures in this chapter are not complete or have correct UML syntax. The arrows in associations always closed although in most cases these should be open. It does not either contain a complete description of the design or the implementation. The presentation is meant to give the reader an understanding of how the different parts of an ActorFrame model are implemented. State machines are almost coded according to the pattern proposed in JavaFrame.

8.1 Traffic news – a context aware service

Context Aware Traffic News (CATN), is a service that restricts the flow of traffic news to only news that is relevant to the user. The selection criterion is based on the current position of the user. A typical situation is a user that is traveling by car who is approaching a traffic jam. The user, who subscribes to this CATN service, receives an sms message on its mobile phone telling the user about the traffic jam 5 km ahead. The user can then drive an alternative route to his destination. The user does not receive traffic news that is not relevant for him.

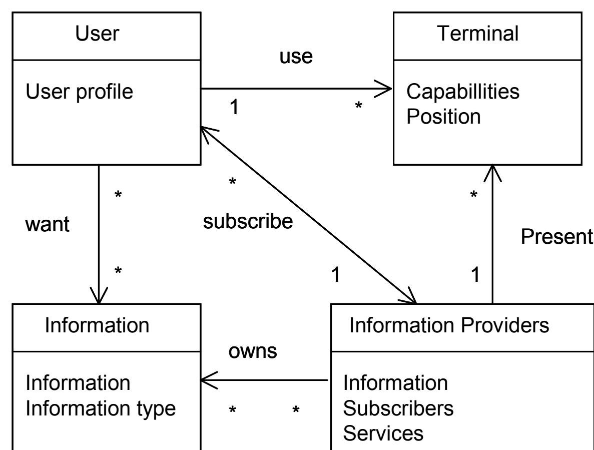

Figure 8-1 Domain model of the CATN service

A UML domain model of this service is shown in Figure 8-1. A domain model describes the most import concepts in the domain of interest. The model describes that

User wants *Information*, which in this case is relevant traffic news. The *User* has *Terminals*, where the user receives relevant information. The terminal has a *Position*. An *Information provider* receives information from different sources and the Information provider selects the subscribed information and presents it to the *User* on his *Terminal* dependent on the current position of the terminal. This domain model does not state anything about the how this service is implemented.

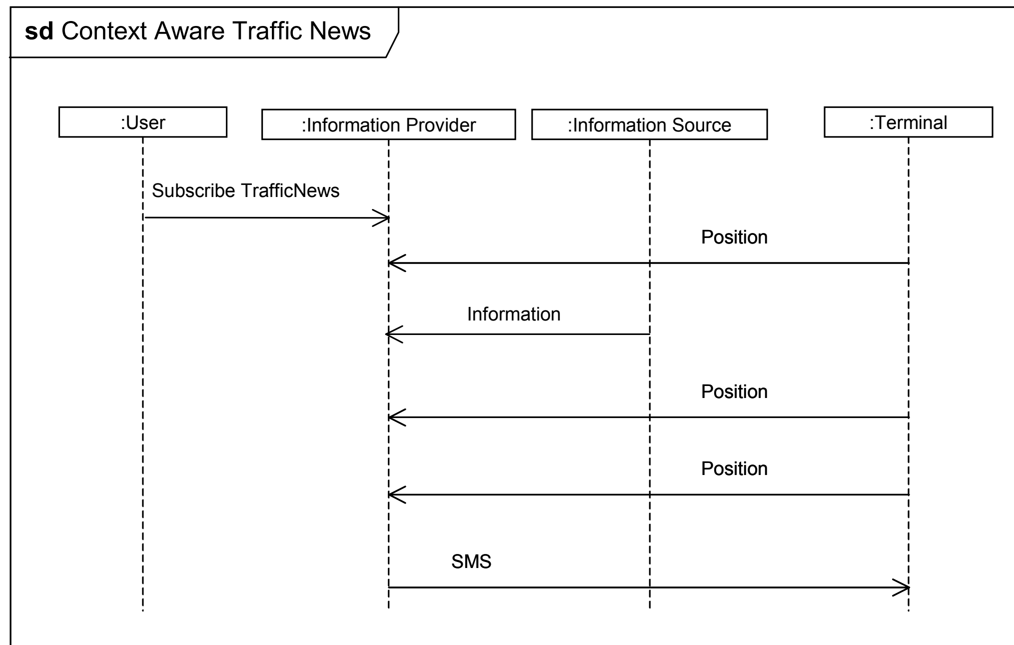

Figure 8-2 CATN service scenario

A typical scenario is shown in Figure 8-2, where the Information Provider sends only relevant information to the User, when the position of the terminal tells that this information is relevant for him.

8.2 Design of CAS

The service CATN that was introduced above is an example of a service that a Service provider would like to offer its customers. An application called Context Aware Services (CAS) is therefore proposed where a different kind of context aware services may be implemented. A design model of the CAS application is shown in Figure 8-3.

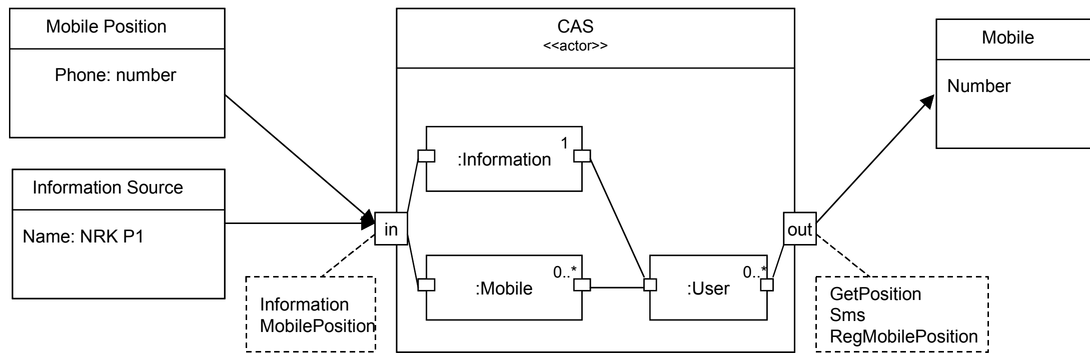

Figure 8-3 Context Aware Services

The CAS application communicates with a mobile positioning system, different information sources like NRK Traffic News, and with mobile phones through GSM systems. The different signals that the application receives and sends through its ports are shown in comments attached to the ports. The interaction diagram in Figure 8-4 shows the scenario described in Figure 8-2, where the application CAS is represented as one instance in the diagram.

