University of Oslo
Department of Informatics

Mapping of QoS-Enriched Models to a Generic Resource Model

Cand. Scient. Thesis

Espen Abrahamsen

18th May 2005

Acknowledgements

This thesis concludes my work for the Cand. Scient. degree at the Institute of Informatics, University of Oslo. The work has been done at SINTEF and Simula Research Laboratory.

I want to thank my supervisors, Jan Øyvind Aagedal and Arnor Solberg, for their great help and patience. I also want to thank my family and my friends for their great support during my work with this thesis.

Contents

Chapter 1

Introduction

1.1 Quality of Service and Model-Driven Development

Model-driven development (MDD) moves the focus from code to models. It has gained a lot of attention, especially in conjunction with software development for middleware platforms such as CORBA, Java 2 Enterprise Edition and Microsoft .NET.

In tradiitional, code-centric development, the creating models is typically merely a tool for sketching before writing the actual code. In model-driven development, the models are intended to become first-class entities, just as important as the code. The models are intended to be used throughout the software lifecycle.

Model Driven Architecture (MDA) [21] is an approach to model-driven development proposed by the Object Management Group, and uses the Unified Modeling Language [25] as default modeling language. Central in MDA are the concepts of platform-independent model (PIM) and the platform-specific model (PSM). A central goal is to support automated *model transformation* from PIM to PSM.

When modeling a PIM the developer should only have to worry about the business logic of the application, as opposed to platform-specific implementation details. The PIM can later be transformed into a PSM. This activity could be performed automatically (through predefined mappings) or with some degree of human intervention. Code generation from a PSM can also be considered a kind of model transformation, where the code is considered a target model. The generated code could be fully executable or could need refinement by the developer in order to run. Today, the UML does not include the necessary semantics to create fully executable models. This is especially due to limited support for expressing system behaviour.

Quality of Service (QoS) deals with qualitative properties of services provided by a system. Some qualities can be measured objectively, through quantitative measures (like latency and throughput), others can only be measured subjectively by humans (such as learnability). QoS requirements differs from the functional requirements of a system. QoS is not concerned with what functions a system can perform, but how well it performs these functions.

In [3] it is argued the capturing and handling of QoS requirements should be an essential part of the MDA. QoS concerns must be identified early in the development process, because they are difficult to manage after the system is designed and perhaps set into operation.

Considerable research effort is put into investigating how QoS could be managed by the middleware platform. In particular, component architectures are developed to support QoS management. The QoS-aware Component Architecture (QuA) platform [29] is one such platform developed as part of a currently ongoing research project. The QuA project looks into how a *service planner* can realize a certain level of QoS by selecting appropriate compositions and configurations of compoents to form services.

Resource availability is a low-level factor that helps determine what level of QoS the system is able to deliver. Thus, resources should be managed appropriately in order to adapt to the needs of the applications. A resource model provides an abstraction of the concrete resources in the system, and serves as a way to access and deal with these resources in a structured way.

We will look at how such a resource model can be incorporated in QoS-enabled, model-driven development. This extends the MDA paradigm to also include a runtime-level model.

A general definition of *resource* from Britannica dictionary is: *"A source of supply and support"*. We need to specialize this definition to support our needs. For example, it does not mention that resource must be limited. In order for QoS management to be meaningful, resource supply must be of limited nature. An unlimited resource would perform perfectly under any condition, and thus does not need to be managed.

It is not our intention to model all thinkable resources in the world, such as natural resources or economic resources. Instead, we deal only with resources usable within a computer system. The UML Profile for Schedulability, Performance and Time [24] defines a resource instance as *“a run-time entity that offers one or more services for which we need to express a measure of effectiveness or quality of service (QoS)”*. This definition implies that resources provide a limited level of Quality of Service, thus having a finite capacity. We use this definition as basis when dealing with resources.

1.2 Problem Statement

In this thesis we propose a method for handling Quality of Service concerns in a model-driven development context. An important aspect of QoS management is the management of limited resources. Our focus is on how resource-level QoS requirements can be treated in a model-driven development process, from design-time QoS specification to run-time resource management.

In order to find out whether our approach is feasible, we will investigate the following:

- How resource QoS requirements can be modeled at design-time in a way such that they can be considered later in the development process.
- How to transform the resource QoS requirements expressed in the model to a form that is manageable by the target platform.
- How a resource model can be designed and implemented on a target platform in order to supports mapping of resource QoS requirements.

We believe that QoS requirements for resources can be specified as part of the application model, and that these requirements can be automatically transformed and mapped to the resource model of the implementation platform. This will allow the identification and specification of QoS at design-time, and ease the implementation of QoS-related aspects later in the development process.

1.3 Thesis Overview

Chapter 2 summerizes the theoretical and technological foundation for the rest of this tesis.

In **Chapter 3** we look at how Quality of Service aspects can be treated in a model driven development context. We look at related work in this area, and discuss how the approach proposed in this thesis fit into this context.

In **Chapter 4** we define a generic resource model with support for model driven development and middleware-level resource management.

Chapter 5 presents an approach to model transformation concerning resource-level QoS requirements. The target model is a platform- specific model targeted towards the resource model.

In **Chapter 6** we demonstrate our model transformation approach and usage of the resource model by an application development case.

Chapter 7 is a discussion of the results and experiences gained from experimenting with the application case. We try to answer the addressed problems and claims based on this.

In **Chapter 8** we conclude our work by summarizing the results of this thesis.

Chapter 2

Background

2.1 Model Driven Development

Model-driven development is by many predicted to be the next big leap in the software development industry, moving the main focus from code to models. The Object Management Group (OMG) has proposed Model Driven Architecture (MDA) as their approach to model-driven development. It defines *model-driven* as "*providing a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification*" [21].

An important advantage of MDA is the separation of implementation details from business concerns [8]. By making business modeling a separate activity, we are able to make models independently of any particular platform. This way, the same model could be deployed on different platforms, thus better supporting changes in technology over time, such as the appearance of new platforms.

2.1.1 Modeling

Modeling is the activity of creating models [16]. A model can represent virtually any system, but we are mainly concerned with modeling information systems.

A model is in [32] defined as *a set of statements about some system under study*. It mentions two purposes a model may serve. The first purpose is a *description*, trying to describe a system already in existence, and the second is a *specification*, that states how the modeler wants the system to become prior to its existence. The latter kind of model is typical in software development.

In order to be understood, models must be created in a modeling language. Typically, models are presented a combination of drawings and text [21]. The Unified Modeling Language (UML) [25] is the default language in MDA, and a def facto standard for object-oriented modeling in general. Even though modeling of object-oriented systems is the most common application of UML, it is a general language that can support other modeling paradigms as well.

2.1.2 Metamodeling

A *metamodel* can generally be considered a model of a set of models. How to model models, on the other hand, is not self-explaining. According to [32], a metamodel makes statements about what can be expressed in the valid models of a certain modeling language. By this, they mean that metamodels describe how models can be constrcuted by defining the abstract syntax of their modeling language. This definition fits well with the definition in the Meta-Object Facility specification [9]: *A metamodel is an "abstract language" for describing different kinds of data; that is, a language without a concrete syntax or notation.*

The specification of UML [25] is such a metamodel, and thus, the model of all UML models.

Meta-Object Facility

The Meta-Object Facility (MOF) [9] from OMG contains a language for creating metamodels. The UML specification, for example, is defined with MOF. The modeling constructs in MOF are a subset of the static UML elements used in class diagrams. For example, MOF models may consist of classes, attributes, data types, packages, and so forth. In the UML metamodel, a MOF class will typically represent a UML element. Relations between the UML elements could be modeled using MOF associations. Even though the MOF elements do not have a specific graphical notation, a subset of the UML notation can be used to represent it. This approach has been used in the UML specification.

The MOF specification includes a separation of four *metalevels* for models:

M3 Meta-metamodels

M2 Metamodels

M1 Models

M0 Information

Meta-metamodels are models of metamodels, and can be seen as a language for creating metamodels. The metamodels define the abstract language of models. Models are specifications of systems. The *information* metalevel represents information in the system specified by systems, for example records in a database.

2.1.3 XMI

The XML-based Metadata Interchange (XMI) [26] is an exchange format for metadata. XMI documents are valid XML documents, and are therefore supported by existing XML tools. XMI documents can hold *any kind of metadata that can be expressed using the MOF specification, including both model and metamodel information* [26]. As well as MOF metamodels, the XMI format can be used to store any model of which MOF is the metamodel. A typical usage of XMI is the serialization of UML models for exchange between UML tools.

2.1.4 UML Profiles

Since UML is a general-purpose language, it may not be suitable for modeling within particular domains [19]. UML provides a set of light-weight extension mechanisms that can be applied without modifying the UML metamodel. UML profiles are in [6] described as a way of representing and structuring these UML extensions.

A profile is typically tailored towards a particular application domain or implementation platform, where the UML extensions reflect domain-specific or application-specific concepts. A profile is a grouping of UML extensions, including:

Stereotypes are attached to elements of the UML metamodel. When using these elements in a model, the modeler may apply the available stereotypes where appropriate. A stereotype is intended to give the element an extended meaning without redefining its original semantics. For instace, in a UML profile for relation databases, a table could be modeled as a class with stereotype *«Table»*.

Tagged values are attributes related to a stereotype. Its values can be set for each instance of the stereotype used in a model. The tagged value can be used to supply additional information about the stereotyped model element.

Constraints can be applied to UML profiles to constrain the usage of modeling elements.

UML profiles were first introduced in [6] as a set of guidelines for how to implement UML profiles. The upcoming UML 2.0 will have a more precise definition of profiles and mechanisms for how to specify them. However, we do not consider UML 2.0 in this work, as it is not yet finalized by the OMG, and the tool support at the time of writing is very limited.

Profiles are a light-weight alternative to defining a completely new domain-specific language (e.g., by defining a new MOF metamodel). Since the UML metamodel must be respected when defining profiles, one can still use the wide range of available UML tools. However, profiles are less flexible than defining a new language. Compatibility over flexibility is a typical trade-off when defining new modeling languages and choosing between a UML profile and a new metamodel [19].

In order to define a UML profile, the first step is to create a model of the element that comprise our platform or system, and the relationships between them [19]. This type of model is refered to as a *domain viewpoint* or *conceptual model*. In order to use these concepts in models, they must be represented by UML extensions such as stereotypes and tagged values. This relationship is defined in another model, often called the *UML viewpoint*. The purpose of this viewpoint is to relate the domain concepts with the UML notation.

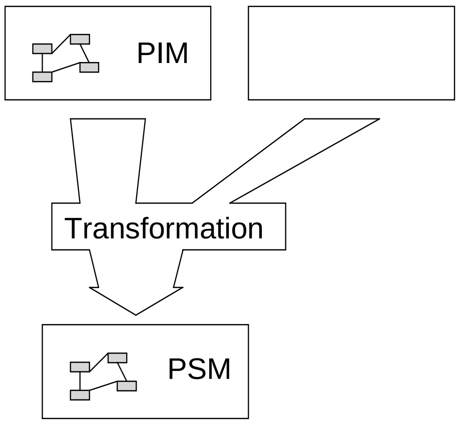
In [4] it is argued that UML profiles provide only content, and not form (linguistic relationships). Therefore, the domain viewpoint of UML profiles is not a *linguistic* metamodel like the metamodel of UML, but instead supplements UML models with additional information.

In [19] it is claimed that UML profiles have a central place in MDA. They can play an important role in describing the platform model and the transformation rules between models. Central in MDA is the transformation between PIMs and PSMs, and UML profiles are very well suited to represent platform-specific concepts.

2.1.5 Model Transformation

The MDA Guide [21] defines model transformation as *"the process of converting one model to another model of the same system"*. In MDA, the source model of the transformation is typically a PIM, and the target model is a PSM. Additional information may also be available to help make decisions about how to perform the transformations.

Figure 2.1 Model Transformations in MDA



An example of a model transformation taken from [21] is shown in Figure 2.1. It shows a transformation from a PIM to a PSM. The empty box represents additional information that may be provided in order to make decisions while performing the transformation.

In [33] *model transformation* is defined as *"automated processes that take one or more source models as input and produce one or more target models as output, while following a set of transformation rules"*. This definition assumes that model transformations must be automatic. However, the MDA also allows manual or semi-manual transformations of models [21].

MOF 2.0 Query/Views/Transformations is an effort by OMG to standardize how to specify model transformations by defining a specialized transformation language for MOF models. Transformations defined in

QVT are based on mappings between MOF metamodel elements. Thus, a transformation from a source model to a target model is represented in QVT as a transformation between the two metamodels. Both metamodels are expressed using MOF.

2.1.6 Code Generation

In [12] it is argued that programmers typically need to achieve the same functionality in different places, and building code generators is an effective tool against this repetitive typing. Writing a code generator means writing code that writes code.

A special kind of code generation is *model-to-code transformation*, where code is generated based on a source model. In an MDA context, the PSM is a typical source model for generating platform-specific code. Code may be generated based on parts of, or a complete model.

The OMG has issued a request for proposal for a *MOF Model to Text Transformation Language* [22]. It aims to provide a language for transforming MOF models (such as UML models) to text. One of the purposes mentioned is the transformation of PIMs and PSMs to code, XMI, human-readable UML format, documentation, and HTML. Since the QVT standard *omg:qvt* does not address model-to-code transformations, this RFP seeks to fill this gap.

It is argued in [22] that XSLT is not a suitable transformation language for MOF models. Even thought it supports transforming an XMI model, the transformation code will be complex and difficult to maintain.

2.2 Quality of Service

Quality of Service (QoS) is in *Open Distributed Processing - Reference Model - Quality of Service* [15] defined as: *"a general term for an abstraction covering aspects of the non-functional behaviour of a system."* Functional behaviour is the actual functionality offered by the system, while non-functional behaviour describes *how well* this functionality is carried out *Performance* and *latency* are examples of non-functional aspects.

A *QoS characteristic* is presented as a basic entity in [15], expressing *various things with respect to one or more values associated with the* *characteristic*. It represents an aspect of QoS as opposed to a measurement.

A *QoS requirement* is an expression that involves one or more QoS characteristics and one or more values, where the value(s) say something about the level of QoS required.

2.2.1 Classification of QoS Characteristics

There are several dimensions that can be used to classify QoS requirements. Abstraction level is one such dimension. The lowest levels of abstraction are closest to the physical implementation of the system, such as requirements related to physical resources. These requirements can be mapped directly to the physical properties of the system. Thus, the realization of these requirements is straight-forward, given that the resources provide sufficient capacity.

More abstract QoS requirements make statements about the behaviour of software entities (such as applications or services) rather than resources. The realization of such requirements is less trivial, and involves more complex reasoning. Often, these kinds of requirements implicitly lead to lower-level requirements (e.g. resource QoS requirements). *Application QoS* is a common name for non-functional requirements of applcations. Application QoS requirements are usually expressed objectively, involving statements that are measurable.

User QoS, or *perceived QoS*, involves even more abstract QoS requirements. These are typically statements about how humans should perceive the quality of a service (for example, *"the video should have a sharp picture with good sound quality"*). This kind of requirement is easy to understand for the non-technical user, but does not map easily to the implementation that is responsible for realizing it.

QoS requirements are often also classified into different domains. Typical domains that deal with QoS requirements are *multimedia*, *real-time* systems, and *networking*.

2.2.2 QoS Specification

In order to handle QoS concerns in a system, the QoS must be specified. Specification can be done at different domains and abstraction levels, and can be expressed in different specification languages. In [16] it is argued that QoS specification is important in several development phases, including analysis and design.

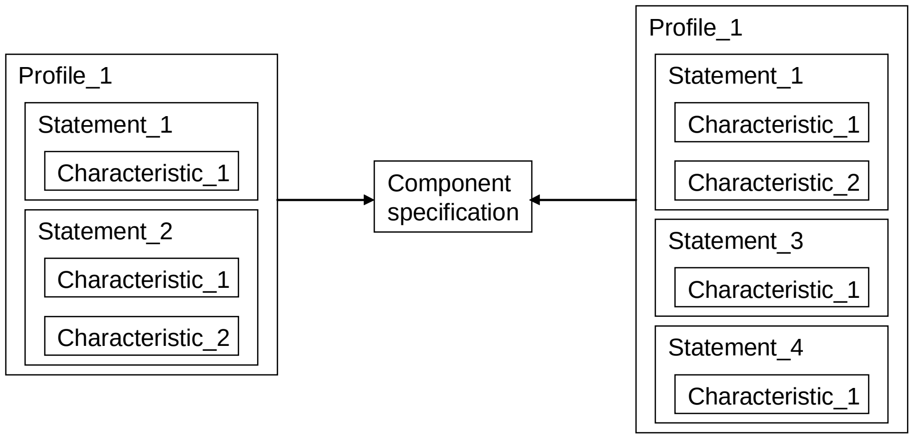
By QoS specification we mean the activity of making statements about QoS in a formal way using a specification language. If we allowed informal QoS specification (e.g. using natural language), a computer would not be able to extract its meaning.

In the following subsections we look at some languages that support QoS specification.

CQML

The Component Quality Modeling Language (CQML) is specified in the Ph.D. thesis by Aagedal [16]. It is a generic QoS specification language, not tied to phase of the software life-cycle or any specific purpose. It is a declarative language, and does not express any beavioural features.

Figure 2.2 CQML Overview



An overview of CQML from [16] is shown in Figure 2.2 CQML adopts some basic concepts, such as *QoS characteristic*, from the ISO QoS framework [13]. QoS characteristics (*QoS characteristics* in CQML) are user-defined types that define how a certain type of QoS is measured. These QoS characteristics can be as simple or complex as the modeler wants, and may involve both simple numerical ranges or statistical properties. Figure 2.2 shows how QoS statements can include one or more QoS characteristics, and QoS statements can be grouped into QoS profiles. A profile can further be associated with a component specification.

In order to express QoS requirements, a *QoS statement* may refer to one or more *QoS characteristics* and put a constrain on the range of valid values.

QoS profiles are used to relate QoS characteristics to components. A profile may incorporate multiple QoS statements.

QoS categories are a mechanism to group QoS characteristics, QoS statements and QoS profiles based on their common properties and domain. An example of a QoS category is *timeliness*, incoroprating *output* and *delay*.

UML Profi le for QoS

A specification called *UML Profile for Quality of Service and Fault Tolerance Characteristics and Mechanisms* is currently under finalization by the OMG. We use a working document of September 2004 [23] in this thesis. The specification provides a framework for the description of QoS requirements and properties, as well as a set of UML extensions to represent these concepts in models. The QoS framework defines the metamodel of the QoS modeling language that is supported by the UML profile.

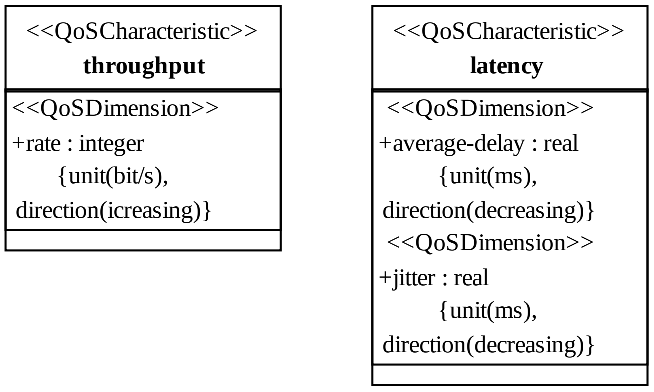
We do not consider the fault tolerance aspects of the UML profile, as this is beyond the scope of this thesis.

The UML profile for QoS supports applying non-functional requirements to UML models. The QoS elements are expressed orthogonally to the functional elements, so that the functional specification of the system will not be affected by non-functional concerns. The UML profile supports both the specification of QoS characteristics (i.e., *how* QoS can be measured) and QoS constraints (i.e., *what* QoS is required).

QoSCharacteristics are model elements that represent quantifiable aspects of QoS. Each characteristic is quantified by one or more *QoSDimensions*. A QoS characteristic is defined independently of the values it quantifies. In order to describe values, a QoS characteristic has one or more QoS values, modeled as stereotyped attributes. Two examples of QoS characteristics are shown in Figure 3.2.

A *QoSDimension* is represented as a stereotyped UML attriute. Its basic type for measurement maps to the UML *data types*, such as *Integer*, *Real*, and *Enumeration*. As well as this value, QoS characteristics are described by some tagged values:

Figure 2.3 QoS Characteristics



direction expresses whether higher values represent higher quality than lower values (increasing), or higher values represent lower quality (decreasing).

unit represents the measuring unit of the QoS dimension.

statisticalQualifier is used to express statistical that the QoS dimension represents statistical properties, e.g., *mean*, *maximum* and *minimum*.

The *latency* characteristic in Fiugre 3.2 is described by the dimension *rate*, with an *increasing* direction and is measured in *bit/s*.

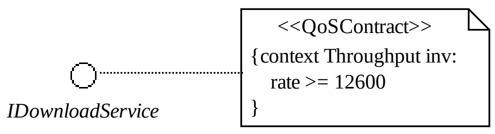
QoS characteristics can be grouped into QoS categories, which are modeled as UML packages with stereotype *«QoSCategory»*. A QoS category typically contains QoS characteristics of the same domain, and describe different parts of the same aspects. For example, the QoS catalogue presented in [23] contains categories like *Performance* and *Dependability*.

A *QoSValue* expresses a specific quantification of a QoS characteristic, and is considered an *instance* of the characteristic. It contains a fixed value defined within the range of allowed values according to the QoS dimensions of the characteristic.

*QoSConstraint*s limit the allowed values of one or more QoS characteristics [23]. It is declared as an abstract metaclass, and has the subclasses

QoSRequired, *QoSOffered* and *QoSContract*. These can be modeled as stereotyped UML constraints with the respective stereotype names, or alternatively, a stereotyped UML dependency to an instance element stereotyped *«QoSValue»*. A QoS constraint must be defined within a certain *QoSContext*. QoS characteristics define QoS context for expressions involving its QoS dimension. QoS context may also involve more than a single QoS characteristic, and be based on other QoS characteristics and/or other QoS contexts. An example QoS constraint expressed in OCL is depicted in Figure 2.4, connected with an interface element.

Figure 2.4 QoS Constraint



```
<<QoSContract>>
{context Throughput inv:
   rate >= 12600
}
```

This OCL expression could for example be defined within a constraint with the stereotype *«QoSContract»* connected with a model element which it describes. The QoS context of the expression is the *throughput* QoS characteristic (from Figure 3.2), and the expression limits the allowed values of *rate*.

2.3 Resource Modeling

By *resource modeling* we mean the creation of resource models. A resource model is a model of reources. In the QoS specification of RM-ODP [15], a resource model is said to make the computing and communication resources explicit. This resource model provides a run-time representation of the resourcs.

Other offorts have been made to support the modeling of resources at design-time (e.g. as part of application models), such as [24] and [30].

Different resource models have been developed, having different abstraction levels and intended usages.

RM-ODP Resource Model

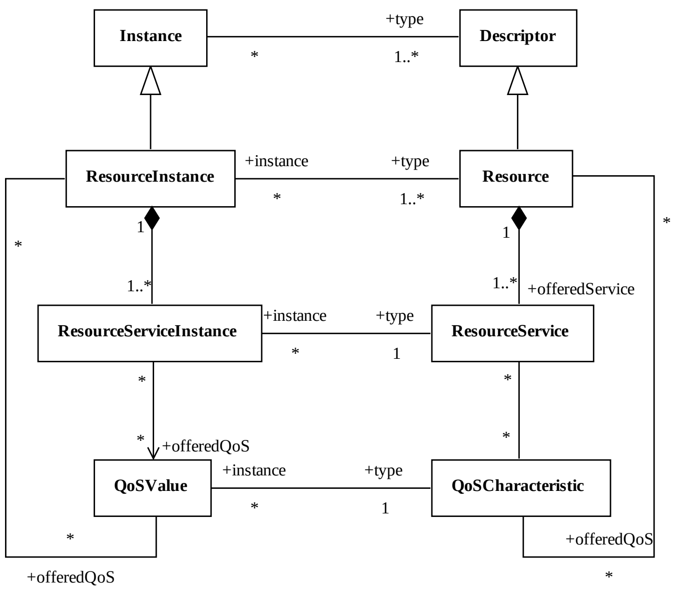
The resource model described in [15] is a refinement of the RM-ODP computation model [14]. It provides two software construction entities; *application objects* and *resource objects*. The limited capacity of resources is represented as QoS constraints.

The resource model allows a recursive hierarchy of resources, in which resource objects can also be application objects and depend on other resource objects. *Pure resources* are resources that do not rely on other resources.

UML Profile for Schedulability, Performance and Time

The core of the UML Profile for Schedulability, Performance and Time [24] is the General Resource Model. This model provides a framework for dealing with resources in analysis of schedulability-, performance- and time-aspects in predictable systems.

Figure 2.5 Core Resource Model



In the resource model, resources are viewed as servers that provide one or more services to their clients. A resource service, again, provides a certain level of QoS. It makes a destinction between descriptor elements (resource, resource service, and QoS characteristic) and instance elements (resource instance, resource service instance, and QoS value). The descriptor elements describe the instance element, for example, a QoS characteristic defines what QoS values could be used.

The profile has a usage model for modeling resource usage. This model is based on a causality model which can represent cause-effect chains. Resource usage can be modeled as static (constant) or dynamic (varies over time).

The General Resource Model also provides classification of resources. Classification can be done based on purpose, activeness, and protection. Purpose is either processing resource (capable of storing and executing code), communication resource (enabling communication between resources) and device (for devices that do not fit in the first two categories). The second classification criteria is activeness. An active resource is capable of generating a stimulus (i.e., make something happen), whereas passive resources must be explicitly prompted in other to be utilized. Resources are either protected or unprotected. Protected resources must be acquired in order to be used, and may only be used by one entity at a time. Acquirements are carried out according to an access control policy. Protected resources must be explicitly released in order for others to use them.

The resource management model of the GRM introduces resource brokers, that administer the access to resources according to access control policies. Resource managers keep track of the status of resources.

This UML profile provides a set of UML extensions to support the modeling of resources in UML models. It is meant to serve as basis for different real-time analysis techniques.

OpenORB Resource Metamodel

The resource metamodel of the OpenORB platform is presented in [11]. The main entity in this model is abstract resources, that represent real system resources. It has a hierarchic structure, where resources can be made up of other, lowel-level resources. For example a team of threads is an abstract resource composed of a set of thread resources. At the bottom of the hierarchy are the physical resources, such as CPUs. The uppermost level of the hierarchy is called a virtual task machine (VTM), capable of performing a certain task.

Each abstract resource has a resource manager. These mangers make up a manager hierarchy aligned with the resource hierarchy. A resource scheduler is a special type of manager, for scheduling lower-level processing resources.

Resource factories create abstract resources. They make up a another kind of hierarchy called the factory hierarchy. Higher-level resource factories can use lower-level factories in order to create their resources.

CQML+

CQML+ [30] is an extension of CQML that includes support for modeling resource demand. *Resource* is introduced as a new construct, but it is left to the user to specify the resource types and properties.

A resource has a name and a list of characteristics describing its QoS properties. CQML does not define any semantics for these characteristics, but instead leave it to the resource managers of the underlying system. Instead of making *quality statements* about components like in CQML, CQML+ supports expression of statements about resources using CQML syntax. Such expression could for example make statements about the required bandwidth of a network resource.

Other Approaches to Resource Modeling

In [5] it is proposed a uniform way to model resource usage. It specifies a model of the basic resource types. Resources can be classified in two ways; resources are either space-shared or time-shared, and either dependent or independent. Time-shared resources are shared in time, and can only be used by one task at each instant. Access to a time-shared resource is handled by a scheduler. Processors and network resources are typical time-shared resources. Space-shared resources can be viewed as a space that is shared among clients. It is a set of identical elements, and each task may use a subset of it. Memory could be viewed as a space-shared resource.

Dependent resources have a dependency-relationship to other resources to carry out their jobs. Independent resources are not dependent on other resources. This makes up a hierarchic structure of resources, where the independent resources are the leaf nodes.

2.4 QoS-Management in Middleware

In this section we look at how QoS can be handled by the middleware platform. We look at some approaches to enable platform-managed QoS, and some activities involved in realizing this.

2.4.1 Component Architectures

Software components are software entities inspired by their hardware equivalents. Components are computational entities whose functional specification is known. Applications can be assembled by certain compositions of components. A major goal of component architectures is providing the ability to re-use components in multiple applications. In distributed systems, component platforms may hide the distribution of components over a network.

An established definition of components by Clemens Szyperski, quoted in [29] is: *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject tocomposition by third parties.* The definition mentions interfaces as the contract between components. Explicit context dependencies mean that the component may pose certain requirements on its environment, such as the requirement of certain interfaces of other components or the component platform.

Common component architectures today are the Java 2 Enterprise Edition (J2EE) and Microsoft .NET platforms. The Object Management Group has released a specification for the CORBA Component Model, which is an evolution Of CORBA with components as the main abstraction.

It has been argued that QoS management should be handled by component platforms. We look at some attempts to manage QoS and resources in middleware and component platforms in particular.

2.4.2 QoS-Aware Component Architecture

The goal of the QoS-aware Component Architecture (QuA) is essential desirable features of current component architectures and to experiment with extensions to support QoS management.

The notion of a *service* is central in QuA. It is defined as *a subset of input messages to some composition of objects and their causally related outputs* [29]. A service is created by making the right composition of components. Services are expressed as *service specifications* that describe these compositions, as well as the inputs and outputs.

A mechanism to support QoS management in QuA is the *service planner*. The service planner takes as input both the service specification and a QoS specification in order to plan a service to fulfil these requirements. The functional and non-functioanl requirements are here expressed independently.

In order to fulfil the QoS requirements, the service planner can choose between different implementations of components with the same functional behaviour but deliver a different quality have different requirements to its environment (e.g., resource requirements). QuA provides a repository of component implementations with different properties.

Instead of having a single, common service planner, QuA supports the implementation of specialized service planners for different kinds of services, since there is not a single, general solution to the service planning problem [29].

The service planner uses utility functions in order to find optimal service compositions. A utility function returns a value between 0 (useless) and 1 (perfect) that represents the level of utility of a service. The utility function can be used to find out what trade-offs (e.g., with regards to resource requirements) will result in the best overall quality.

Components have access to their own implementation and the *service context* through reflection. Reflection can be used for QoS-based adaptation.

2.4.3 QoS Monitoring

QoS monitoring is concerned with observing the acheived QoS from parts of the system. Monitoring can take place in different parts of the system, for example within bindings.

In the QoS management architecture presented in [16], Aagedal includes a *QoS monitor* that monitors system behaviour according to a set of constraints. Thus, monitoring is concerned with making sure QoS requirements are fulfiled.

Monitoring the QoS provided by resources can be used as a mechanism for deciding when to perform adaptations in a system [18].

2.4.4 Resource Management

According to [7], resource management should be performed at the middleware level. Two main purposes of resource management are mentioned. The first is to acheive a high utilisation of system resources in order to enhance system performance. The second is to be able to allocate resources in order to meet application requirements. Both resource awareness and support for dynamic configuration of resources are important characteristics.

Chapter 3

QoS in Model-Driven Development

In this chapter we look at how Quality of Service concerns can be handled in a model-driven development context. We discuss where and how QoS concerns come in, and look at related research in this field.

By doing this, we define a context for the main topics of this thesis, and discuss in what ways we can contribute to the realization of this vision.

Figure 3.1 Managing QoS in MDA

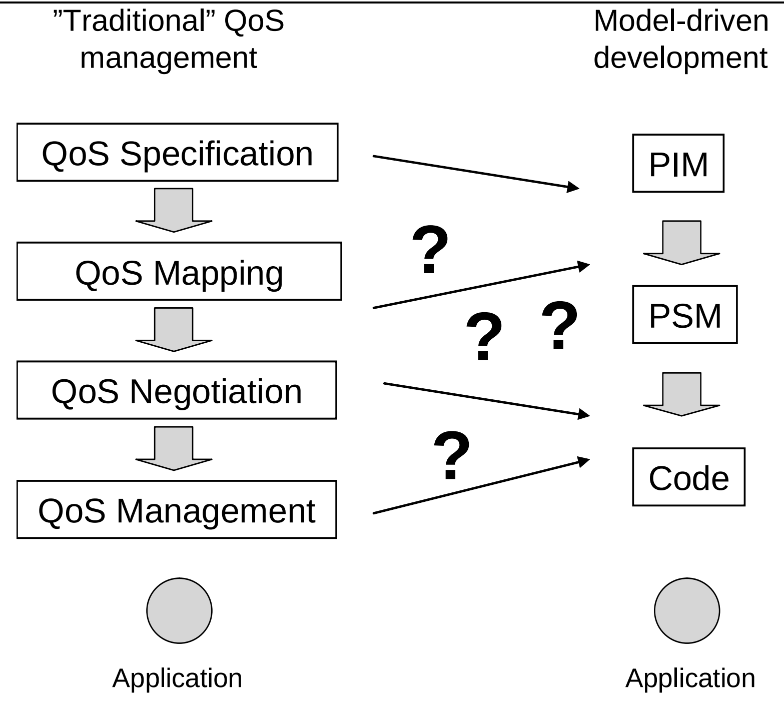


Figure 3.1 shows how we believe QoS management could be incorporated in model-driven development. The left-hand side of the figure illustrates some activities traditionally related to QoS-aware systems. It starts with specification of the required QoS. Mapping of QoS requirements to lower abstraction levels are typically necessary in order to realize high-level QoS requirements. QoS negotiation and QoS management are required also required in order to realize the specified QoS.

On the right-hand side, we show the typical abstraction levels of MDA, starting with a platform-independent model of the application, and refining it to meet the platform-specific concepts.

We believe that the traditional activities related to QoS management can be successfully incorporated in a model-driven development process.