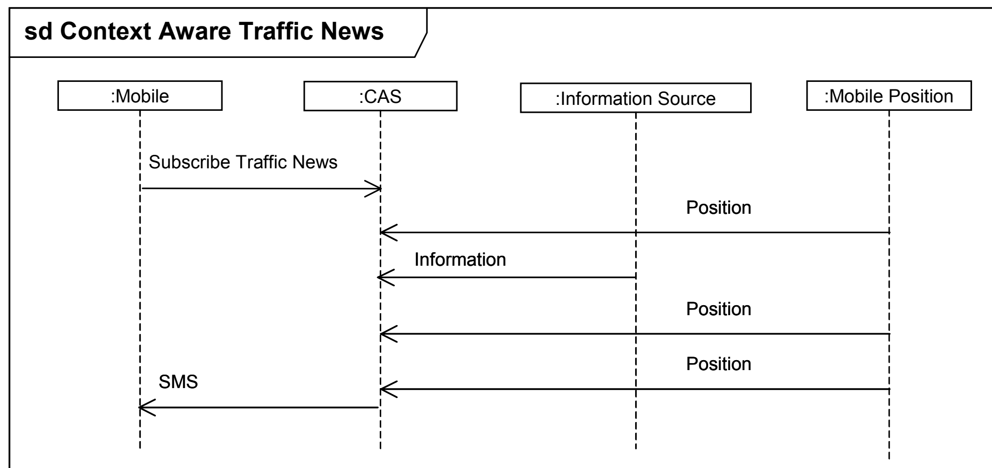

Figure 8-4 CAS - interaction with environments

The CAS application consists of one instance named *TrafficNews* of class *Information*, zero or more instances of classes *Mobile* and *User.*

The class User is shown in Figure 8-5 and it consists of one Position instance and zero or one instance of class *TrafficNews*. All these classes are stereotyped with *actor* like the class User shown in the figure below.

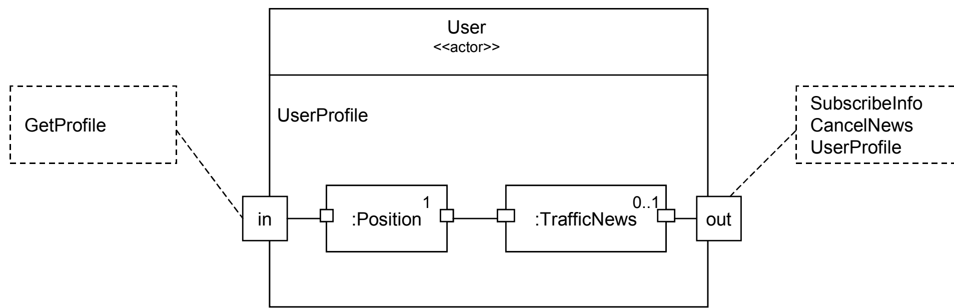

Figure 8-5 Class User

The sequence diagram for the TrafficNews service is shown in Figure 8-6. This diagram uses UML2.0 notation for sequence diagrams. The diagram describes two sets of alternatives and a loop.

- Loops describe a loop of sequences that are repeated until a condition is satisfied.
- Alternatives describe a set of possible sequences of interactions separated with a dotted line that may happen.

An interesting note to this diagram is that *TrafficNews* and *Position* is part of *User* (see Figure 8-5) and these instances therefore could be modeled in one lifeline, which will have simplified the diagram. A new sequence diagram for *User* would have shown the internal interaction of *User*.

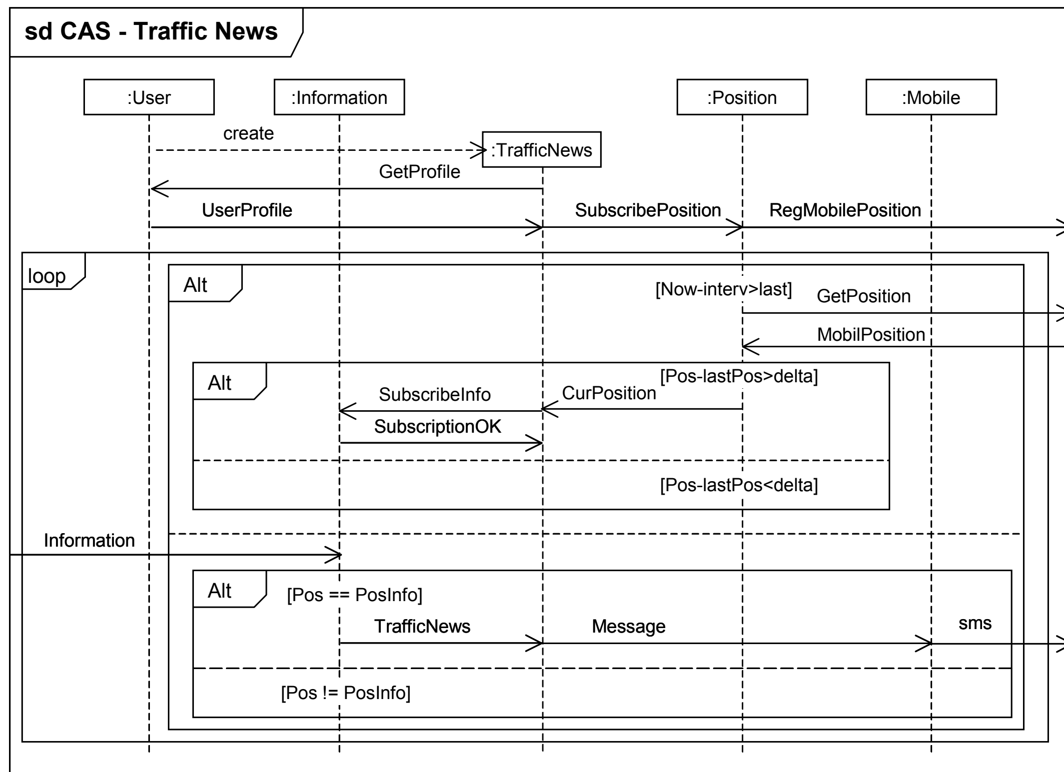

Figure 8-6 Sequence diagram of CAS application

The class *TrafficNews* is shown in Figure 8-7. It inherits the class Service, which is common for all CAS services. The class Service specifies common properties and behavior such as *UserProfile* and the communication with the User to obtain the user profile. The class *TrafficNews* adds the service specific attributes and behavior including signals.

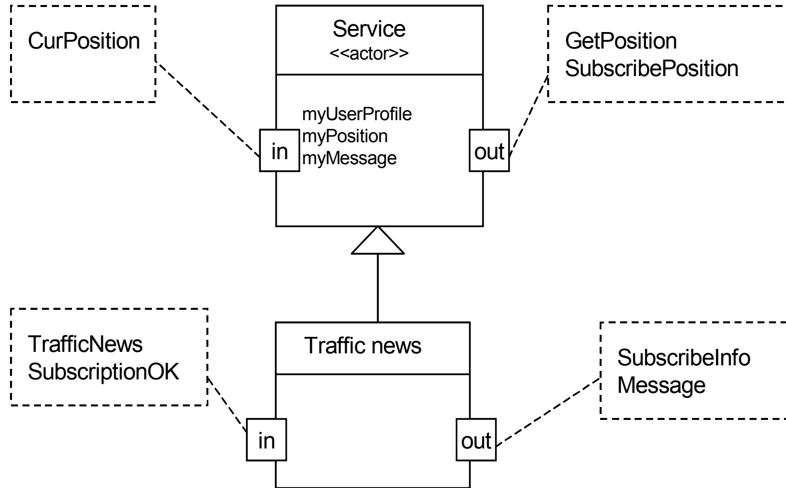

Figure 8-7 Class TrafficNews extends class Service

The behavior of class *TrafficNews* is modeled as a state machine as shown in Figure 8-8. The state machine of TrafficNews extends the behavior of the state machine of the super class. The transition caused by reception of the event *UserProfile* in state *Idle* is redefined in the state diagram for class *TrafficNews.* An output signal *SubscribePosition* is sent and the next state is set to *Waiting.*

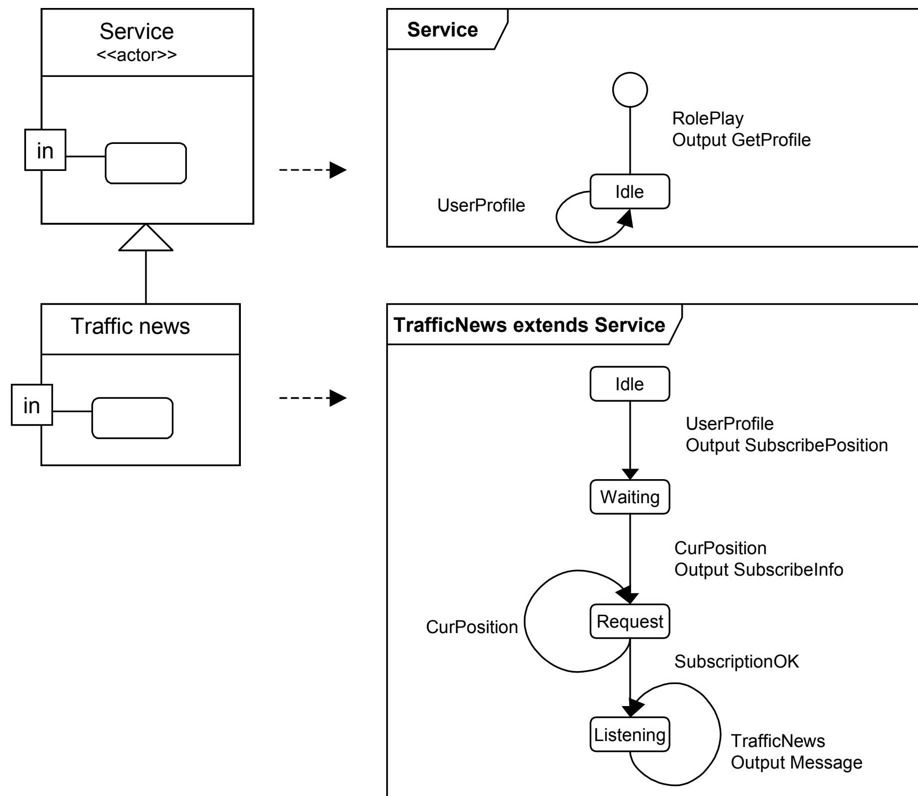

Figure 8-8 Behavior of class TrafficNews

8.3 Implementation of CAS

The CAS application is implemented by extending the classes defined in the EJBActorFrame Java package. In the figure the classes that must be implemented for each actor are marked in grey. The Figure 7-3 on page 65 describes how different parts of actors are mapped to these classes. The UML class *TrafficNews* and its super class Service are used as an example of how each Java class is implemented.

The UML class TrafficNews inherits the class Service. The class TrafficNews does not add new properties or state data. It is only the state machine that is extended. Therefore the classes that implement the state data as entity beans are omitted as illustrated in Figure 8-9.

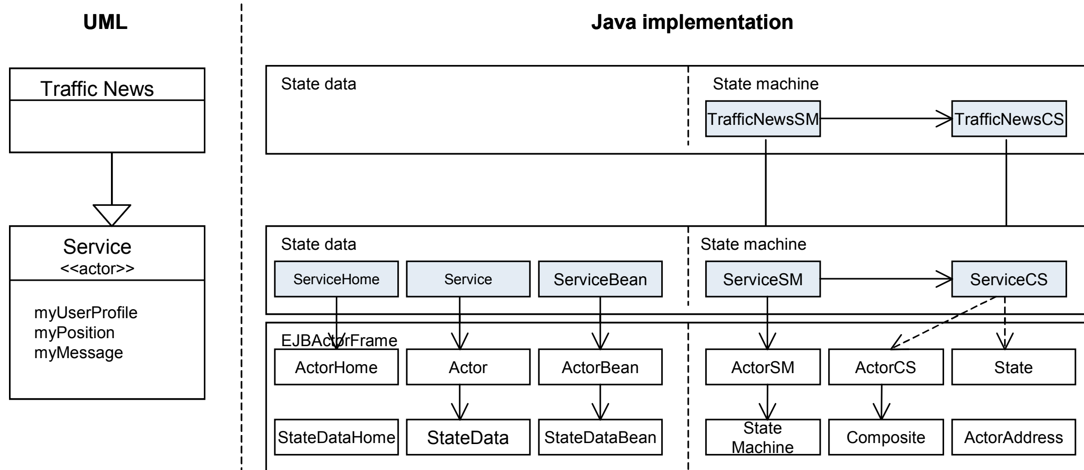

Figure 8-9 Implementation of the UML class TrafficNews

8.3.1 Mapping of behavior of state machines

The behavior of the UML class *TrafficNews* is implemented in the class *TrafficNewCS*. It extends the class *ServiceCS* as illustrated in Figure 8-10. The figure describes how the different parts of the state chart of class *TrafficNews* are implemented. The method *execTrans* is called when this state machine receives a signal. The second statement in this method shows how the state machine of its super type Service is called. According to the UML2.0 a transition may be redefined, but it cannot be extended. That means that if the subclass accepts the signal, the super type class shall not be called. The JavaFrame pattern does the opposite by calling the super type first. If the super type accepts the signal, the sub type discards the signal. In this example the super type is called first, and if the sub type has defined the same signal, the transition in the subtype is also executed. The semantic is that the subtype may also extend the transition and it allows new target state to be defined. This solution ensures that the behavior defined in the super type will be executed, but it allows adding behavior in the sub type. In this case an output of the signal *SubscribePosition* is added and next state is redefined to state *waiting*.

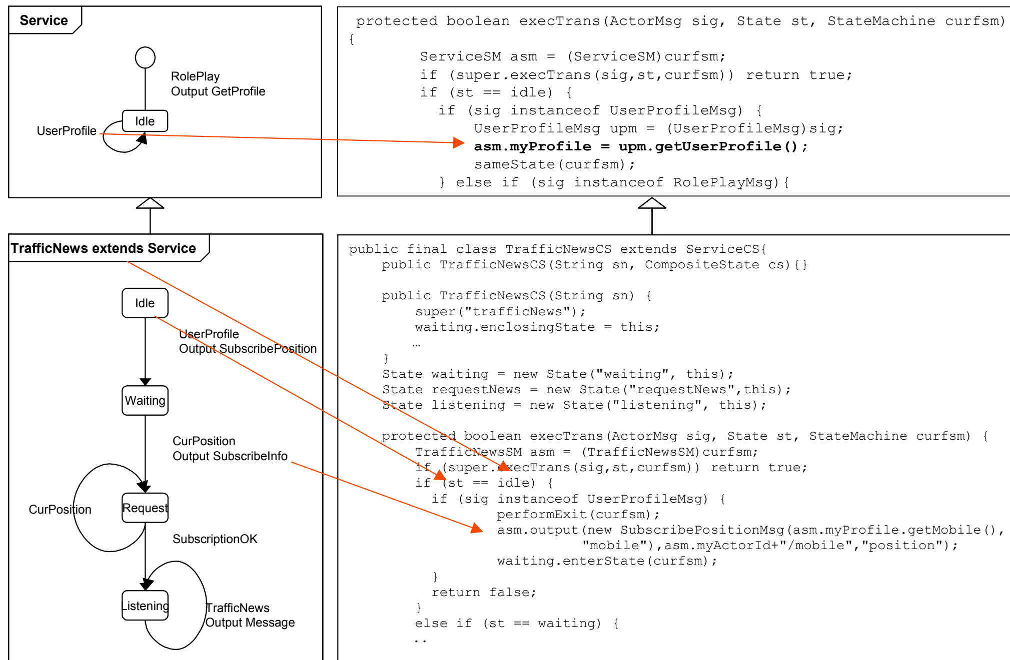

Figure 8-10 Implementation of behavior – class TrafficNewsCS

Figure 8-10 shows also how state data is accessed in a transition. Transition for class *Service* for the signal *UserProfile* defines an assignment of *asm.myProfile* to the profile carried as data in the received message. *Asm* is a reference to current state machine, which contains the state data. This is according to the JavaFrame pattern. The variable *myProfile* was read from the entity bean before the *ExecTrans* was called and it is stored again after the transition.