3.1 QoS Specifi cation in Models

The UML Profile for Schedulability, Perfromance and Time Specification and Mechanisms [24] and the UML Profile for QoS [23] both provide support for QoS specification in UML models. Both provide support for QoS specification at different abstraction levels.

The latter UML profile supports specification of custom QoS characteristics for any domain imaginable. It is therefore well suited for both platform-independent and platform-specific models.

3.2 Platform-Independent QoS

A goal of MDA is to specify applications independently of implementation technology. This should also be the case when modeling QoS.

In [17] the authors motivate the introduction of QoS concepts at the plantform-independent level. It argues that services should be specified at a level of abstraction that does not consider the supporting infrastructures, and in that way be able to reason about qualitative aspects in the inital phases of the design.

By specifying QoS at in a platform-independent model, the realization of the QoS requirements can be handled when transforming it to a platform-specific model. The transformation process involves mapping the PIM

QoS requirements down to lower-level QoS requirements that the platform is able to manage. QoS requirements can also affect functional decisions when performing model transformation.

In order to be meaningful, QoS requirements must be associated with functional entities, so that we know what functionality it is describing. Since models are only abstractions of complete systems, the requirement must be connected with the system in such a way that it is clear what it really means. This means we must have strict rules for how QoS-requirements are associated with functional elements in the PIM.

3.3 QoS Vocabulary

An application developer needs to know what QoS properties he is allowed to express. We believe there is no general solution to modeling QoS in ways that the system can understand. Instead, a set of QoS characteristics must be provided along with information about how they are managed. Thus, a catalogue of predefined QoS characteristics should be available for the application developer. It should also be possible to extend this catalogue by defining new QoS characteristics and associated management descriptions, such as transformation rules. A similar approach is presented in [31], where *measurement designer* (a person responsible for defining ways of measuring QoS) is a separate role in the development process.

In the PIM, QoS requirements are modeled as QoS constraints using the UML profile for QoS. These QoS constraints must conform to QoS characteristics in the catalogue.

3.4 Considering QoS in Model Transformation

The model transformation activity takes the PIM as input and generates a PSM for a given platform.

We consider model transformation to be a suitable place for performing QoS mapping. By QoS mapping we mean translation of QoS requirements from higher to lower abstraction levels, as well as the assignment of QoS requirements to mechanisms capable of realizing them.

A framework for performing QoS-aware model transformations is proposed in [2]. Based on the assumption that QoS requirements impact the system design, the model transformation should be able to choose particular patterns in order to meet the requirements. The model transformer could get hold of suitable patterns by querying a broker service with access to a *QoS library*.

In [31], a model-driven development process supprting non-functional properties is suggested. The process targets the COMQUAD platform [10], which is developed as part of a research project. It argues that QoS should be expressed at multiple levels of abstraction, and model transformations should be specified to map between these levels.

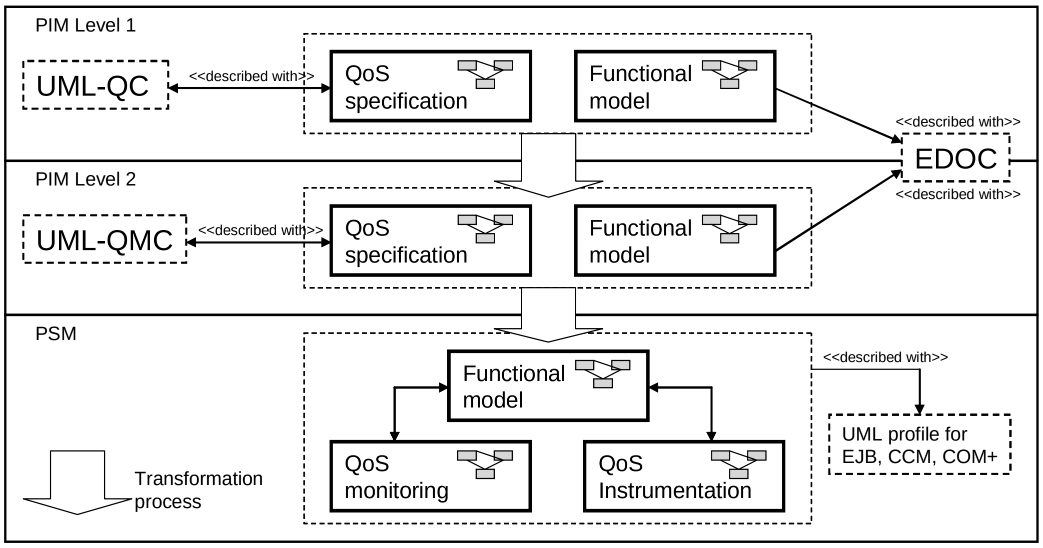
The development process defines two separate roles; *measurement designer* and *application designer*. The measurement designer defines the measurements, which are synonymous with QoS characteristics. The application designer, should be able to focus merely on modeling the application logic, and be able to use the measurements provided by the measurement designer without having to worry about how it is implemented.

Along with the measurement definition (expressed in CQML+) are *context models* that specify the context of the measurements. The context model describes the features of an application system that must be known in order to establish the value of a measurement [31]. Mapping between abstraction levels with different measurements is performed through transformations between context models. The definition of context models is said to allow measurements to be defined independently of the application. The context models and transformation specifications are all responsibilities of the measurement designer. For this approach to work, it is stated that a mapping between the context model and a component model (e.g. COMQUAD or QuA) must exist.

A method for model-driven development of component-based applications with QoS-support is proposed in [28].

QoS requirements are specified in the form of QoS contracts, and can be modeled independently of target platform (as part of the PIM). This method uses the UML profile for EDOC (Enterprise Distributed Object Computing) for the functional, platform-independent specification. The PIM model is divided into two steps, where the second step can be automatically generated through transformation from the first step. In the first step, called PIM level 1, the application is modeled as components, as well as QoS contracts (using a custom UML profile for QoS-aware com-

Figure 3.2 Process Overview



ponents.) The transformation to the second step, PIM level 2, generates monitors responsible for enforcing the QoS contracts specified in the first PIM. Monitors are modeled using another custom UML profile called the UML profile for QoS monitoring components.

The QoS characteristics available are included in a global QoS catalogue. The second PIM, containing QoS monitoring mechanisms, can be further transformed to a target platform technology, such as EJB, as exemplified in [28]. The example assumes the ability to monitor QoS at runtime. Components in the PIM are for example transformed to EJBs in the target platform.

3.5 Run-Time Support for QoS in MDA

In his master thesis [20], Lars Lundby presents a framework for modeling QoS-aware applications using UML and CQML, facilitating the generation of QoS management- and configuration code. The run-time support of the framework includes a run-time implementation of CQML and support for CQML monitoring.

Resource management on a middleware platform can also support model-driven development. In [3] we discuss how a platform-level resource model can be used to add and release resources in order to meet QoS requirements.

By modeling resources as *services*, we can quantify their usage and capacity in terms of QoS (e.g. bandwidth). This allows the realization of reosurce-level QoS requirements on the platform resource model, and could used be a target for QoS mapping in a model-driven process.

Chapter 4

Resource Model

In this chapter we present a generic resource model. It is intended to be used as a basic framework for dealing with resources in Quality of Service aware systems. A version of this resource model has been published in [3].

The design of the resource model is presented in a set of UML diagrams. We also describe a prototype implementation of the resource model. The prototype implementation is the target platform for our experimentation with QoS management in model-driven development. Also, to support model-driven development, we have defined a UML profile for modeling resources, providing a syntax for modeling resource concepts in UML.

4.1 Basic Concepts

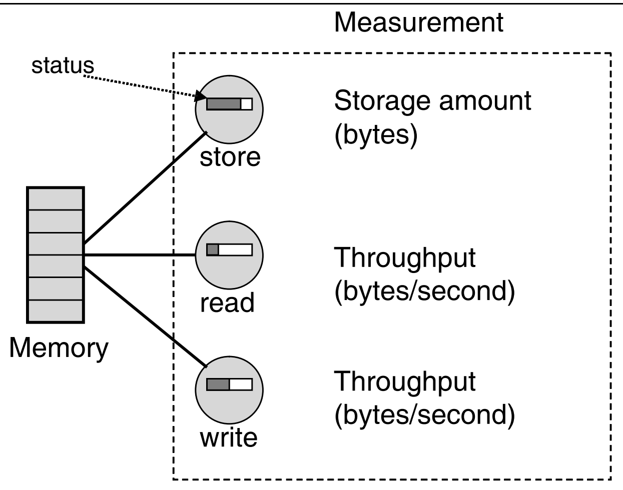
We need to define the basic concepts to be used later in the thesis. For instance, the word *resource* has some very broad definitions. It means different things to different people, and within different domains. Thus, we need to specify what it means in our context.

4.1.1 Resource

The main unit of abstraction in a resource model is the *resource*. A resource represents an entity in the system, physical or logical, that can be used by a client. A client is a generic term meaning any system entity allowed to use resources (such as a process or a thread).

Resources provide a set of services to their clients, each with a capacity indicating the maximum level of QoS. We refer to this as the *capacity* of the service. A resource is limited and has a fixed capacity. Furthermore, resources have a status reflecting the currently available and currently delivered QoS. Thus, resource usage can be quantified in terms of Quality of Service measurements.

Figure 4.1 Memory Resource



Resource services can be seen as an level of abstraction above the resources, each representing one kind of resource usage. For example, as illustrated in Figure 4.1, a memory resource may provide the services *store*, *read* and *write*, each providing a certain level of QoS. Each service maintains a status indicating how much of the offered QoS is currently in use.

From our viewpoint, a resource is a provider of services, and not the “material” delivered through the service. If we (for some awkward reason) wanted to model a goldmine, our resource abstraction would be the goldmine, not the gold. Likewise, we model a processor as a resource and not as CPU cycles.

System performance is largely dependent on resource performance, and can be optimized and customized by having access to resource management facilities. In order to support QoS management, some additional requirements must apply to resources:

- The resource can be managed by a resource manager.
- The resource has a measureable status.
- Resource usage can be monitored and constrained in order to enforce reservations.

Examples of resources include physical resources such as *memory*, *processors* and *network connections*, and logical resources such as *buffers*, *thread pools* and *virtual processors*.

4.1.2 Resource Model

A *resource model* is a model of resources. According to [32], a model is a *description* or *specification* of a system under study. A specification represents something to be made before making it, while a description represents something in existence. The resource model is actually something in between, as it is a description of the resources already in the system, but is also a specification of a new run-time representation of resources that doesn't exist on beforehand.

The resource model is a description of the resources in the system from a specific point of view, and from a specific level of abstraction. It also includes additional properties, such as resource management mechanisms, that do not exist in the system, but we have to implement ourselves.

An implementation of the resource model is a platform for resource management, providing facilities that can be used to carry this out. This includes abstract representations of resources and resource managers and the interfaces they provide. The run-time entities such as resources and maangers are instantiations of the resource model.

In our context, the resource model serves another purpose as well; it acts as a domain model, or “virtual metamodel” for a UML profile for modeling resource instances. It shows the relation between the concepts included in the profile and how they fit into their context.

Resources are typically heterogeneous entities, and a goal in designing a resource model is to find an abstraction level on which resources can be modeled in a uniform way.

Resource-related concepts include:

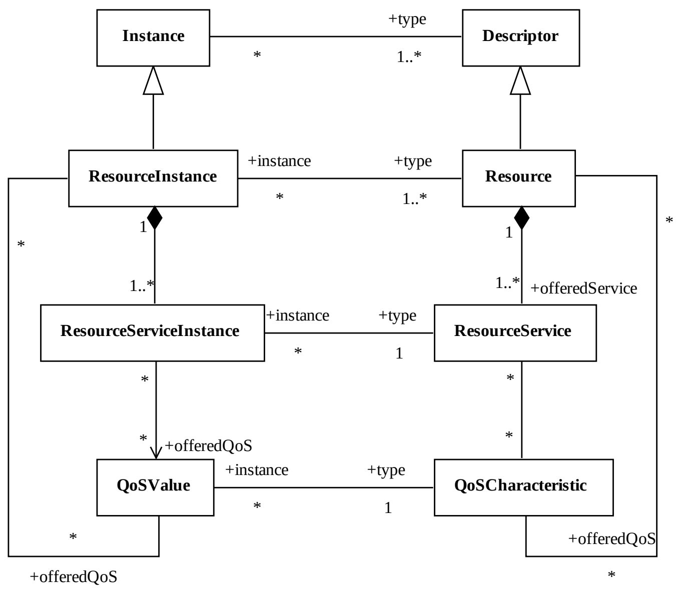
- Resource representation, expressing how resource instances are represented on the target platforms.
- How resource usage and capacity is measured.
- How to interact with resources and resource managers.
- How to represent relationships between resources, such as dependencies.
- Resource protection and admission control.

4.1.3 Resource Model Implementation

The *implementation* of the resource model is a piece of runnable code that implements the concepts of the resource model. It supports the instantiation of run-time resource models.

4.1.4 Resource Model and Run-Time Resources

Figure 4.2 Resources and Resource Instances



We adopt the concepts of the core resource model from the UML Profile for Schedulability, Performance and Time [24]. In our resource model, the left-hand side of Figure 4.2 represents run-time instances that exist on the platform on which the resource model is deployed. The right-hand side represents the design-time elements.

Like in [24], resources provide one or more services, and services can be quantified using QoS characteristics. However, unlike Figure 4.2, we do not allow a resource service to provide any number of QoS characteristics. Instead, a single QoS characteristic quantifies the service, and at run-time, its usage level and capacity must be values of this characteristic.

4.2 Requirements and Design Goals

We here present a set of requirements that the resource model must fulfil.

In [7] a set of evaluation criteria for adaptive resource management in middleware are presented. We use these as an input for what is required of the underlying resource model, since we consider the resource model as a basis for the QoS management mechanisms. The first requirement presented is that *resource sharing* must be controlled and predictable. Thus, the resource model must provide the necessary facilities to control resource sharing. In order to solve this, we require that *reservation* of resources must be supported. This guarantees that clients get the level of QoS they need as long as sufficient resource capacity is available.

The authors further mention resource awareness as a requirement, meaning that the systet must be aware of the availale resources. This is an inportant purpose of the resource model.

The resource model must be *generic*, meaning it must support all kinds of resources. To be a valid resource, it must satisfy the definition in Section 4.1.1. A necessary assumption is that the environment (e.g., operating system) provides the necessary control of resources; the resource model cannot support management of all thinkable resources without explicitly providing management methods. However, there must be no inherent limitation in the resource model per se that hinders certain (valid) resources.

The range of supported resources must be *extensible*, in order to support the introduction of new resource types. It must be possible to model new resources in the same way as existing resources.

In [7] it is argued that resources are very heterogenerous, and this must be tackled by the resource model. Therefore, we try to model resources as *uniformly* as possible, meaning that resources are represented the same way and can be managed using the same kinds of interfaces despite their heterogeneity. This results in a more consistent model that is easier to use and understand, since it hides the underlying details about the resoures.

The resource model must support *admission control*. Some resources have certain access control policies that make sure clients have the necessary rights to access them. This must be reflected in the resource model by implementing mechanisms for access checking and querying.

Resource mapping from design-time models to run-time resources must be supported. A mechanism for this mapping must be available in order to facilitate our vision of using the resource model as a part of model-driven development.

The resource model must support *logical resources* as well as physical resources. It must be possible to deal with resources that are implemented as software entities (for example thread pools or buffers) as well as physical resources. Such resources typically depend on hardware entities to carry out their services, and these relationships should be considered.

4.3 Design

We here present our design of the resource model. It is a specification of the resource concepts from an implementation point of view.

The resource model is specified in UML using class diagrams. The classes are metaclasses, and meant to be instantiated in runtime resourcs.

We present a set of small diagrams that all represent parts of the model. The purpose of this is to leave out details not related to the respective sections. We summarize the design with a figure containing the complete resource model.

4.3.1 Relation to the UML Profile for QoS

We use the UML profile for QoS [23] as our QoS vocabulary in the resource model. It is used for dealing with resource QoS in different contexts:

- for modeling with QoS concepts in the resource model.
- for handling resource QoS for run-time resource instances (part of the resource model implementation).
- for modeling resource QoS in application models (using the UML profile for resource modeling defined later in this chapter).

The UML profile for QoS is a general approach to supporting QoS specification in models, and a wide range of different QoS characteristics can be constructed. Our needs only stretch to QoS specification and for resources. This means we that only need a subset of the expressibility in the UML profile.

A QoSCharacteristic is composed of one or more QoSDimensions. A dimension defines the type of one aspect of a QoSValue. For our usage of the QoS profile, we only allow one QoSDimension per QoSCharacteristic. This means that a QoSValue is a single value.