8.3.2 Mapping of state data

The state data is defined in the sub type of class *StateMachine*. It is these variables that are accessed during a transition. In the proposed mapping the state data is also stored in entity beans to ensure persistent storage of data. The class *TrafficNews* does not define new data other than that which it inherits from its super type *Service*. *Service* specifies the remote interface, *ServiceHome*, and the *SeriviceBean* class defines the classes used for storage of the state data as entity beans. These classes are shown in Figure 8-11, and they are implemented according to the EJB standard for CMP beans.

```
public interface ServiceHome extends EJBHome {
    Service findByPrimaryKey(String key) throws FinderException, RemoteException;
    Service create(String myId, String stateId) throws CreateException, RemoteException;
    Collection findAll() throws FinderException, RemoteException;
}

public interface Service extends Actor {
    UserProfile getUserProfile() throws RemoteException;
    void setUserProfile(UserProfile userProfile) throws RemoteException;
    Position getPosition() throws RemoteException;
    void setPosition(Position position) throws RemoteException;
    String getMessage()throws RemoteException;
    void setMessage(String message) throws RemoteException;
}

public abstract class ServiceBean extends ActorBean {
    public abstract UserProfile getUserProfile();
    public abstract void setUserProfile(UserProfile userProfile);
    public abstract Position getPosition();
    public abstract void setPosition(Position position);
    public abstract String getMessage();
    public abstract void setMessage(String message);
}
```

Figure 8-11 Implementation of state data as CMP bean

```
public interface ServiceHome extends EJBHome {
    Service findByPrimaryKey(String key) throws FinderException, RemoteException;
    Service create(String myId, String stateId) throws CreateException, RemoteException;
    Collection findAll() throws FinderException, RemoteException;
}

public interface Service extends Actor {
    UserProfile getUserProfile() throws RemoteException;
    void setUserProfile(UserProfile userProfile) throws RemoteException;
    Position getPosition() throws RemoteException;
    void setPosition(Position position) throws RemoteException;
    String getMessage()throws RemoteException;
    void setMessage(String message) throws RemoteException;
}

public abstract class ServiceBean extends ActorBean {
    public abstract UserProfile getUserProfile();
    public abstract void setUserProfile(UserProfile userProfile);
    public abstract Position getPosition();
    public abstract void setPosition(Position position);
    public abstract String getMessage();
    public abstract void setMessage(String message);
}
```

Figure 8-11 Implementation of state data as CMP bean

The abstract methods listed in Table 3 on page 70 must be specified when new state data is added to a sub type of class Actor. The method *findEntityBean*, which is listed in Figure 8-12 calls the home object of the entity bean Service and returns a reference of type Service. The reference is used by the methods *getData* to retrieve state data and *storeData* to store data of the entity bean. These methods are shown in Figure 8-12. Both methods call the respective super type method, which reads and stores the inherited state data.

```
 protected StateData findEntityBean(EJBHome home, String myId)throws RemoteException{
        Service service = null;
        try {
            service = ((ServiceHome)home).findByPrimaryKey(myId);
        } catch (FinderException e) {
            service = null;
            e.printStackTrace();
        }
        return service;
    }

protected void getData(StateData smr) throws RemoteException{
        super.getData(smr);
        Service s = (Service)smr;
        myProfile = s.getUserProfile();
        myPosition = s.getPosition();
        myMessage = s.getMessage();
    }

 protected void storeData(StateData smr) {
        super.storeData(smr);
        Service ref = (Service) smr;
        try {
            // store local state data
            ref.setUserProfile(myProfile);
            ref.setPosition(myPosition);
            ref.setMessage(myMessage);
            // end of storing local state data
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
```

Figure 8-12 Storage of state data

Class *StateMachineSM* calls *getData* before a transition is executed. The state data stored in the *StateMachineSM* is updtated with the state data stored in the entity bean for that particular state machine instance. After the transition is executed the entity bean is updated with state data stored in the *StateMachineSM*.

8.3.3 Use of JMS and JNDI

JNDI is used to find JMS queues, entity beans and to read input variables defined in the deployment descriptors. Most of the code is implemented in the class *StateMachine*. The constructor of class *StateMachine* does most of the initialization as shown in Figure 8-13. This constructor is called from the sub type with the name of the actor type as a parameter. This may be changed to allow the constructor to read from an environment variable that defines the actor type as the statement commented as "todo" in the code. The name of entity beans, JMS queues and environment variables are defined in the deployment file and the container reads these names and stores them into JNDI.

```
public StateMachine(String actorType){
    currentState = null;
    currentMessage = null;
    currentStateId = null;
    saveQueue = new MailBox();
    myActorType = actorType;
    try {
       context = new InitialContext();
       queueConnectionFactory = (QueueConnectionFactory) context.lookup("ConnectionFactory");
       queueConnection = queueConnectionFactory.createQueueConnection();
       queueSession = queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
       message = queueSession.createObjectMessage();
       queueSender = queueSession.createSender(null);
       // todo myActorType = (String)context.lookup("env/actorType");
       System.out.println("ActorType read from JNDI is: " + myActorType);
       objectRef = context.lookup("ejb/"+ myActorType);
    } catch (NamingException e) {
       System.out.println("JNDI API lookup failed for queue connection: " + e.toString());
    } catch (JMSException e) {
       e.printStackTrace();
    }
}
```

Figure 8-13 Initialization of JMS and JNDI

8.3.4 Implementation of parts

The class *CAS* Figure 8-3 on page 77, which defines the CAS application consists of 3 parts: *Information*, *Terminal* and *User*. The class diagram for CAS defines the initial number of instances that shall be created when an instance of the containing class is created and the maximum number of instances that are allowed to be instantiated during the lifetime of the containing class. Figure 8-14 shows how this information is stored in a hash table of the containing class, which in this case, is the class *CasSM*. When an instance of class is created or when new instances of the inner parts are created, this information is used in management of the lifetime of the parts.

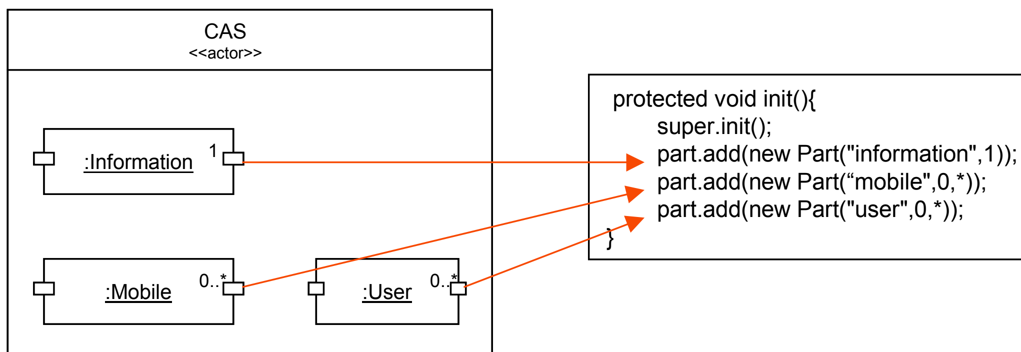

Figure 8-14 Implementation of parts

An actor manages its own inner parts and this information is stored in the message bean. The information is necessary when inner parts shall be deleted.

8.4 Deployment of CAS

Deployment descriptors are used for deployment of actors into EJB application servers. Figure 8-15 describes the deployment descriptor for an entity bean for the actor type *TrafficNews*. The local interfaces *local-home ServiceHome* and *local Service* optimize the access for other beans located in the same deployment file. The message bean and the entity bean that implement an actor should always be deployed into the same container so the message bean can therefore use the local interfaces. Calls to the entity bean are then implemented by the call-by-reference mechanism.

State data for the actor that shall be persistent is defined with the <cmp-field> tag. Because the EJB standard does not support inheritance of beans and descriptors, <cmp-fields> for all attributes that the actor inherits must be defined.

```
 <entity>
    <ejb-name>C</ejb-name>
    <local-home>se.ericsson.eto.norarc.diplom.ejb.trafficnews.ServiceHome</local-home>
    <local>se.ericsson.eto.norarc.diplom.ejb.trafficnews.Service</local>
    <ejb-class>se.ericsson.eto.norarc.diplom.ejb.trafficnews.TrafficNewsBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.String</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <cmp-field><field-name>myId</field-name></cmp-field>
    <cmp-field><field-name>currentStateId</field-name></cmp-field>
    <cmp-field><field-name>context</field-name></cmp-field>
    <cmp-field><field-name>myUserProfile</field-name></cmp-field>
    <cmp-field><field-name>myPosition</field-name></cmp-field>
    <cmp-field><field-name>myMessage</field-name></cmp-field>
    <primkey-field>myId</primkey-field>
</entity>c
```

Figure 8-15 Deployment of entity bean for state data of class TrafficNews

Figure 8-16 shows the deployment descriptor of the message bean for the actor type *TrafficNews*. The JMS destination type is set to Queue. The name of the JMS queue is defined in a container specific deployment file. In the JBOSS [29] application server the name of this deployment file is *jboss.xml*.

An environment entry called *actorType* is defined, which defines the name of the entity bean to be used in finding the entity bean. Class StateMachine uses this environment variable to access the local home interface of the entity bean that stores the state data.

```
<message-driven>
   <ejb-name>TrafficNews</ejb-name>
   <ejb-class> se.ericsson.eto.norarc.diplom.ejb.trafficnews.TrafficNewsSM</ejb-class>
   <transaction-type>Container</transaction-type>
      <message-driven-destination>
         <destination-type>javax.jms.Queue</destination-type>
      </message-driven-destination>
      <env-entry>
        <env-entry-name>actorType</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>TrafficNews</env-entry-value>
      </env-entry>
</message-driven>
```

Figure 8-16 Deployment of class TrafficNews as message bean

8.5 Summary

EJBActorFrame and EJBFrame packages have been implemented as part of this thesis work. The implementation of class *CompositeState* in the EJBFrame package is a modified version of the corresponding class in the JavaFrame package. Two state machines have been tested using the "pingpong" example included in the appendix. This test has proven that the concept proposed in this thesis works. JBOSS [29] has been used as an application server. All state machines were run in the same container, but they could easily be distributed in different containers.

The RoleRequest protocol of ActorFrame is only partly implemented. Some of the more advanced features such as play and remove of contained instances are left out. The ActorFrame protocol itself is an application built on the EJBFrame package.

The implementation of this example has shown that the pattern for implementation of behavior part of state machines from JavaFrame can be used also for EJBActorFrame. The only difference is a new constructor in the ActorCS class. This ensures that earlier implemented services can easily be moved to the EJB platform using the EJBActorFrame package.

Chapter 9 Discussion and conclusion

The solutions presented in this report are validated against the problem statements described in chapter 1.6. Each of these problem statements are presented in the following subchapters. Other issues such as performance and optimization of the presented solution are also discussed. This chapter also proposes some additional research topics, before the main conclusion is drawn.

9.1 Concurrency and asynchronous communication

The first problem stated in chapter 1.6 was *"How to achieve concurrency between state machines and asynchronous messaging between state machines?"* Concurrency does not mean in this context that the state machines have to execute in parallel. Concurrency between state machines is a conceptual way of organizing the software. Each state machine can be run independently and if it is desirable, be executed in parallel. Asynchronous communication is tightly coupled to independent execution of state machines. Synchronous calls between state machines will cause dependencies between them.

ActorFrame models, which are based on UML2.0, are state machines that communicate by sending signals to each other. Chapter 6 discussed ways to map state machines to EJB beans. A solution was selected which was a tradeoff between performance and resource usage. All instances of one state machine type were implemented by one message bean that received signals to these instances from one JMS queue. Asynchronous communication between state machines was achieved using this solution. All signals were sent through JMS queues, which resulted in state machines that were decoupled from each other both in time and space.

The solution described does not allow distribution of instances of the same state machine. All instances have to be executed in the same container. EJB allows message beans of the same type to receive signals from the same queue, but the container does not guarantee the order of the signals that are received by the message bean.

Signals received by the different message beans that are addressed to the same state machine instance, can cause conflicts in the access of state data of that state machine. There is also a possibility to use the selection mechanism in JMS to choose between signals. One message bean could receive signals for a sub set of the state machine instances. How to handle concurrent access of state data needs further investigation.

Another solution is to use bean managed transaction control. The entity bean that keeps the state data for an instance can be locked during the transition, which will prevent "dirty" reads from other message beans.

Performance issues have not been investigated. Use of JMS may cause significant performance reduction when the number of signals sent between state machines

increases. Tightly coupled state machines could, in these situations, be implemented in the same message bean and "lightweight" queues could be used instead of JMS queues for signaling between the state machines.

The conclusions that can be drawn from this work regarding problem statement number 1 are:

- *Concurrency between state machines can be achieved using entity beans combined with the quasi-parallel execution of beans supported by an EJB container.*
- *Asynchronous communication between state machines is achieved by using JMS queues combined with message beans.*

9.2 Aggregation of actors

The actor concept may contain other actors is a mechanism needed to able to be model complex structures and behavior. For instance in ServiceFrame, used by the AVANTEL research project, a user may participate in different services at the same time or the user can play multiple roles simultaneously. The part concept is also supported by UML2.0, and the mapping of parts was discussed in chapter 6.6. There is no counterpart to this concept in EJB, which is the basis for the second problem statement in chapter 1.6 "*How to map aggregation of actors?"*

Chapter 6.6 described how structural information, such as parts, are used both to control what is allowed to be done at run time and to manage the lifetime of actors such as deleting inner parts. The implementation of the part concept was shown in chapter 8.3.4 and this solution seemed adequate for most cases. The structural information mapped from the UML model was stored in the session bean, while the information about existing parts was stored as part of the state data of a state machine instance. The state data was stored in the corresponding entity bean. The parent object of a part uses this information when new inner parts shall be created or deleted.