QoSConstraints can be defined, limiting the range of allowed values within a QoSCharacteristic. The QoS profile supports practically infinite ways to specify QoSConstraints. For our resource QoS usage, we will only support simple invariant expressions having the form

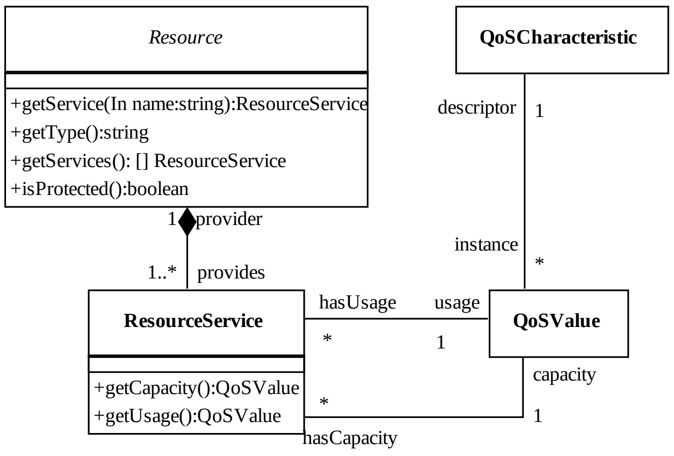
```
<QoSDimension> <ComparisonOperator> <Value>
```

For example: `rate >= 100`. We support three kinds of basic types for QoS dimensions: *float*, *integer* and *enumeration*.

4.3.2 Core Resource Model

The core resource model (Figure 4.3) shows the basic resource concepts that are common to resources. It borrows its QoS vocabulary from the UML Profile for QoS [23]. A *QoSCharacteristic* defines the range of allowed values of a *QoSValue*. We only use a minor subset of the expressibility of the QoS profile necessary to specify resource QoS.

Figure 4.3 Core Resource Model



Resource objects represent system resources. A resource provides access to its services, either by providing the serivce name (`getService`) or by retrieving a list of all services (`getServices`). Also, a resource can be queried about its type. Resource types are identified by their names (e.g. “read”).

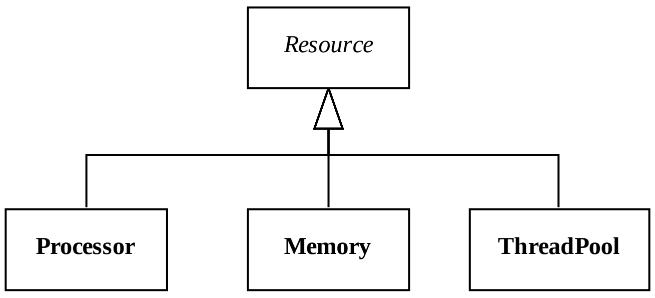
A resource provides one or more *ResourceSerivce*s. Resource usage and capacity is quantified through these services in terms of *QoSValue*s. The range of valid *QoSValue*s is defined by their type, or *QoSCharacteristic*s. Each resource service must have an associated *QoSCharacteristic* defining how it can be measured. The *ResourceSerivce* can be queried for current usage and capacity through the `status` and `capacity` operations. The `isProtected` method is used to check whether or a reource is protected.

The *Resource* class is declared abstract because it meant to be subclassed to specific resource types. Since every resource must be of exactly one type, we do not want to instantiate the *Resource* class.

4.3.3 Resource Types

Every resource must be of a single type. Resource types are either direct subclasses of the abstract class *Resource* or subclasses of other resource types.

Figure 4.4 Resource Types



The behavoiur of a resource is determined by its type. This is ensured by requiring that resources of a certain type always provide the same set of services. We know by its type that a resource is capable of performing a certain task, but we know nothing about the QoS (capacity) of the resources. The capacity may differ between resource instances. The fixed set of services provided by a specific resource type share the same QoSCharacteristic, which is instantiated as QoSValues for resource instances.

Resource types may subclass other resource types, and make up a resource type hierarchy. For example, *RISCProcessor* and *CISCProcessor* may be specializations of the more general *Processor* resource. Subclassing is necessary until we reach a level where the behaviour of the resource type resembles the behaviour of the actual resource.

The implementation of a resource type includes “glue code” that maps the abstract representation of the resource to the actual resource. This glue code could for example be for utilizing resource management APIs provided by a realtime-capable operating system. If the OS completely hides all control over resource allocation, it is not possible to implement a resource type for it.

In Figure 4.4, three resource types are shown. Processor and Memory are typically physical resources, while ThreadPool is a logical resource. Each one must provide the set of services required by the type. These are only examples; the set of aviable is meant to be extended to support as many different resources as possible. In each context where the resource model is in use, there must be a common set of resource types available. A resource type is identified by its name, and has a fixed set of resource services attached.

4.3.4 Resource Type Catalogue

An underlying assumption for using this resource model is the existence of a global *resource type catalogue*. By this, we mean that the range of available resource types and some of their characteristics must be known in order to use the resource model. The following must be true:

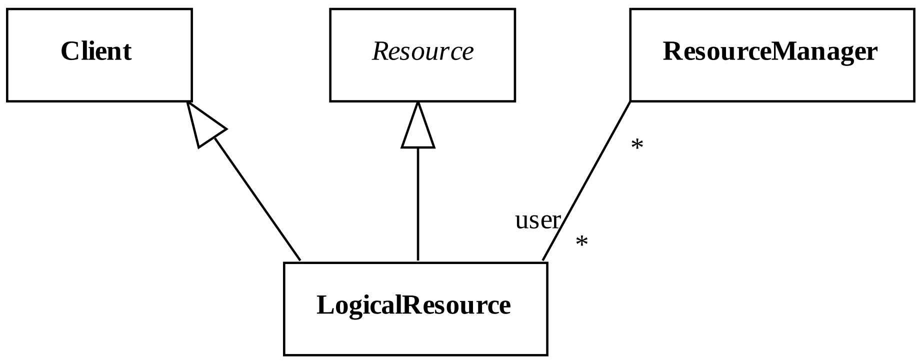
- The set of services provided by a resource and their behaviour must be known to the modeler that has to deal with resources. A resource and its services are represented by names, that uniquely defines them anywhere in the development process.
- The QoS characteristics that quantify resource services must be known by the modeler dealing with resource QoS. It must be possible to express these characteristics using the UML profile for QoS.
- A runtime representation of QoS the QoS characteristics of resource services must available at run-time as well as design-time, as these are required to hold values related to resources.
- Information about resource types must be available at run-time. This information can for example be used when performing model transformation and code generation.

This ensures awareness of all the available resources in the system and how to deal with them.

4.3.5 Logical Resource Model

Logical resources (Figure 4.5) do not map to real resources in the system, but resemble the same structure and properties as physical resources. Logical resources may use other resources (physical or logical) to carry out its services. This allows a hierarchy of resources, where the logical resources are above the physical resources. Physical resources are always leaf nodes. Logical resources can also be leaf nodes if they do not depend on any other resources. As well as using other resources, logical resources have their own level of logic.

Figure 4.5 Logical Resource Model



As seen in Figure 4.5, logical resources are sublcasses of Client. This means that they are allowed to reserve resources. Unlike physical resources, logical resources do not implement “glue code” that binds them to the actual resources. The resource manager is used to reserve other resources. For example, a thread pool is a logical resource that depends on a physical processor resource.

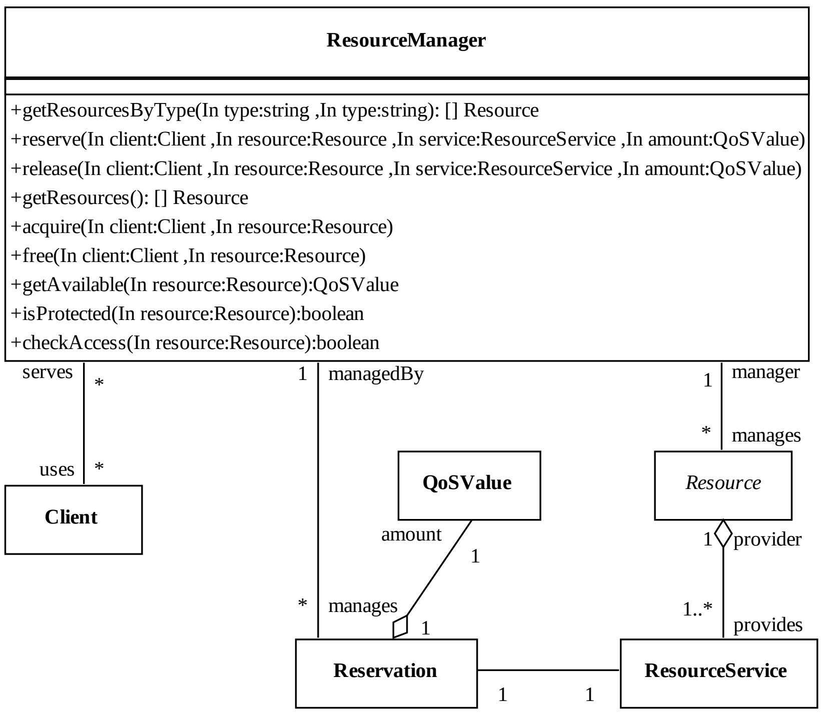
A logical resource that requires other resources in order to perform its services, is responsible of reserving these resources.

4.3.6 Resource Management Model

The resource manager is responsible for providing resource management facilities to the clients. A resource manager can manage a number of resources, while a resource can only be managed by a single manager. A resource manager may only manage resources that exist within the same address space.

Resource managers are responsible for providing access to resources, reservation of resources, enforcement reservations by constraining usage, and adminssion control.

Figure 4.6 Resource Management Model


ResourceManager
+getResourcesByType(In type:string ,In type:string): [] Resource
+reserve(In client:Client ,In resource:Resource ,In service:ResourceService ,In amount:QoSValue)
+release(In client:Client ,In resource:Resource ,In service:ResourceService ,In amount:QoSValue)
+getResources(): [] Resource
+acquire(In client:Client ,In resource:Resource)
+free(In client:Client ,In resource:Resource)
+getAvailable(In resource:Resource):QoSValue
+isProtected(In resource:Resource):boolean
+checkAccess(In resource:Resource):boolean
serves
*
uses
*
Client
1
managedBy
*
manages
QoSValue
amount
1
1
Reservation
1
1
ResourceService
manager
1
*
manages
Resource
1
provider
1..*
provides

Resource Access

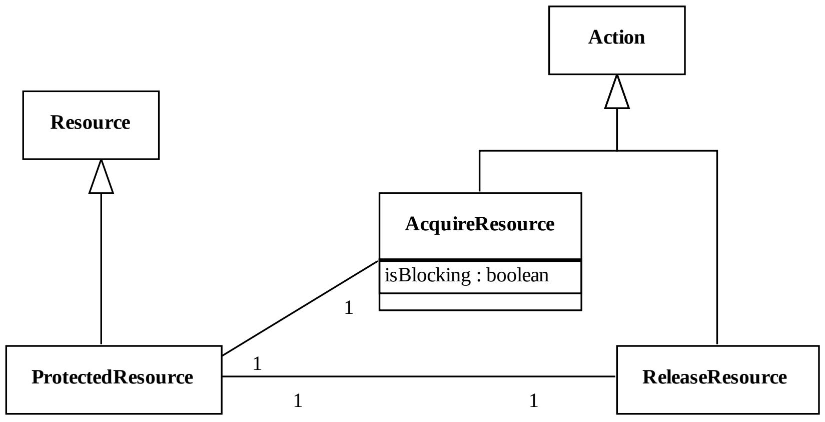
In order to get access to resources and resource services, clients must query the resource manager of that resource. The resource manager provides two methods for gaining resource access; `getResources` which returns an array of all the managed resources, and `getResourcesByType` which returns only the resources of a specific type.

Reservation

Reservation can be defined as acquiring the right to use a resource. A reservation is a contract between a client and a resource service, and includes a value with the amount that is reserved. It is the resource manager that is responsible of keeping track of reservations. A reservation is made using the `reserve` method. Input arguments are client, resource service and amount. A reservation is active until it is explicitly released, using the `release` method. It takes the same arguments as `reserve`, and `amount` is the amount to be released. The last argument is optional, and if it is not given, the entire reserved amount will be released.

Protection

Figure 4.7 Resource Access Model



Resource protection is concerned with access to resources. This is also a responsibility of the resource manager. We adopt the notion of a protected resource from the General Resource Model in [24]. We use a simplified conceptual model (Figure 4.7), where a resource provides a single service instead of providing per-service protection.

The *AcquireResource* action is either blocking or non-blocking. If blocking, the client will be blocked when the operation is called, and must wait until access is granted. If it is non-blocking, the client will either get a negative result immediately if access cannot be granted, or it will gain access.

The operations `acquireResource` and `releaseResource` are concerned with admission control. Some resources are *protected*, meaning they cannot be used concurrently by any number of clients as long as there is spare capacity. A protected resource requires the client to *acquire* it before using it. When done using it, the client must *release* it to make it available to other clients.

The *ResourceManager* in Figure 4.6 is responsible for keeping track of a set of resources on a platform. It is responsible for providing clients with access to resources, and performing admission control.

Decisions related to admission control, such as deciding to let a client acquire a resource, are made based on `access control policies`. An access control policy might contain statements about things like number of concurrent users, access levels of clients, etc. An *AccessControlPolicy* is associated with a resource.

Resource Managers provide two methods for providing access to the resources they manage. `getAllResources` returns a set of references to all resources managed by the respective manager. `getResourcesOfType` returns only the resources of a certain type.

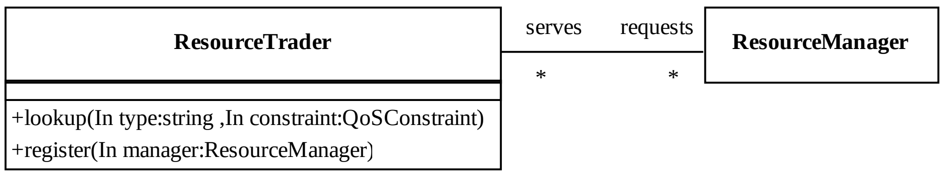
The attribute `isProtected` is true for protected resources. The default value is false.

4.3.7 Resource Trader

The resource trader (Figure 4.8) can be used to locate and select resources based on client requirements. It provides the service of resource discovery based on QoS constraints.

The `lookup` method is used by clients to request appropriate resources. Two parameters are given; `resourceType` and `requiredQoS`. The first

Figure 4.8 Resource Trader


ResourceTrader
+lookup(In type:string ,In constraint:QoSConstraint)
+register(In manager:ResourceManager)
serves
requests
*
*
ResourceManager

is a string, containing the name of the resource type. The second is an array of QoS constraints which specifies the QoS required from the resource service(s). The constraint may include one or more services provided by the resource, and one QoS constraint is given for each service we want to constrain.

If a resource matching the specified criteria is found, a reference to the resource is returned. If no match is found, the `null` value is returned.

In order to access and discover resources, the resource trader uses one or more resource managers. A manager can be registered with a trader using the `register` method. A reference to the resource manager is given as parameter.

4.3.8 QoS Repository

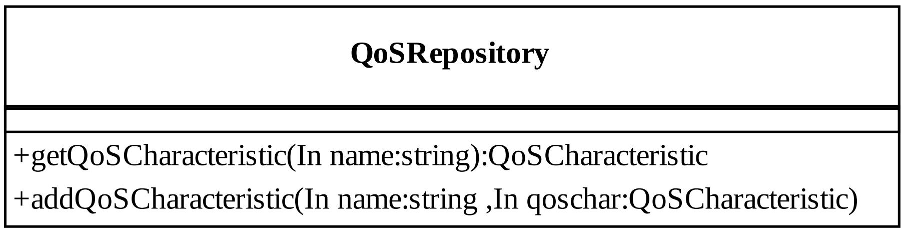
QoS characteristics represent the types of values in the UML profile for QoS. These characteristics and their values must be available at run-time as well as design-time. It would be a lot of redundant work to have to define these characteristics every time we need them. To solve this, we define a QoS repository to store QoS characteristics at run-time.

The QoSRepository is shown in 4.9. A QoS characteristic is identified by its name (e.g. *CommunicationThroughput* or *StorageAmount*). The operation `getQoSCharacteristic` takes the name as an argument, and returns the QoS characteristic.

A QoS characteristic is capable of instantiating QoS values, thus having access to the QoS characteristics is sufficient for being able to define values.

The QoS repository can be extended with new QoS characteristics by using the `addQoSCharacteristic` operation. The name of the charac-

Figure 4.9 QoS Repository


QoSRepository
+getQoSCharacteristic(In name:string):QoSCharacteristic
+addQoSCharacteristic(In name:string ,In qoschar:QoSCharacteristic)

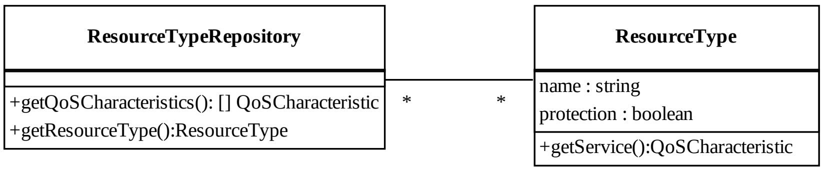
teristic and the characteristic itself is given as arguments, in order to be stored in the repository.

The QoS repository must be globally available to any entity that works with QoS characteristics. This enables a common QoS vocabulary throughout the resource model.

4.3.9 Resource Type Repository

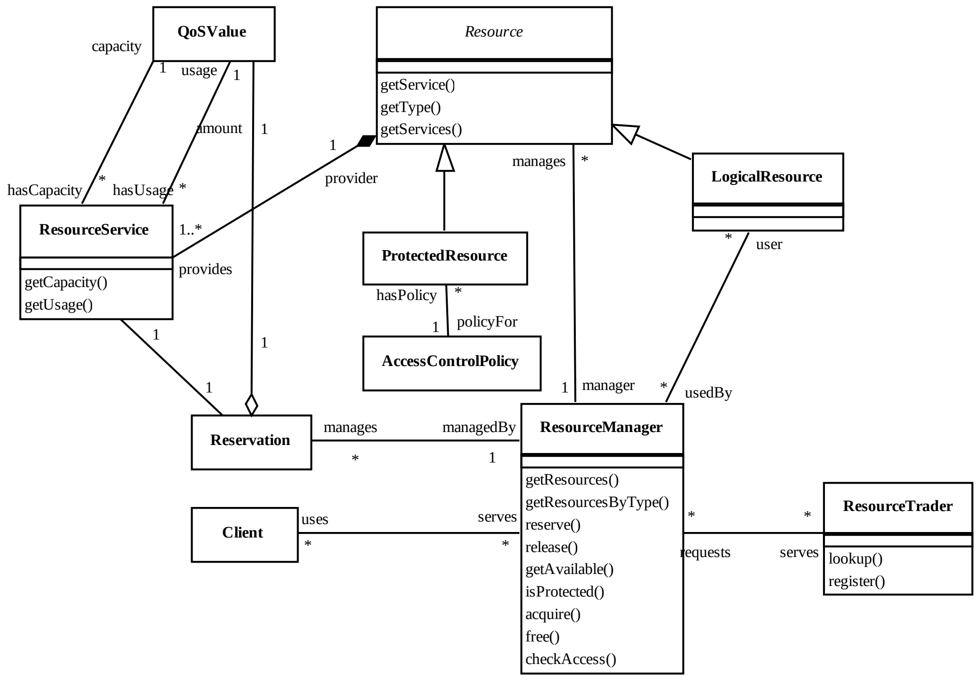
The resource type repository (Figure 4.10) can be used to acquire information about resource types at run-time. A resource type is identified by its name. The `getQoSCharacteristics` method takes the name as parameter, and returns the list of QoS characteristics for the services provided by the resource. `getResourceType` returns a *ResourceType* object, which contains information about the resource type, such as its *name* and whether it is a protected resource. The `getService` method returns the QoS characteristic of the requested service.

Figure 4.10 QoS Repository


ResourceTypeRepository
+getQoSCharacteristics(): [] QoSCharacteristic
+getResourceType():ResourceType
*
*
ResourceType
name : string
protection : boolean
+getService():QoSCharacteristic

4.3.10 Complete Model

Figure 4.11 Resource Model



We have gathered the main entities of the resource model in Figure 4.11.

4.4 Prototype Implementation

We have done a prototype implementation of the resource model in order to validate the resource model design, and to create a target platform for experimentation with resource management.

The programming language of choice for implementing the resource model is *Python*. Python is an object oriented, interpreted dynamic langeuage, and is well suited for rapid prototyping [1]. Also mentioned in [1] is the ability to test individual code segments in the interactive interpreter. This is a great advantage for us as well, in order to tackle the complexity and “play around” with concepts. A particular feature in Python useful for the resource model is the ability to implement *special methods* that emulate mathematical operators. This enables us to treat QoS values like primitive values such as integers. For example, one could support adding two QoS values together using the regular plus (+) operator.

4.4.1 QoS Profile Implementation

We have implemented a small subset of the concepts of the UML profile for QoS [23]. This includes the vocabulary necessary to quantify resource usage. Our mission is not to validate the QoS profile through implementation, but that resources can be quantified in multiple way by using its concepts.

We have also added some new operations to the classes of the QoS profile's domain model to realize some functionality, such as functions for the instantiation of QoS values from QoS characteristics.

The concept of a QoS characteristic is greatly simplified. In the QoS profile, a QoS characteristic can be composed of several QoS dimensions. A QoS value is an instance of a characteristic, and thus an instance of all these QoS dimensions combined. In the resource model, we describe a QoS characteristic by a single dimension. In order to reduce code complexity, we do not use the QoS dimension concept at all, but instead include its contents in the QoS characteristic. Thus, QoS characteristics represent types and QoS values represent values.

QoSCharacteristic

Instances of this class represent the QoS characteristics at runtime. They contain the information needed about how QoS values are measured. QoS characteristics have the ability to instantiate QoS values, and to compare two QoS values. A QoS characteristic can be created with:

```
qc = QoSCharacteristic(name, basictype, unit, direction, [enum])
```

where *name* is a string with the name of the characteristic, and *basictype* is the primitive type of the of QoS values it describes. Unit is the meausring unit the characteristic describes, and direction (either *'increasing'* or *'decreasing'*) decides whether resource usage goes up when the value increases, or whether it goes up when the value decreases. The last argument for creating QoSCharacteristic is *enum*. This argument is only required if the basic type is enumeration.

We have implemented support for three kinds of basic types. These were selected because they map directly to Python's basic types, and they are sufficient to cover most resource measurements.

integer All integers.

float All float values (same precision as the float type of the Python implementation)

enumeration The range of allowed values is defined as an enumeration of states. It is assumed the difference between one state and the next is the same for any two subsequent states. It maps to a Python list.

Two methods exist for instantiating QoS values from a QoS characteristics; `makeZero` for the zero (i.e., non-usage) value and `makeValue` for any value. `makeZero` calls `makeValue` with the zero value of the characteristic. For the two numerical types, zero is the number 0. For the enumeration type, zero is either the first or last state (depending on whether the direction is *increasing* or *decreasing*).

`makeValue` creates a QoSValue object with the specified value.

QoSValue

A QoS value is an instance of QoS characteristic. Instantiation is done by creating a new QoSValue object. A QoS value can created with the following syntax:

```
qosval = QoSValue(qoschar, [value])
```

The first parameter is the characteristic of the value, and the second is the initial value. The value parameter is optional, and if not provided, the value will be set to None (undefined). It can then be set at a later time using the `set` method.

The `value` method is used to retrieve the primitive value of the QoS value (which is either a integer, a float, or a string representing a state within an enumeration). `set` is used to set this value. `getQoSCharacteristic` is used to retrieve the QoSCharacteristic object from the QoS value. The QoS characteristic and primitive value are attributes of the object.

Special methods for arithmetic and comparison operators have been implemented for QoS values. This is done to be able to use QoS values in expressions as if they were primitive values.

`__lt__`, `__le__`, `__gt__` and `__ge__` implement the comparison operators. In order to compare values, the methods call the `compare` method of their QoS characteristic. The characteristic holds enough information to decide which value is the highest.

In addition to comparison operators, we have also implemented special methods for addition and subtraction of QoS values. The methods `__add__` and `__sub__` support inclusion of QoS values in expressions like `val1 + val2` and `val1 - val2`. Here, too, the calculation is forwarded to the QoS characteristic by calling the `addValues` method. In order to support the notation `val1 += valu2` (a convenient shortcut for `val1 = val1 + val2`), we have implemented the `__radd__` and `__rsub__` special methods.

QoSConstraint

As mentioned in Section 4.3.1, our usage of QoS constraints is limited to expressions on the form:

```
<QoSDimension> <ComparisonOperator> <Value>
```

Instead of representing the *QoS dimension* part of the expression in the QoSConstraint object, we consider it implicit. Instead, functionality to evaluate values against the constraint has been added. By evaluation we mean checking that a QoS value is within the legal range of the constraint.

A QoS constraint is constructed with two arguments. The first one, `refval`, is the reference value on the right-hand side of the expression. The comparison operator is given in the `op` argument. The argument is a string that identifies the operator (`'lt'` for less than, `'le'` for less than or equal to, and so on).

The `evaluate` method evaluates the QoS constraint against a QoS value. It first checks which operator is defined for this constraint, and then compares the value to be evaluated against the reference value.

4.4.2 Resource Model Implementation

The resource model implementation depends on the QoS profile for reasoning about QoS. It is an implementation of the concepts presented in the resource model. However, we have not implemented resource types for any physical resources such as network connections or processors. Instead, we look at how this mapping has been realized in other systems.

Resource

The *Resource* class is the superclass of all resource types. It implements some methods common to all resources, that are inherited by the subclasses. Its constructor is called from the subclasses when instantiated, with a list of *ResourceService* objects and the name of the resource type as parameters.

Resource also contains methods to access the resource services. `getServices` returns the list of all provided services, and `getService` takes the service name as argument, and find this service in the list of services.

Resource types are subclasses of *Resource*. These implement the resource specific code that glues the resource model representation of the resource with the actual resource in the system.

ResourceService

Resource services are provided by resources. A resource service is created with:

```
srvc = ResourceService(name, qoschar, cap)
```

The *name* is a string that identifies the service. *qoschar* is the QoS characteristic for this service, and *cap* is the capacity (which has *qoschar* as its type).

The *ResourceService* object provides methods for checking status; *getUsage* for to see how much is currently used and *getCapacity* to check the capacity of the service. `getName` and `getQoSCharacteristics` are used to inspect the service name and its QoS characteristic.

ResourceManager

The *ResourceManager* provides resource management functions to its clinets. A resource manager object keeps a list of the resources it managers, and a dictionary of reservations.

The methods `getResources` and `getResourcesByType` are used to acquire references to the resources. The first returns the whole list of managed resources, while the second iterates through the list of managed resourecs, and picks out the ones of a specific type. The type is given as an argument.

The manager keeps track of how much a resource service's capacity is available for reservation. This can be checked using `getAvailability`. The method calculates the sum of all reserved amounts of this service, and then finds the difference between this sum and the capacity.

Reservation of resources is done with `reserve`. It takes the following arguments:

```
reserve(client, resource, service, amount)
```

Thus, a client may reserve a certain amount of a certain resource service. `reserve` first calls the `getAvailability` method to see if there is enough spare capacity. If not, the False value is returned. If enough is available, we go on to check whether a reservation of this resource service to this client already exists. If so, we add the amount to reserve to the amount of the reservation. If not, we create a new reservation.

`release` is the opposite method, and is used to release reserved amounts. We provide the *service* and the *client* as arguments, along with the *amount* to be reserved. Amount is optional, and if no value is given, everything that the client has reserved of the resource will be released.

A *Reservation* object represents each reservation. It keeps the reserved amount, and implements special methods for adding and subtracting amounts.

Resource Confi gurations

We have used a custom XML-based format to store information about the resources that exist on a platform. A configuration file can be read by a resource manager by calling a method named `readConfig` that takes the filename as parameter.

For each resource, we store the resource type (e.g., *Memory*) and the capacities of all the services it provides. We also store the *location* of the resource, so that it can be mapped to the actual resource it represents. Since we have not implemented resource types for any physical resources, the location is left *unspecified* in our usage examples.

AccessControlPolicy

We have implemented a very simple type of access control policies, where the only property is the number of clients allowed to use the resource concurrently. The *AccessControlPolicy* object is created in each resource type implementation, and is thus resource type-specific.

ResourceTrader

The *ResourceTrader* object has access to a set of managers that it can query in order to find appropriate resources. New resource managers can be added to this set by calling

```
rtrader.register(manager)
```

The `lookup` method is used to discover resources based on the resource type and some QoS constraints. The name of the resource type (e.g., *'Memory'*) and a list of tuples of QoSConstraint objects and names of the services they refer to are provided as parameters.

The method iterates all the resource managers it has access to. For each of these, it requests a list of all resources of the required type using the `getResourcesByType` method. For each of the QoS constraints provided by the caller, it uses the `getAvailability` method of the resource manager to check how much is available for reservation. If there is enough avaiable capacity for all the required services, the resource broker returns the matching resource.

ResourceTypeRepository

The *ResourceTypeRepository* is used for acessing information about different resource types at run-time. The resource type information is stored in a file. It can be read using the `readResourceTypes` method and stored using the `saveResourceTypes` method. `getQoSCharacteristic(rname)` returns a dict, where each key is the name of a service provided by the resource, and the value is the QoS characteristic. The QoS characteristic is retrieved from the *QoSRepository*.

The resource type repository can also return information about a single resource type with `getResourceType(name)`. The method returns a *ResourceType* object, holding information about *protection* and a list of the provided services.

4.5 Platform Integration of the Resource Model

The resource model presented in this chapter is meant for implementation on a middleware platform. Most importantly, the resource management interfaces, such as the resource manager and the resource broker must be available to tasks in the system responsible for managing resources.

For example, in reflective component architectures, the resource model could be made available through a meta-interface. In QuA [29], the service planner would typically need access to resource model in order to manage resources when planning services.

4.6 UML Profi le for Resource Modeling

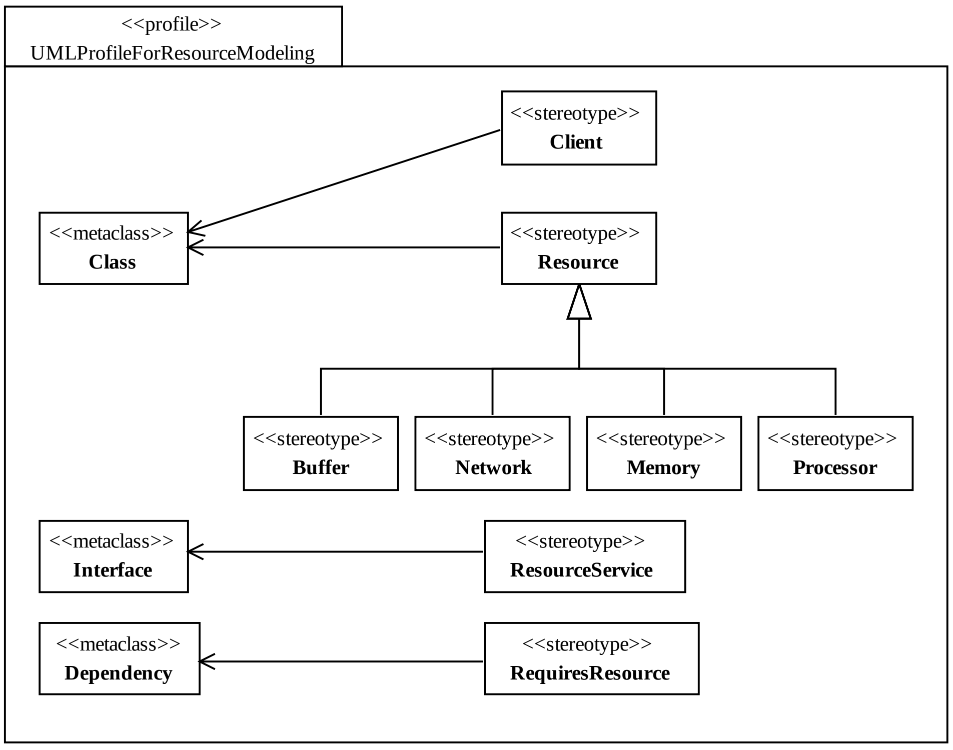
We have defined a UML profile for modeling with the concepts of thes resource model in UML models. This will allow us to deal with resources at design-time, modeling them and adding QoS requirements to them. The reosurce model defines the domain for the profile, and is in this sense similar to a metamodel. Unlike a metamodel, it does not define the rules for how the profile can be used.

For the QoS part of the resource modeling, we use a subset of the UML profile for QoS, including some rules about how it may be used.

The UML profile has been defined with the support for model-driven development in mind. However, is could be used for modeling resources in general.

The UML viewpoint of the profile is shown in Figure 4.12. The intended interpretation of the stereotypes is as follows:

Figure 4.12 UML Profile for Resource Modeling



Client

A client is the consumer of resources. A class stereotyped with *Client* can reserve and use resources, and may have dependencies to them in the model.

Resource

This stereotype is the superclass of all resources. It is abstract because only its children are meant to be instantiated. Resources are modeled using sublcasses of *Resource*, such as *Network* or *Memory*. These subclasses are all in the global resource type catalogue.

ResourceService

Every resource has one or more resource services. In this UML profile, services are modeled as interfaces provided by the resource, stereotyped with *ResourceServce*. The resource type determines what services the resource provides, and thus defines the range of what resource services that are valid to specify for a resource.

ResourceRequires

Resource requirements are modeled using stereotyped dependencies. A dependency is specified between a *Client* and a *ResourceService*. The dependency alone only states that the client requires the resource service. Using a QoS constraint (a constraint stereotyped *QoSRequires*), the resource requirement can be further detailed with QoS requirements. This constraint is connected with the resource requirement dependency.

4.6.1 Resource QoS Modeling

We use a subset of the UML Profile for QoS [23] as QoS vocabulary for resources. QoS characteristics of resource services and QoS constraints involving these can be epxressed using the profile. The definition of QoS characteristics is part of the resource type catalogue (Section 4.3.4).

Each resource type implies a set of resource services with specific QoS characteristics. When adding QoS constraints to resource dependencies, we must use values that comply with the QoS characteristic of the resource service. It is an assumption that the QoS characteristics of the resource types exist in the model.