The related name problem was discussed in chapter 6.7. The implemented solution, which used the instance names of the parent object recursively, ensured a uniqueness of part names in the scope of the system.

The solution presented in this thesis represents one possible solution to the problem of aggregation of actors. However, the solution has a drawback because the information about the parts is distributed to all state machines. Another solution could be to store this management information about created actors in a central database. This will enable a more efficient management of the lifetime of actors and the solution could also be used for implementation of a lookup service for executing actors.

The conclusion that can be drawn from this thesis work is that the implemented solution is one way to solve the problem with aggregation of actors (problem 2) on the EJB platform.

9.3 Persistent state machines

Reliability of applications is the underlying reason for the third problem stated in chapter 1.6 "*How can an implementation of state machines take advantage of the support of persistency in the J2EE platform?"*

The specification of EJB requires that an EJB container shall support persistency of entity beans. This is called Container Managed Persistency (CMP). The mapping solution proposed in chapter 7.6 used CMP beans to store the state data of state machines, ensuring persistency of state data.

However using this solution will decrease the performance of an application. For every signal the state machine receives, the state data is first read, the transition executed and then the state data is written back to the entity bean. This solution ensured backward compatibility with previously implemented applications based on ActorFrame. Another solution could be to pass the entity bean reference to the transition and then use get and set methods to access the state data during the transition. This solution will reduce the overhead of using the entity bean for storing state data considerably.

The proposed solution implemented all actors as persistent state machines. This is probably not necessary. A property of the actor stereotype could be used to define if an actor should provide persistent state data. If no persistency is needed, the state data could be stored as local data in the message bean.

If applications require that state data should be persistent, this must be provided regardless of the type of implementation that is used. In these situations the persistent state machines could be implemented as proposed in this thesis without any added performance reduction relatively to traditional implementations.

The conclusion is that the implemented solution that used container managed persistency types of entity beans for storage of state data, is a solution to the problem of how to achieve persistent state machines on the J2EE platform.

9.4 Actor modeling using UML profiles for EJB

In chapter 5 the concept oriented modeling approach was presented. ActorFrame supports this approach to modeling. The actor concept is used in modeling of services. Reuse of models is an import issue, which is the reason for the last problem statement "*Can existing UML profiles for EJB be used for making ActorFrame models that survive changes in the implementation platforms?*"

The UML profile for Enterprise Java Beans proposed by the Java community was described in chapter 5.3. This profile represented tags that were used to stereotype the UML model with platform specific tags.

How the actor concepts are expressed in the model is important. Chapter 5.4 showed that the UML profile for EJB could not be used to express the actor concept in a beneficial way. It was augmented for having an own UML profile for ActorFrame modeling. An example of such a profile was presented in chapter 7.2 and used in the PIM model for the CAS application presented in chapter 8.

An important argument for not using an UML profile for EJB is that a change of middleware platform may affect on the PIM model as shown in Figure 9-1. The reason for this was discussed in chapter 5.4

The PIM model represents the implementation independent description of the functionality of an application. The goal is that this model can be reused if the underlying technologies are changed. An interesting question is how a change in the middle platform will affect the different parts in Figure 9-1.

The actor model, which is a PIM model, can be directly implemented following the mapping rules for the actor to the specific platform. The EJBActorFrame described in chapter 7, contains the Java classes that represent the ActorFrame concepts. The ActorFrame model was implemented by extending these classes according to the guidelines used in implementation of the example in chapter 8.

The UML model was first translated to a PSM model using the UML profile for EJB described in chapter 5.3. The PSM model can then be mapped to the corresponding classes in the EJB Java package.

In Figure 9-1 the parts that are affected by the change in the platform from EJB to .NET are marked in grey. The major difference is that in the concept-oriented approach the PIM model is unaffected by the change of platform, while in the MDA approach both the PIM and PSM models are affected. Obviously the PSM model is affected because the stereotypes are different in EJB and .NET.

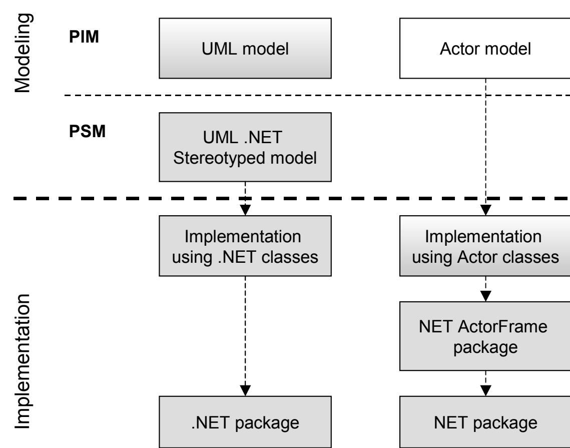

Figure 9-1 Change of middleware effect on the different approaches

The UML model may also be affected by a change of platform. The reason for this was presented in chapter 5.4. Chapter 5.4 illustrated that the PIM model must reflect the underlying concepts so it is possible for a translator to map actor classes to a corresponding stereotype defined in the UML profile for the new platform. If the platform does not support the same concepts as the old platform, this has to be reflected in the PIM model. For instance, if the new platform has support for persistent state machines, then an actor can be represented by one class rather than two classes as shown in Figure 5-4.

In the concept oriented approach the PIM model is tagged with an actor stereotype. The translator recognizes the actor stereotype and can map the UML class to a specific platform provided that the information exists in the ActorFrame model.

If a manual translation is used to map the ActorFrame model, it is desirable that a great deal of the implementation remains unaffected by a platform change. The *.NET*

ActorFrame package offers the same classes as the ActorFrame. However this is not always possible, as the experience from making the *EJBActorFrame* has shown. Some new classes had to be implemented for mapping of state data to entity beans.

The conclusion is therefore that a UML profile for EJB cannot be used to make ActorFrame models that are not affected by a change of middleware platform. An UML profile for ActorFrame, which can be used to make ActorFrame models that survive changes in implementation platform, was therefore proposed.

9.5 Other issues

Entity beans have been used to achieve persistency of state data. Persistency causes performance reduction because at each transition the state data in principle has to be stored again in a database. It is not always the case that state data needs to be persistent. For instance in the example presented in chapter 8, the TrafficNews actor gets parts of its state data from the User actor and this data can easily be reproduced again from the User. In such cases the state machines could be implemented without use of entity beans. To achieve improvement of performance, state data could be stored in a simple database as part of the state machine. The class StateMachine could implement a hash table where the state data for all instances of that type is stored. The state data has to then be implemented as an own class, which can be extended by subclasses of the state machine. This optimization could be specified by an attribute *isPersistent* of the actor stereotype.

Another optimization is to implement many state machines in one message bean as described in chapter 6.3.1, to avoid the overhead of communicating through JMS queues. This would improve performance especially in situations where there are many interactions between a set of state machines. This solution combined with the optimization regarding persistency of state data, will reduce the overhead considerably.

Inheritance is a powerful mechanism and this technique is used both in ActorFrame modeling and in the implementation of EJBActorFrame as shown in Figure 8-7 and Figure 7-2. Inheritance of an actor class requires that if the subtype adds new attributes to the state data, two interfaces and one class must be defined and used to implement the entity beans that keep the state data. Interfaces and classes can be extended by using ordinary inheritance mechanisms in Java. EJB does not support inheritance of the bean concept. The deployment descriptors of entity beans must completely define all fields including those inherited from the super type. Changes in the super type will therefore cause the interfaces and classes of all subtypes to be changed. This is a major drawback when implementation is done manually.