QoSRequired

Resource QoS requirements are specified using a constraint with a *«QoS-Required»* stereotype. We use OCL to express the constraint, where the QoS characteristic of the required service is the context. The first variable of the expression is the QoS dimension of the QoS characteristic. Then comes a comparison operator (e.g. > or >=), and at last the value that constrains the resource QoS. The value must of course be in the range of allowed values within the QoS characteristic.

Chapter 5

Model-Driven QoS Mapping

In this chapter we investigate how the resource model can be included in a model-driven development context.

A *resource model* is a model of the resources on a platform. This model can be utilized by defining mappings from other system models. In this chapter we define a framework for model-based mapping to the platform resource model as a means for performing QoS-based resource management.

We first discuss the motivation for mapping to the resource model, and how this can facilitate QoS management. Then, we define a set of requirements for what functionality our QoS mapping framework should provide. We define how mappings to the resource model are defined, and how mappings can be automatically generated from models. Finally, we describe how these mappings are applied on the platform resource model.

5.1 Basic Concepts

We describe the basic terms used in this chapter, along with the intended interpretation in our context.

5.1.1 QoS-Enriched Models

QoS-enriched models are models that include QoS requirements. They are said to be *enriched* because the QoS requirements are supplements to already meaningful models. Thus, the model may function as a structural and functional specification without the QoS requirements.

The QoS elements in the model are specificed orthogonally to the non-QoS elements. This means that the QoS requirements are related to the non-QoS elements, but not the other way around. This orthogonality is a property of the UML profile for QoS, which we use for the QoS part of the model.

5.1.2 Mapping

By *mapping* we mean the transition from the representation of a concept in one context to a representation of the same concept in another context.

The use of *mapping* in this work is concerned with the transition from resource- and QoS-representation in a design-time model to the representation of the same concepts in a run-time resource model.

5.2 Motivation

We try to show how the resource model can be useful as part of a model-driven development process. It is a way of testing the usability of the resource model by performing mappings of resources and QoS from design-time models to run-time resources as represented by the resource model. This involves finding a way of representing the design-time concepts in the resource model.

It also aims to realize a small part of the vision of handling Quality of Service requirements in model-driven development.

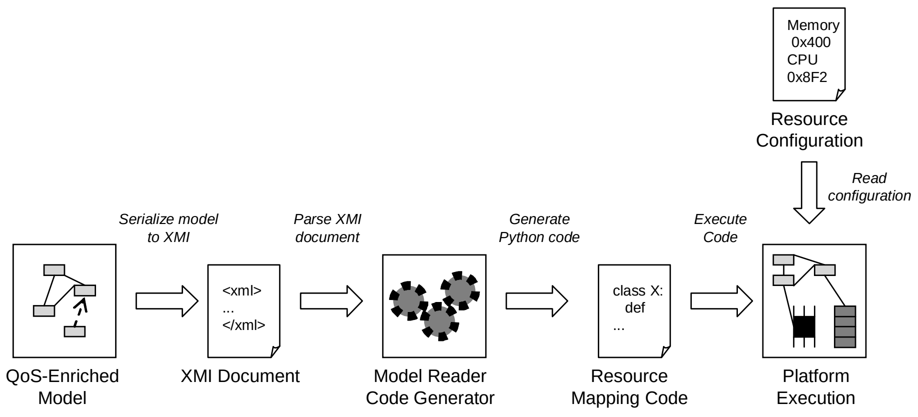
The practical purpose of this mapping is to generate QoS management code in order to ease the development process, especially coding by hand.

5.3 Resource QoS Mapping

We suggest a set of process steps that enable mapping from design-time to run-time resource QoS management. A set of rules for how to perform these steps, along with a description of the implementation of the tools to automate parts of the process, is presented in the following subsections.

5.3.1 Overview

Figure 5.1 Resource Mapping Overview

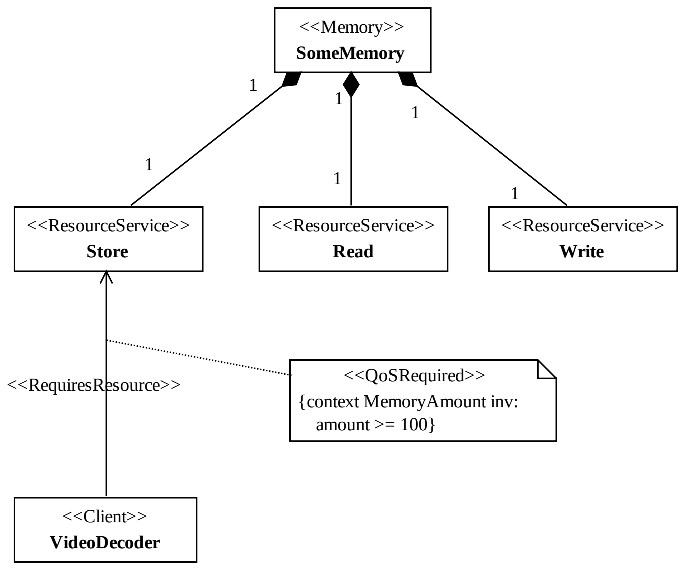


An overview of the resource QoS mapping process is shown in Figure 5.1. It shows the steps from an application model decorated with resource QoS requirements to the execution on a runtime platform that supports resource management.

The model is first serialized as an XMI representation of the model. The XMI document is fed into a new tool, which to generate code. We have divided this tool into two parts; a model reader which parses the XMI data and read it into a set of data structures for further processing, and a code generator that generates code based on this data. The code generator outputs a source code skeleton consisting of Python class definitions and methods for resource management (discovery, authorization, reservation, etc.). Application code is inserted into this skeleton.

When executed on the target platform (the resource model), the elements of the source model are mapped to runtime entities on the platform. In order to initialize the runtime resources, the resource model depends on a resource configuration file.

Figure 5.2 Source Model Example



5.3.2 Source Model

The source model is an application model enriched with resource QoS requirements. A simple example is shown in Figure 5.2. It features a component with a single resource requirement. It has been modeled using two UML profiles; the UML Profile for QoS [23] and the UML Profile for Resource Modeling, presented in Chapter 4.

Our QoS mapping method assumes that this model is designed in a specific way, and constraining how the UML profile constructs are used and interpreted.

Resource requirements are specified as dependencies stereotyped with *«ResourceRequires»*, where a client depends on resource services. Only classes with stereotype *«Client»* are allowed to require resources.

Resources are modeled using classes stereotyped with the type of the resource (such as *«Memory»* or *«Processor»*. The resource type implies the set of services provided by the resources. However, not all resource services must be included in the model, only those involved in a resource requirement. Both the resource and the services required by clients must be included.

Resource services are connected with resources using a 1:1 composition relationship. Resource services are classes stereotyped *«ResourceService»*, and the class name is the name of the service.

A resource requirement is further detailed using a constraint with stereotype *«QoSRequired»* (from the QoS profile). The constraint is expressed using an OCL expression whose context is the QoS characteristic describing how the resource service usage is measured. In Figure 5.2, *MemoryAmount* describes the QoS of a *Memory* resource. The expression compares the QoS dimension of the QoS characteristic with a QoS value, stating how much of the resource is required.

Interpretation

We interpret the model in Figure 5.2 in the following way:

The client *VideoDecoder* poses a requirement on the *Store* service of a *Memory* resource. It needs 100 kb or more from the resource service capacity. (We know that the measuring unit is *kb* because it is part of the characteristic *MemoryAmount*).

5.3.3 Model Reader

The model reader reads a QoS-enriched model into a data structure that facilitates code generation. It only deals with the QoS- and resource-parts of the model, in order to generate code for resource QoS management.

Before the source model can be read, it must be transformed into a supported format. We use XMI because this is a standard format for UML models serialized to XML. It is fairly modeling tool-independent, and supported by most tools, including Objecteering, which we have used.

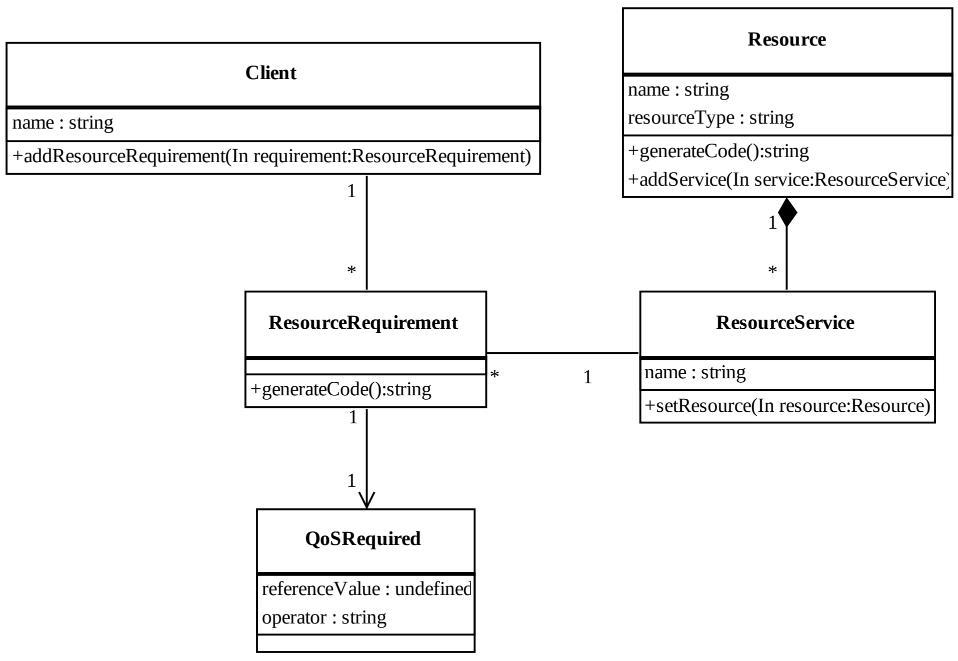
Preconditions

In order for the model reader to be able to process the input model correctly, the model must be well-formed according to the rules described in Section 5.3.2. The QoSProfile implementation presented in Chapter 4 must also be available and used by the model reader.

Implementation

Figure 5.3 shows the classes and relationships used to store the information gathered from the model. We leave the description of the `generateCode` methods to Section 5.3.4.

Figure 5.3 Model Reader



Client represents a single client. A client has a name, and may have a set of resource requirements. These are associated with the client using `addResourceRequirement`. A *Resource* has a name and a resource type. Its resource services can be included using `addService`. Likewise, a *ResourceService* has a name, and can be assigned to a resource using `setResource`. Finally, a *ResourceRequirement* consists of the client that poses the requirement, the resource service that is required, and a *QoSRequired* constraint (from the UML profile for QoS implementation).

The classes above will be instantiated as the script reads the model from the XMI file. The methods are called when relations between the model elements are discovered.

XMI Parsing

An XMI document is XML, so we may use a standard XML *DOM* parser to read it. We use the `xml.dom.minidom` Python module for this.

The stereotypes are an essential part of the models we want to read. These provide information about how to interpret the QoS- and resource-related model elements, and help us disctinguish these from the remaining model elements.

In XMI, all model elements are uniquely identifed by an `xmi.id` attribute. References to other model elements are expressed with `xmi.idref` attributes, refering to the `xmi.id` of the target elements. We use Python *dictionaries* (often called *dicts*) for intermediate storage while reading the XMI document. It is convenient to use the key for the `xmi.id` and the value for the model element information.

First, we make a dict holding all the stereotypes. The stereotypes can be found using:

```
res = doc.getElementsByTagName('UML:Stereotype')
```

We iterate the result and generate a dict with (<id>, <stereotype name>) mappings. This enables us to inspect the stereotype of a model element by checking `stereotypes[idref]`. The next step is gathering all classes in the model:

```
res = doc.getElementsByTagName('UML:Class')
```

This will return every class in the model, but we are only interested in the classes with stereotype *«Client»*, *«ResourceService»*, or one of the resource types, e.g. *«Network»*. We iterate all the classes in the model, and inspect the stereotype of each one. These are split into three different dicts according to stereotype; `clients`, `resourceservices` and `resources`. The `xmi.id` is the key in each of the dicts, and the value is set in the following way:

- `clients[id] = Client(name)`
- `resourceservices[id] = ResourceService(name)`
- `resources[id] = Resource(name, resourcetype)`

Now that we have instantiated all the resources, resource services and clients, the relations between these and the costraints must be read.

We need a relation between resources and their provided services (in order to check what resource a service belongs to). This is done by getting all `UML:Association` elements and checking which ones go from a resource to a resource service. The connection is stored the following way (`resourceid` is the `xmi.id` of the resource and `serviceid` is the `xmi.id` of the resource service):

```
resources[resourceid].addService(resources[serviceid])
resourceservices[serviceid].addService(resources[resourceid])
```

If the association is not a resource-service relationship, we check if it has the stereotype *«RequiresResource»*. If so, we insert it in a separate list of resource requirements to be processed later.

We then collect all UML constraints with stereotype *«QoSRequired»*. We assume that this type of constraint only exists for resource requirements in the model. We parse the OCL expression, and create a *QoSConstraint* object to represent it. The (`xmi.id`, QoSConstraint) mappings are put into a new dict.

To relate the QoS constraints with the resources and clients, we iterate the list of resource requirements gathered earlier. For each requirement, we find both ends of the association and get hold of the id of the client and the resource service involved. Also, we find what UML constraints apply to this association. Based on this information, we create a *ResourceRequirement* object:

```
resreq = ResourceRequirement(clients[clientid],
                             resourceservices[serviceid],
                             qosconstraints[constraintid])
```

This requirement is associated with the client using:

```
client[clientid].addResourceRequirement(resreq)
```

Now, the classes in Figure 5.3 hold all the information necessary to start generating code.

Figure 5.4 Example Output Code

```
class VideoDecoder(Client):

    def __init__(self):
        self._reservations = []

    def run(self):
        self._reserveResources()
        # do stuff
        self._releaseResources()

    def _reserveResources(self):
        constraints = []

        qoschar = qosrep.getQoSCharacteristic('MemoryAmount')
        amount = qoschar.makeValue(100)

        constraints.append(('Store', QoSConstraint(amount, 'ge')))

        match = rtrader.lookup('Memory', constraints)
        if not match:
            raise 'NoSuchResourceFound'

        qoschar = qosrep.getQoSCharacteristic('MemoryAmount')
        amount = qoschar.makeValue(100)

        service = match.getService('Store')
        resmngr.reserve(self, match, service, amount)
        self._reservations.append((match, service, amount))

    def _releaseResources(self):
        for reservation in self._reservations:
            resource, service, amount = reservation
            resmngr.release(self, resource, service)

            if resource.isProtected():
                resmngr.free(self, resource)
```

5.3.4 Mapping Code Generator

Based on the data read by the model reader, we are ready to generate code for resource mapping. The method named `generateCode` is responsible for generating the code for each client.

Each client object now contains a list of the requirements it poses, and each requirement consists of a service and constraint, along with the client.

If the client has at least one resource requirement, the method `_reserveResources` is declared. We now need to make a separate list of constraints for each resource that the client requires. These requirements are put into a dict with the *resource* as a key and the list of required services as keys:

```
for req in self.requirements:
    if not resources.has_key(resource):
        resources[resource] = [req]
    else:
        resources[resource].append(req)
```

We now loop through each resource (iterate through `resources.keys()`). The code for each resource should first make a list of constraints to pass to the resource broker in order to discover an appropriate resource. We get hold of the type of the resource, and check whether it is *protected* or not.

We then go through the requirements this client has on the services of the resource. The code generated for each resource consists of two parts; resource discovery (the establishment of QoS constraints on resource services and a lookup request to the resource trader) and resource reservation (if the resource trader returns a matching resource). Since discovery must take place before reservation, we put the reservation code in a separate string in order to append it to the code later.

For each requirement, we must first generate code to build the necessary QoS values:

```
qoschar = qosrep.getQoSCharacteristic('<characteristic_name>')
amount = qoschar.makeValue(<value>)
```

First, we get the QoS characteristic of the resource service from the QoS repository. Then we use this characteristic to instantiate a QoS value. The `<value>` is a basic value, and can be get from the *QoSValue* object of the requirement. We use this code to create QoS values for both resource discovery and resource reservation, so it is both outputted immediately, and put in the reservation code string.

The resource discovery code builds a list of resource service constraints, where each requirement is added in turn:

```
constraints.append(('<service_name>',
                    QoSConstraint(amount,'<operator>')))
```

The `<operator>` is also a part of the QoS constraint associated with the requirement.

Now, we are ready to generate reservation code for the requirement:

```
service = match.getService('<service_name>')
resmngr.reserve(self, match, service, amount)
self._reservations.append((match, service, amount))
```

The `match` is the resource returned from the resource broker (if a matching resource was found). We then reserve the specific amount of the matching resource, and finally we append it to the client's list of reservations in order to keep track of what's reserved. The `amount` is the QoS value declared on beforehand.