As pointed out in the summary of chapter 8, services implemented previously using the ActorFrame package can easily be moved to the J2EE platform by changing to the EJBActorFrame package, with only a few changes in the Java code. Most of the changes are hidden in the framework classes.

9.6 Future work

J2EE consists of a large set of technologies and this study has studied many of them. However, some issues have not been studied in detail. These issues have been noted for further study and are summarized below:

- Integration of EJBActorFrame with web services needs to be done. Although J2EE is integrated with web services, it does not have transparent support for asynchronous messaging. How this should be handled is subject for further study.
- An important issue is how to extend web services with support for asynchronous communication. There is some interesting work going on in Nielsen [48], who is proposing asynchronous communication between web services. They have also introduced the concept of SOAP routers, which seems promising. This approach is very much inline with the ServiceFrame concept and should therefore be investigated.
- Inheritance of behavior is rather immaturely understood in UML. The solution selected in UML2.0 on how to extend a state machine creates problems when UML2.0 is used to describe frameworks. The approach selected here for the implementation of state machines is different from UML2.0, but it makes inheritance of classes in a framework easier. Definition of a behavior concept that is also beneficial for framework design should be investigated further
- Inheritance of EJB is not supported and is an important issue. The component concept in UML is not studied during this thesis, but the concept looks interesting and it may be a way to improve the EJB component technology.
- Optimization of EJBActorFrame is important if this solution is to be used in a commercial setting. Some improvements have been presented here, others should also be investigated based on a better understanding on how the container concepts work. For instance, can more message beans be used to run the same state machine but for different instances. This becomes an important issue if high capacity is needed in the application server.
- This thesis has not studied in detail the transaction service in EJB. How to utilize transaction control should be studied and an adaptation in EJBActorFrame should probably be done.
- Timers are important in ActorFrame modeling to handle for instance timeouts where actors are waiting for a response from external resources. Timers are supported in ActorFrame. In the latest version of EJB a timer concept is specified. A study should be conducted to find out if this timer concept could be used to implement a timer functions in EJBActorFrame.
- Security issues have not been taken into consideration in this thesis work, but should be studied.

9.7 Conclusion

In this thesis the problem of mapping of ActorFrame models to the J2EE platform has been discussed. The thesis has proposed mapping solutions addressing the problems

stated in chapter 1.6. A concrete mapping solution has been proposed and a Java framework for implementation of actors has been developed. A concrete example has also been developed using this framework and part of the example was executed on a J2EE application server and the result has been evaluated and discussed.

The main conclusion is that ActorFrame models can be implemented on J2EE application servers in a beneficial way. J2EE has technologies that support asynchronous communication and it is possible to combine this communication style with implementation of persistent state machines. It is also possible to map the structural concepts as parts, connectors and gates to this platform in such a way that the implementation also benefits from these concepts.

This study has also shown that the UML profile for EJB from the Java community does not support the concept oriented approach that ActorFrame is founded on. This profile caused the need for an extra platform dependent model to be defined. This approach, which is propagated by OMG, is not robust against change of underlying platforms. A UML profile for ActorFrame, which enables a concepts-oriented approach, was proposed and the implementation of the CAS example has shown that this approach is more robust in regard to change of platforms.

The J2EE platform gives additional benefits compared to the current implementation of ActorFrame:

- Integration of emerging technologies such as Web services
- Management support such as deployment of actors "on the fly"
- Flexible distributions of actors
- Scalable platforms
- Transactional services
- Persistent state machines

However this thesis work has experienced that

- Application servers for J2EE are complex requiring detailed competence on application servers
- Software development tools are not very well integrated with the J2EE platforms
- The proposed solution may cause performance problems for applications.

These issues have to be taken into consideration when a decision shall be taken about using J2EE based application servers for implementation of ActorFrame models.

The thesis work has also shown that many mapping solutions exist and not all mapping issues have been elaborated. However this work forms a basis for further work in this area with the goal to be able to design frameworks for development of services, which can be deployed in the future service network.

Chapter 10 Abbreviations

ACID - Atomicity, Consistency, Isolation and Durability
CORBA – Common Object Request Broker Architecture
CS – Composite State
SM – State Machine
EJB – Enterprise Java Beans
IIOP - Internet Inter-Orb Protocol
J2EE – Java 2 Enterprise Edition
J2SE - Java 2 Standard Edition
JMS – Java Messaging Service
JNDI – Java Naming Directory Interface
JVM – Java Virtual Machine
MDK – Modeling Development Kit
MSC – Message Sequence Chart
RMI – Remote Method Invocation
RMI/IIOP – Remote Method Invocation over Internet Inter-Orb Protocol
RPC – Remote Procedure Call
SDL - Specification and Description Language
SOA – Service-Oriented Architecture
SOAP – Simple Object Access Protocol
UDDI - Universal Description, Discovery, and Integration
UML – Unified Modeling Language
UMTS – Universal Mobile Telecommunications System
WAP – Wireless Application Protocol
WSDL – Web Services Description Language
XML – Extensible Markup Language
API – Application Programming Interface
JAX-RPC – Java XML based Remote Procedure Call
MDA – Modeling Driven Approach
PIM – Platform Independent Models - used in MDA terminology
PSM – Platform Specific Models – used in MDA terminology
ALIN – Application Layer Internet working
MDA – Model Driven Architecture
SMTP – Simple Mail Transfer Protocol
HTP - Hyper Text Protocol
FTP – File Transfer Protocol
HLR – Hosting Location Register
AAA – Authentication, Authorization, and Accounting
JDBC – Java Data Base Connectivity
CMP – Container Managed Persistency
BMP – Bean Managed Persistency
MOM – Message Oriented Middleware
DNS – Domain Name Server

UMTS – Universal Mobile Telecommunications System
GSM – Global System for Mobile communication
ICT – Information and communication Technology
OSA – Open Service Access
SCS – Service Capability Server
JMS – Java Messaging System
3GPP – 3 Generation Partner Program

Chapter 11 References

1. Bræk, Rolf, Husa, Knut Eilif and Melby, Geir. *ServiceFrame Whitepaper, draft 1.9.2001,* Ericsson NorARC, 2001.
2. Haugen, Øystein and Møller-Pedersen, Birger. *JavaFrame: Framework for Java-enabled modelling*, ECSE2000, Ericsson NorARC, Stockholm, 2000.
3. Husa, Knut Eilif. *Serviceframe Software Architecture Document, Initial Version 29/01/02,* Ericsson NorARC, 2002.
4. Sun Microsystems, Inc. *Java Remote Method Invocation Specification,* Available from: http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC.
5. Couloris, George, Dollimore, Jean and Kindberg, Tim. *Distributed Systems: Concepts and Design (*3rd Edition), Addison-Wesley, 2001.
6. Winer, Dave. *XML-RPC specification,* Available from: http://www.xmlrpc.com/spec
7. W3C. *Simple Object Access Protocol (SOAP) 1.1*, 2000, Available from: http://www.w3.org/TR/SOAP/
8. Haugen, Øystein. *JavaFrame 2.5 Modeling Guidelines*, Ericsson NorARC, 2001.
9. Microsoft's .Net web page. Available at: http://www.microsoft.com/net/defined/whatis.asp.
10. The Java™ Messaging Service web page, Documentation available from: http://java.sun.com/products/jms/
11. The Java™ 2 Standard Edition, release 1.4 web page, Documentation available from: http://java.sun.com/j2se/1.4/.
12. The JavaBeans specification. Available from: http://java.sun.com/products/javabeans/docs/
13. JSR 26 UML/EJB Mapping Specification; http://salmosa.kaist.ac.kr/~course/DrBae/cs650_2001/LectureNotes/UMLProfileForEJB.pdf
14. The J2EE Tutorial for the SUN ONE Platform; http://java.sun.com/j2ee/1.3/docs/tutorial/doc/index.html
15. EJB & JSP – Java On The Edge; Lou Marco; METT Books; ISBN: 0-7645-4802-06
16. Developing Java Enterprise Application; Stephen Asbury, Scott R. Weiner; Wiley; Second edition; ISBN: 0-471-40593-0
17. SOAP 1.1 specification, available from http://www.w3.org/
18. UML 2.0 Superstructure Proposal (2nd revision) January 2003; U2 Partners; http://www.u2-partners.org/artifacts.htm
19. U2-partners home page: http://www.u2-partners.org