The code for expressing the list of constraints to the resource broker is already generated. We now add the lookup request:

```
match = rtrader.lookup(self, '<resource_type>', constraints)
```

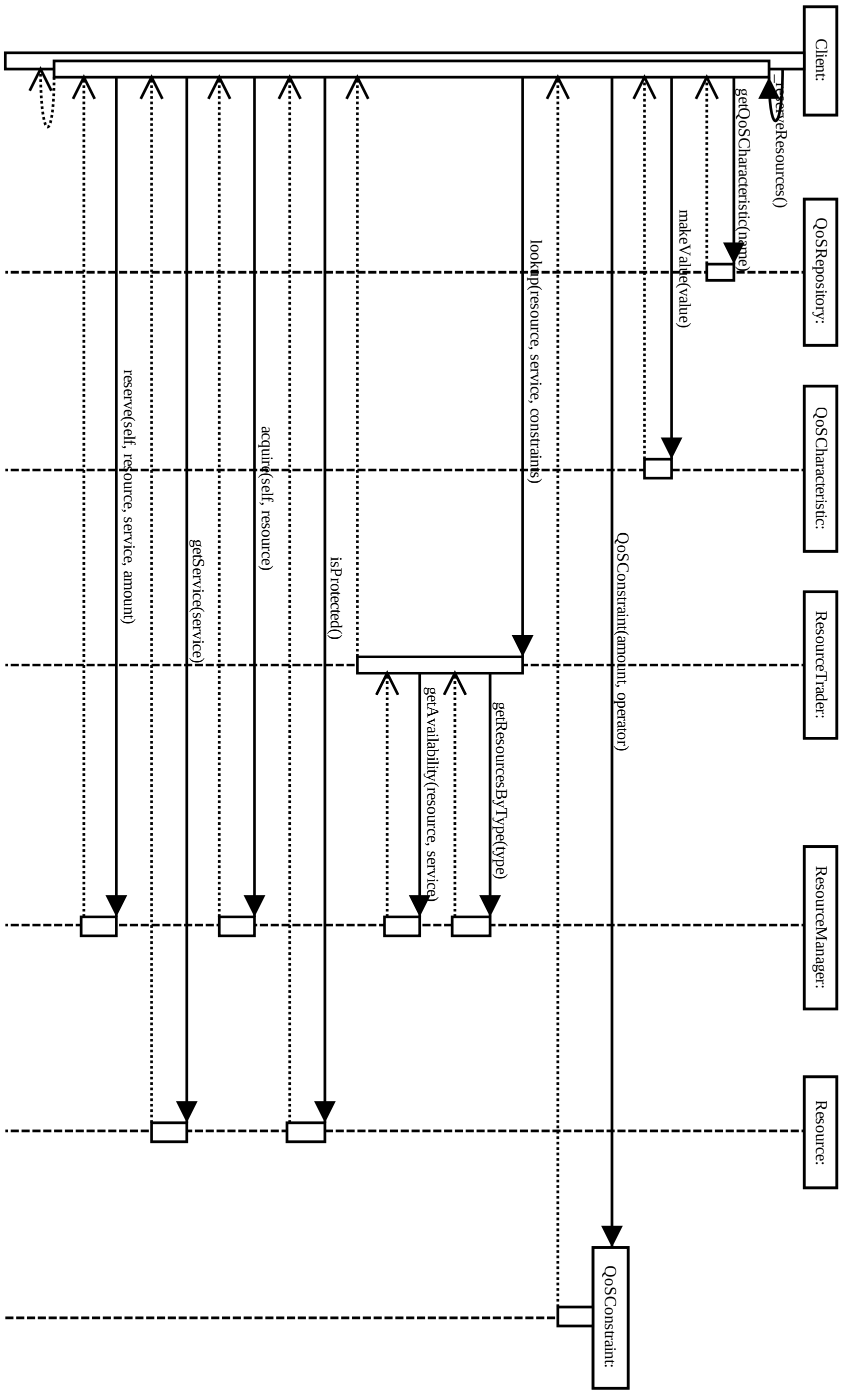
We also supply code to raise an exception if no match is found. At last, we append the code to perform the actual reservations. The code for all clients can now be generated by calling the `generateCode` method of all objects in the `clients` list.

The code generator also adds a method `_releaseResources` to the clients with resource requirements. This code is common for all clients, as seen in Figure 5.4.

5.4 Resource Model Execution

We illustrate the behaviour of the generated code produced by the code generator in two sequence diagrams (Figure 5.5 and Figure 5.6). These illustrate behavoiur of the `_reserveResources` and `_releaseResources` methods.

Figure 5.5 Resource Reservation



Resource Reservation

After `_reserveResources` is called, we first get the QoS characteristics needed to quantify resource requirements. Given the name of the QoS characeteristic, the *QoSRepository* can return the characteristic object. The name is known by the code generator, and is included in the output code. The QoS characteristic objects have a `makeValue` method that returns the QoS value(s) needed for defining resource constraints.

Resource constraints are represented as a list of `(resource service, QoSConstraint)` tuples. For each constraint, a *QoSConstraint* object is instantiated using the QoS value.

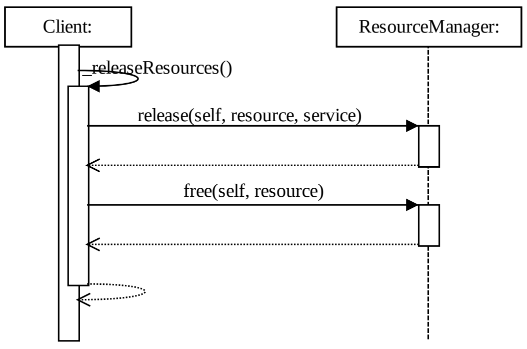
Now that we have the required resource type and a list of constraints on its services, we are ready to perform a *lookup* in the *ResourceTrader*. The resource trader queries the *ResourceManager* for all resources of the needed type. It also queries the manager for the availability of each resource (the amount available for reservation). If available, the resource trader returns a matching resource to the client.

If the resource type in mind is protected, it must be acquired by the client. Acquirement is the responsibility of the resource manager. Before performing the reservation, we need a pointer to the resource service instance. Since we already have the name, we can get it by calling `getService` on the resource object. Finally, the reservation is performed, with the client, resource, service, and amount as parameters.

Resource Release

The execution of the `_releaseResource` method is much simpler. It tells the resource manager to release the entire amount of each resource. For the protected resources, it also calls the `free` method to make it available to others.

Figure 5.6 Resource Release



Chapter 6

An Application Development Case

In this chapter we try out our proposed approach in a case study, involving a part of the development of a QoS-sensitive application. By trying it out on this development example, we hope to get some clues about the usefulness of the approach, as well as weaknesses and ideas about what could be done to make more useful.

We chose an audio conferencing application for the case study, since it is relatively simple to model and generally well understood. It can be modeled as a set of components with individual resource requirements in a straight-forward way. The components are typically distributed among nodes. Thus, it matches our development approach well.

We do not try to create a complete model in any way. Instead, we look only at how it can be modeled in terms of components and resource QoS requirements.

6.1 Component Modeling in UML

UML 1.5 and prior versions have very limited support for the modeling of software component. This has been addressed in UML 2.0 [27]. It introduces a *Compenent* package, and offers modeling elements for component modeling and new types of diagrams for component models.

Even though UML 2.0 has better support for components, the QoS mapping approach presented in Chapter 5 is based on UML 1.5. Instead of using the new component elements, we represent components with classes. Thus, we model class structure and not component instances. In UML 2.0, one can express interfaces provided by a component as *facets* and interfaces required as *receptacles*. Since this cannot be expressed directly in UML 1.5, we model provided interfaces using UML interface elements connected to the components with an association relationship. Components requiring other components

In accordance with our QoS mapping approach, we model components as classes stereotyped with *«Client»*. Client is a general term that represents entities that use resources. In this case we assume that the components are the users of resources.

6.2 Application Model

Figure 6.1 Audio Conference Application

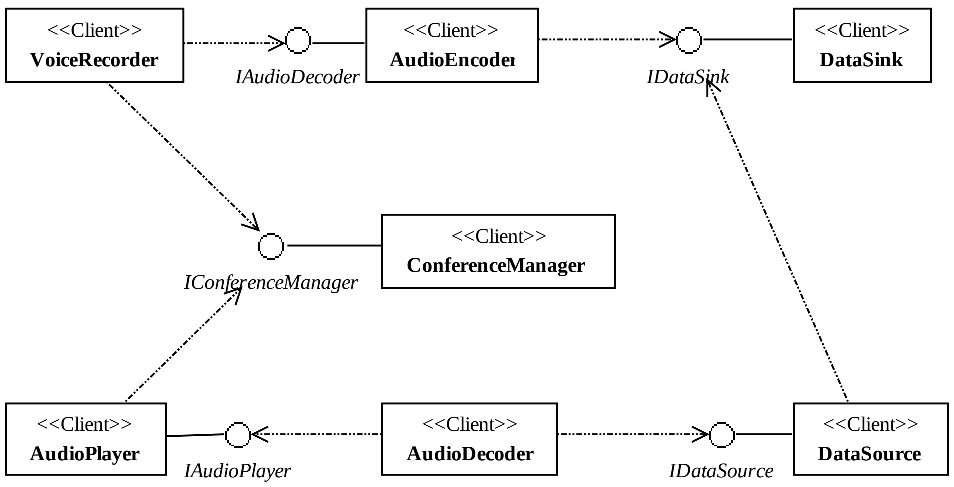


Figure 6.1 shows a model of the components involved in the audio conference application. The relationships between the components are only modeled as dependencies to interfaces provided by other components. Thus, information about multiplicities or anything instance-related is not considered.

The *VoiceRecorder* is responsible for recording voice input from the audio conference participants. This component must be present at the nodes of each participant.

The *VoiceRecorder* uses the *AudioEncoder* component to encode the recorded audio to an appropriate format, in order to reduce the amount of data needed to be transfered.

A *DataSink* component is responseible for transmitting the audio data over a network. It is used by the *AudioEncoder*. The *DataSource* component is the receiving end of the audio connection, and receives data from the *DataSink*.

The *DataSink* provides an interface to the *AudioDecoder* for receiving the audio data to be decoded. Finally, the audio can be played back using an *AudioPlayer* component.

The components mentioned above model sound recording, audio streaming over a network, and finally, audio playback. An audio conference application must support these features, and they must be supported both ways; a listener is also a recorder, and vice versa. It could also support multiple participants. The *ConferenceManager* component controls the audio flows among participantants by providing an interface to the recording- and playback-components. It managemes the setup and termination of conferences, and control the routing of audio among the participants.

6.3 Resource QoS Models

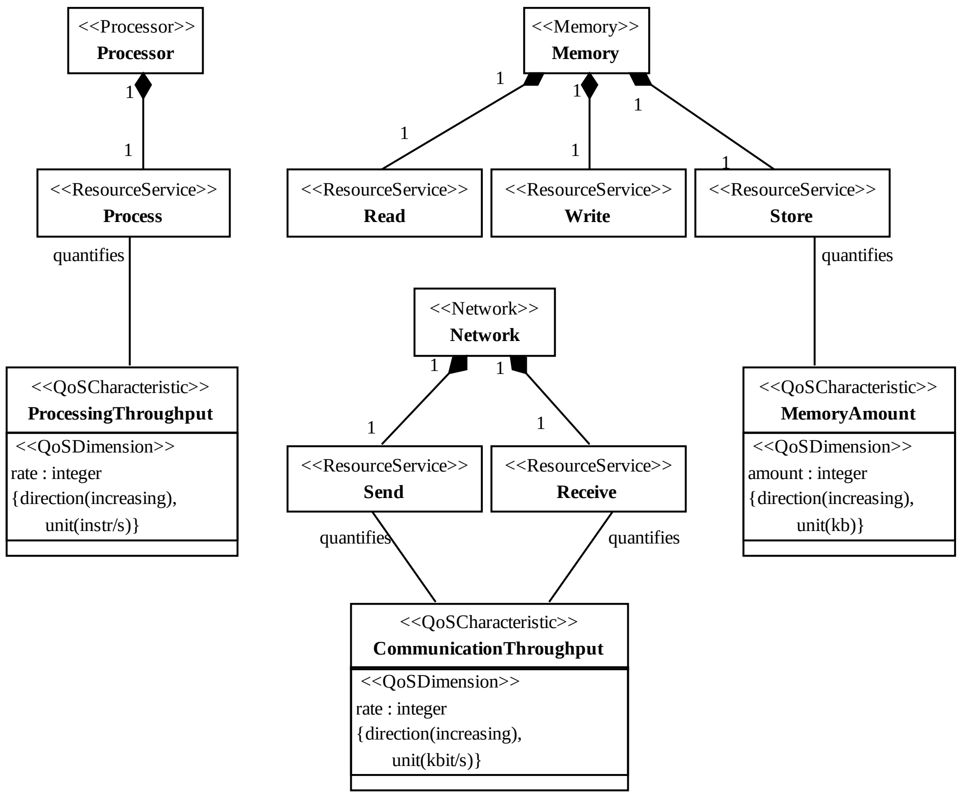
The application model in Figure 6.1 also includes a set of resource requirements that involve QoS constraints. Resource reuqirements are exposed by the clients, and involve resource services. These requirements are modeled using the UML profile for resource modeling specified in Chapter 4, as well as the UML profile for QoS [23].

In general, every piece of computer code, such as a component, will need at least a little bit of processing power and a little bit of memory, and typically some other resources as well. In order to keep this development case simple, we only model the resource requirements that put a significant strain on the resources and raise a real need for resourcee management.

Since our approach assumes the existence of a global catalogue of QoS characterististcs and resource types, we present a portion of this catalogue used in this chapter in Figure 6.2. Like all resource types and

QoS characteristics in the catalogue, these are also available to the code generator and resource model implementation.

Figure 6.2 Resource Types and QoS Characteristics



<<Processor>> Processor
<<Memory>> Memory
<<ResourceService>> Process
<<ResourceService>> Read
<<ResourceService>> Write
<<ResourceService>> Store
quantifies
quantifies
<<Network>> Network
<<QoSCharacteristic>> ProcessingThroughput
<<QoSDimension>> rate : integer {direction(increasing), unit(instr/s)}
<<ResourceService>> Send
<<ResourceService>> Receive
<<QoSCharacteristic>> MemoryAmount
<<QoSDimension>> amount : integer {direction(increasing), unit(kb)}
quantifies
quantifies
<<QoSCharacteristic>> CommunicationThroughput
<<QoSDimension>> rate : integer {direction(increasing), unit(kbit/s)}
1

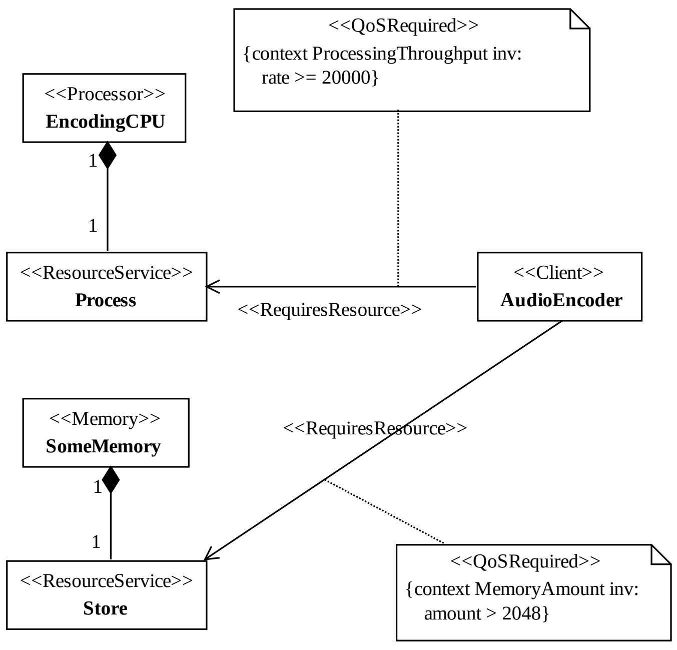
We use associations between resource services and QoS characteristics to express that the characteristics quantify the services. The *Read* and *Write* services of the *Memory* resource have no associated QoS characteristic because these services are not considered in the case.

In the UML profile for resource modeling, resource requirements are modeled using dependency relationships with stereotype *«RequiresResource»*. Due to problems with adding constraints to UML dependency elements in the modeling tools, we modeled resource requirements using association elements instead.

AudioEncoder

The reosource requirements of the *AudioEncoder* is shown in Figure 6.3. This component requires two types of resources, *Processor* and *Memory*. It requires that the processor resource is able to process at least 20000 instructions per second. The required memory resource must be able to store more than 2048 kilobytes. The QoS characteristics that represent the context of the *«QoSRequired»* expression are seen in Figure **??**.

Figure 6.3 QoS Model for AudioEncoder Component



DataSink and DataSource

The responsibility of the *DataSink* component is to send audio data over the network. To do this, a *Network* resource is required that is able to *send* more than 645 kilobits per second.

The *DataSource* is the recevier of the audio data sent by the *DataSink* component. It requires a *Network* resource able to *receive* more than 645 kbit/s.