20. SDL - Specification and Description Language, CCITT recommendation Z100;
21. MSC – Message Sequence Chart, CCITT recommendation Z120
22. T.Reenskaug, P. Wold, and O. A. Lehne; Working With Objects. Manning: Prentice Hall, 1995.
23. Rolv Bræk et al; Quality by construction exemplified by TIMe – The Integrated Method; http://www.sintef.no/time/
24. Rolv Bræk; Using roles with types and objects for service development. IFIP International Conference on Intelligence in Networks, Smartnet 99, Bangkok, November
25. Haugen, Ø. and B. Møller-Pedersen. JavaFrame - Framework for Java-enabled modeling. in ECSE2000. 2000. Stockholm.
26. Lars Boman; Ericssons Service Network: a “melting pot” for creating and delivering mobile Internet services. Ericsson review 2/2000.
27. Robit Khare; Soap routing; the missing link; O’Railly Emerging Technologies Conference; UC Irvine & Know How, Inc
28. Bræk, Rolv;On Methodology Using the ITU-T Language and UML; Teletektonikk 4.2000.
29. JBOSS; J2EE Application Server; http://jboss.org/
30. Web-logic; BEA Systems; http://www.beasys.com/products/weblogic/
31. Jambala; Ericsson; http://www.ericsson.com/products/product_selector/JSCS_hpprod.shtml
32. Web services; W3C; http://www.w3.org/2002/ws/
33. J2EE – Java 2 Enterprise Edition; Sun; http://java.sun.com/j2ee/
34. AVANTEL – Research project; http://www.item.ntnu.no/avantel/
35. ARTS – Research project; http://www.item.ntnu.no/avantel/arts.html
36. OMG – Object Management group; http://www.omg.org/
37. CORBA - http://www.corba.org/
38. .NET; http://www.microsoft.com/net/
39. David Harel; Paper http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/OOStatecharts.pdf
40. OCL – Object Constraint Language; Part of the UML standard.
41. OMA – Open Mobile Alliance; homepage: http://www.openmobilealliance.org/
42. 3GPP – Third Generation Partner Ship; homepage: http://www.3gpp.org/
43. OMG; IDL - Interface Definition Language
44. Sun Java Community; UML Profile for EJB;

45. SISU - research project; http://www.sintef.no/units/informatics/projects/sisu/
46. UML; Object Management Group OMG; Specification found at http://www.omg.org/
47. W3C; SOAP 1.2; Proposal: http://www.w3.org/TR/soap12-part1/ and http://www.w3.org/TR/soap12-part2/
48. Nilsen; Microsoft Corporation; Presentation SOAP routing and Message Path Modeling; PDC 2001, October 22-26, 2001
49. TIMe – The Integrated Methodology; SISU project; http://www.sintef.no/time/
50. W3C – XML; Specification: http://www.w3.org/XML/

Chapter 12 List of figures

Figure 1-1 Next generation network 2
Figure 1-2 ServiceFrame [1] 3
Figure 2-1 Composition versus parts [18] 11
Figure 2-2 Class with internal structure [18] 12
Figure 2-3 Ports connected to classes [18] 12
Figure 2-4 Connectors and ports [18] 13
Figure 2-5 Definition of Exit / Entry points [18] 14
Figure 2-6 Use of Exit / Entry points [18] 15
Figure 2-7 Specialization by extension [18] 16
Figure 2-8 Specialization of state machines [18] 16
Figure 2-9 Defining UML extensions 17
Figure 2-10 Use of stereotypes 17
Figure 3-1 ServiceFrame - a model driven service development kit [1] 19
Figure 3-2 ServiceFrame layers [1] 20
Figure 3-3 Class Actor 21
Figure 3-4 RoleRequest protocol [1] 21
Figure 3-5 Multiple roles and actors 22
Figure 3-6 A simple service. 22
Figure 3-7 Play 23
Figure 3-8 JavaFrame classes [2] 23
Figure 4-1 EJB Architecture 29
Figure 4-2 Client view of enterprise beans 29
Figure 5-1 Three main aspects [28] 42
Figure 5-2 Realization versus concept tags 44
Figure 5-3 Stereotypes and tagged values for EnterpriseBean [44] 47
Figure 5-4 Use of UML profile for EJB for ActorFrame modeling 48
Figure 6-1 All state machines into one bean 53
Figure 6-2 One State Machine into on bean 54
Figure 6-3 State machines of same type into one bean 55
Figure 6-4 All state machines share same queue 57
Figure 6-5 One queue for each state machine 58
Figure 6-6 State machine of same type share one queue 59
Figure 6-7 Parts in UML2.0 60
Figure 6-8 Name scope for actors 61
Figure 7-1 Implementation of state machines 63
Figure 7-2 EJBActorFrame 64
Figure 7-3 Mapping of Actor class to EJBActorFrame classes 65
Figure 7-4 Profile for ActorFrame 66
Figure 7-5 Class Actor 66
Figure 7-6 Operation of an actor state machine 67
Figure 7-7 EJBActorFrame classes 69
Figure 7-8 State data classes 69

Figure 7-9 Dependency of the state hierarchy ... 71
Figure 7-10 Current state represented as a state id ... 72
Figure 7-11 JNDI names of actors ... 73
Figure 8-1 Domain model of the CATN service ... 75
Figure 8-2 CATN service scenario ... 76
Figure 8-3 Context Aware Services ... 77
Figure 8-4 CAS - interaction with environments ... 77
Figure 8-5 Class User ... 78
Figure 8-6 Sequence diagram of CAS application ... 78
Figure 8-7 Class TrafficNews extends class Service ... 79
Figure 8-8 Behavior of class TrafficNews ... 80
Figure 8-9 Implementation of the UML class TrafficNews ... 81
Figure 8-10 Implementation of behavior – class TrafficNewsCS ... 82
Figure 8-11 Implementation of state data as CMP bean ... 84
Figure 8-12 Storage of state data ... 85
Figure 8-13 Initialization of JMS and JNDI ... 86
Figure 8-14 Implementation of parts ... 86
Figure 8-15 Deployment of entity bean for state data of class TrafficNews ... 87
Figure 8-16 Deployment of class TrafficNews as message bean ... 88
Figure 9-1 Change of middleware effect on the different approaches ... 92