Figure 6.4 QoS Model for DataSink Component

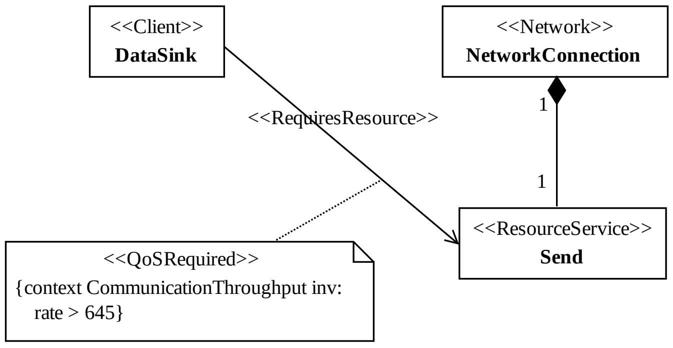
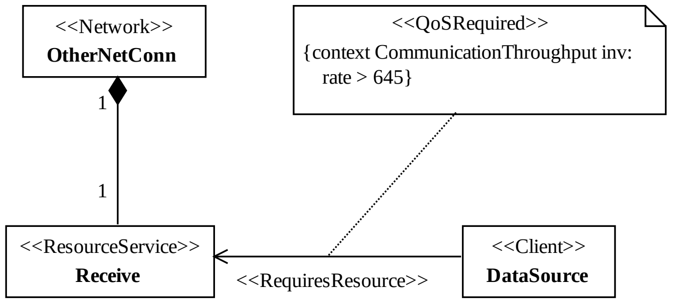


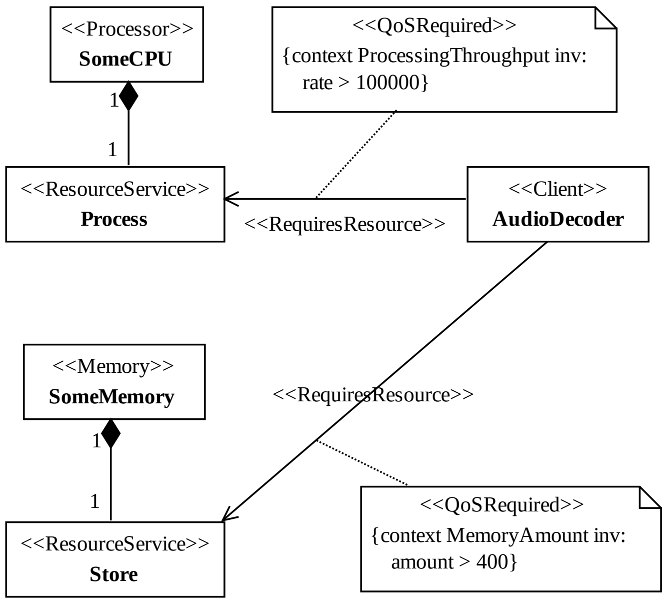
Figure 6.5 QoS Model for DataSource Component



AudioDecoder

The *AudioDecoder* component poses requirements on a *Network*- and a *Memory*-component. It requires more than 100 000 instructions per second of processing throughput and more than 400 kilobytes of storage in memory.

Figure 6.6 QoS Model for AudioDecoder Component



6.4 Code Generation

The resource QoS models from the former sections are automatically transformed into code that maps the QoS specifications onto a platform-level resource model.

The model presented in the former section was serialized to XMI using the Objecteering/UML XMI module. It is then fed into the code generator. The output is a Python code skeleton containing declaration of classes representing the clients, and methods to support resource management and reservation in order to realize the modeled resource requirements.

The output from the code is included in Appendix A. We here describe very briefly what the code does. Each *«Client»* in the model resulted in a class with the name of the client. It is declared as subclass of Client. The interesting method of each client is `_reserveResources`. In *Data-Source*, this method first performs a lookup using the resource trading, for a *Network* resource. The constraints given states that *Receive* service that can deliver more than 645 kbit/s is required. If no such resource is found, an exception is raised. Else, the resource manager is told to reserve 645 kbit/s of the *Receive* service of the *Network* resource.

The *AudioEncoder* class first does a resource trader lookup for a *Memory* resource having a *Store* service that can store more than 2048 kb. Just like the *DataSource*, it raises an exception if now match is found, or else continues to reserve the specified amount. Then it performs a lookup for a *Processor* resource that can process at least 20000 instuctions per second.

The *AudioDecoder*'s `_reserveResources` method is almost identical to that of the *AudioEncoder*, in that it requires the same resource services. Unlike *AudioEncoder*, it requires less of both.

The *ConferenceManager*, *VideoRecorder* and *AudioPlayer* methods do not contain the `_reserveResources` and `_releaseResources`, but only declare the `__init__` constructor method.

6.5 Reosurce Model Execution

By including some text output in the resource model implementation, we can see what happens when the generated code is run.

Simplifi cations

The following simplifications have been used for the execution environment environment of clients:

- All clients exist within the same address space. This simplifies the resource configurations and deployment of the code. A real audio conference application would be of little use if all components executed on the same node. However, it is sufficient to demonstrate mapping to a resource model.

- There is only a single resource manager available in this address space, that manages all available resources.
- The code skeletons created by the code generator have not been enriched with application logic. This is sufficient because the resource management code is the interesting part for resource mapping.

Resource Confi guration

A resource configuration to be read by the resource model has been provided. A part of the configuration for a network resource is shown in Figure 6.7.

Figure 6.7 Resource Configuration

```
<Resource>
  <ResourceType>Network</ResourceType>
  <ResourceService>
    <Name>Send</Name>
    <Capacity>1500</Capacity>
    <Location>Unspecified</Location>
  </ResourceService>
  <ResourceService>
    <Name>Receive</Name>
    <Capacity>3000</Capacity>
    <Location>Unspecified</Location>
  </ResourceService>
</Resource>
```

The resource configuration specifies the resources available to a resource manager. We have left the location *unspecified* because these resource type implementations are not mapped to real resources. As well as the resources depicted in Figure 6.7, the configuration includes a *Memory* resource having a *Store* service with capacity 512000 kb, and a *Processor* resource with processing capacity 2000000 instrunctions per second.

Execution

In order to observe what is going on in the resource model, we have implemented test outputs when doing resource trader lookups, reservation of resources and release of resources. An excerpt of the test output is shown if Figure 6.8. It shows the execution of the code generated for the *AudioEncoder* client.

Figure 6.8 Test Output

```
Looking for a Memory resource with constraints:
  Store service with at least 2048 kb
Memory resource found with services:
  Store service with available 512000 kb
Reserved 2048 kb of Store service (Memory resource)
Looking for a Processor resource with constraints:
  Process service with at least 20000 instr/s
Processor resource found with services:
  Process service with available 2000000 instr/s
Reserved 20000 instr/s of Process service (Processor resource)
Released 2048 kb of Store service (Memory resource)
Released 20000 instr/s of Process service (Processor resource)
```

The resource QoS requirements of the audio conference application components were all successfully transformed to resource trader lookups and reservations, and later released. Since none of the resources involved in the case were protected, there were no *acquire* or *free* requests to the resource manager.

Chapter 7

Evaluation

In this chapter we discuss and evaluate the design and implementation of the resource model presented in Chapter 4 and the resource QoS mapping framework in Chapter 5. We compare the results with our own requirements, and requirements posed by others in solutions of similar problems.

The case study in Chapter 6 is also an input to the evaluation, as it involves experimentation with the resource model and resource mapping method. In particular, it gives an indication of the usefulness of the approach suggested in this work.

Finally, we discuss how the results answer our research problem, and how it contributes to the problem domain.

The research method used in mainly qualitative, involving model- and software-construction and evaluation through comparison with certain design goals and requirements, including related work.

7.1 Evaluation of the Resource Model

In this section we discuss how the design of the resource model fulfils the requirements and goals stated in Chapter 4.

Generic

The first design goal of the resource model was that it should be *generic*. By this we mean support for many different types of resources, and support many different usages.

We believe that this goal has been met, since the resource model is not tied to any particular platform or technology. We assume it could be implemented on a wide range of platforms and be a tool for resource management. We also believe the resource model can support any type of resource, as long as there exists some way to monitor it and control its usage, and a resource type implementation exist. With the semantics of the UML profile for QoS [23] we use QoS values to quantify resource usage, both at design-time and run-time. Since the UML profile is very general, we are able to express any thinkable QoS statement about resources. The prototype implementation of the QoS profile is not complete, but this is due to simplification, not technical limitations. A more complete implementation could be created if necessary.

The UML profile for resource modeling supports representation of the resource model concepts in UML models. This profile can be used to model any type of resource that may exist on a run-time platform, and associated QoS requirements.

Resource Sharing Control

A requirement of the resource model is to support control over the resource sharing in a predictable way. We have not included resource sharing mechanisms as part of the resource model, but instead that such a resource model can be built as a layer on top of the resource model. Resource reservation can be used to guarantee the availability of resources to clients. Thus, reservation can be used as a mechanism to implement resource sharing. It does not advocate any particular methods or policies for how to share resources among clients.

Extensible

It is a requirement that the resource model is extensible. By *extensibility* we understand the ability to support new resource types that might evolve, for example due to the emergence of new hardware devices. This is supported by the ability to add new resource types by supplying an implementation for controlling it. A mapping function between the actual resource status and the QoS value as described by the QoS characteristic(s) of the resource must also be supplied. As mentioned, we assume the QoS profile support is general enough to support new kinds of measurements.

Uniform

A uniform resource model provides a uniform way of representing resources. We believe that hiding the heterogeneity of resources reduces complexity and thus eases the usage. Uniformness is also a trade-off with expressiveness. We must be able to express enough for the resource model to be useful, but nothing more than we need.

We think that this goal to a large extent has been met. From a certain level of abstraction, all resources can be viewed a unit with a name, a type, and a set of services, each with a QoS value representing its offered QoS described by a QoS characteristic. Making all resource quantification QoS values enables us to have common interfaces for the management of all resource types. The implementation details of each type is hidden from the resource model's point of view. A property that can not be hidden is the *protection* property; a protected resource must use the `acquire` and `free` methods while unprotected resources do not.

Deign-time/Run-time Consistency

In order to support resource QoS mapping, we require the presence of a consistent vocabulary at design-time and at run-time. This implies that design-time concepts can be represented at run-time and vice versa.

The implementation of parts of the UML profile for QoS enables a common QoS vocabulary for resources. This allows the modeler to use OMG standards for QoS specifications (when the standardization process of the UML profile is completed). The UML profile for resource modeling presented in Chapter 4 provides UML extensions for modeling resources in UML models using the resource model concepts.

Having a global repository of resource types and QoS characteristics allows us to speak of the same objects, typically referenced by their name, at design-time and run-time. In particular, the resource type vocabulary allows mapping from type-level to instance-level resources, thus facilitating automated resource discovery. This is possible since the resource type identifies the behaviour of the resource. Because of this, several resource instances may be able to perform the same task.

However, the resource model lacks support for modeling resource instances at design-time. This makes it unsuitable for real-time analysis techniques, which is a typical application of the resource model in the UML profile for schedulability, performance, and time [24]. Neither does it support resource modeling in UML behavioural modeling, such as in sequence diagrams.

Support for Access Control Policies

Access control for resources is handled by the resource manager. We have implemented support for very simple policies in the resource model prototype, covering how many clients may use a resource at once.

RM-ODP Resource Model

The resource model in *Open Distributed Processing - Reference Model - Quality of Service* [15] describes a set of properties of a resource model. In order to comply with RM-ODP, these properties should be supported by our resource model.

The RM-ODP resource model involves QoS contracts for resources. We do not model such contracts, but provide a QoS vocabulary and resource management system that would support the implementation of QoS contracts.

A second requirement of the RM-ODP resource model is *recursiveness*. Resources can encapsulate other resources, creating a hierarchy of dependency relationships. *Logical resources* in our resource model can encapsulate other resources. Logical resources are both resources and clients. The implementation of a logical resource type is responsible of managing (such as reserving and acquiring) the resources it uses.

Other Resource Model Requirements

In his Ph.D. thesis [11], Hector Duran-Limon expresses a set of requirements about resource management in middleware. We discuss some of these requirements with respect to our resource model.

The first requirement is the support of *various levels of abstraction*. This can be supported by implementing logical resource types on top of other resource types.

The second requirement states that resource management must be *configurable* in order to support deployment on a platform. This requirement is met in the resource model, as resource types are implemented independently of deployment. A separate resource configuration in each address space holds the information necessary to map resource types to resource instances.

Resource reconfiguration is another requirement, refering to dyanmic reconfiguration of resources at runtime. Since we have not considered resource management over time on the resource model, this requirement is beyond our scope. However, a mechanism for resource reconfiguration could presumably be built as a layer above the resource model. The reconfiguration manager could reserve and release resources for the different clients dynamically to meet QoS requirements.

Duran-Limon further requires support for *evolution*, refering to extensibility as discussed above, as well as *good performance*. We have no basis for discussing the performance of our resource model, but we believe the performance is largely dependent on how the resource types are implemented. We are not aware of any design decisions that should hinder good performance.

Prototype Implementation

A prototype of the resource model was implemented in Python. The implementation work has shown that the resource model design could be implemented without major modifications. The implementation of a small subset of the UML profile for QoS showed how some basic QoS characteristics could be used at run-time.

In order to demonstrate its usefulness in managing real-world resources, it would have been appropriate to implement a resource type for a physical resource. Instead, we implemented a logical resource type (license pool) with virtaully no physical underpinnings as an example resource type. However, we think we have provided enough evidence to prove that resource types can manage real resources provided that the subsystem does not hide too much of the access to them.

7.2 Resource QoS Mapping

In the problems tatement in Section 1.2, two of the subproblems addressed were:

- How resource QoS requirements can be modeled at design-time in a such way that they can be handled later in the development process.
- How to transform the resource QoS requirements expressed in the model to a form that is manageable by the target platform.

In Chapter 3, we discussed in a more general way how QoS requirements can be handled in model-driven development. In this context, how have we contributed to this challenge?

Backed by the resource model, the resource QoS mapping approach presented in Chapter 5 is the main contribution in investigating these problems.

We have showed how resource-level QoS requirements can be modeled in UML as part of application models using a UML profile for QoS. Even though applications may have many QoS requirements that are not directly related to resources, resource QoS must still be considered. Some research work in QoS management in component architectures has related resource QoS specifications to components (e.g., [30]).

In some applications, the QoS of interest can be more or less directly mapped to resources. For example, the speed of a download service is closely related to the QoS of a network resource.

As discussed in Chapter 3, it is desirable to include QoS mapping from higher to lower absctraction levels in PIM-to-PSM model transformation. In this context, low-level QoS such as QoS for resources will be part of the PSM. Thus, our approach can support the part of this process that involves transforming the PSM to code and to the platform-level resource model.

The code generated from QoS-enriched models map the modeling concepts to the resource model on that platform on which it is executed. The resulting classes generated for each client provides one method for reserving all necessary resource capacity and one for releasing all reservations. This demonstrates the mapping from the UML model to the resource model, but supports a very static kind of QoS management. A possible extension for supporting more dynamic QoS management is to model different resource requirements for the same elements (e.g., compoments), representing different operational models. This information could be used to generate methods for resource reconfiguration.

The case study in Chapter 6 gave us an impression of the usefulness of our resource QoS mapping approach. While modeling the audio conference application, it was fairly easy to identify resource QoS requirements of each component. This is likely to be the case of several types of application. There was generated a considerable amount of code from the application model, which would have been a time-consuming effort to write by hand. The generation of code and execution on the resource model demonstrated the ability to map design-time QoS requirements onto a run-time resource model. Even though we did not develop a complete audio application, the experiences gained from modeling and automatic transformation gave us the impression usefulness. Being able to identify QoS requirements at design-time, and not having to worry about these when writing application code seems to be very useful.

Chapter 8

Conclusion

In many applications, Quality of Service concerns are critical, and must be specified and managed in the system. It is argued by many that QoS should be managed at the middleware level, and be specified independently of application logic.

Model-driven development raises the abstraction level of models above the implementation technology, allowing platform-independent application models. It is envisioned automatic transformation from platform-independent to platform-specific models, mapping the application logic to platform-specific concepts.

The emergence of middleware platforms with QoS management support makes it feasible to include modeling of QoS requirements in model-driven development. This would allow modeling QoS independently of implementation platform, and make QoS mapping to a form realizable on the platform a part of model transformation.

We have in this thesis shown how to realize a part of this vision, by focusing on *resource QoS*. It is solved by modeling QoS requirements on resources as part of application models and performing automatic mapping of these models to a platform-level resource model. This way, the QoS model is realized on the target platform.

Resource QoS requirements are only a subset of the QoS requirements of interest in most application. Nevertheless, resource management is a concern of virtually any QoS-aware system, and a typical target of QoS mapping. Therefore, we believe that the approach presented in this thesis is useful in the handling of QoS in model-driven development in general.

The transformation of platform-specific models to code is one place where this approach could be implemented, making resource QoS a part of the PSM and support automatic mapping to a platform-level resource model.

The resource model suggested in this work seems to provide the expressibility necessary to represent and manage all kinds of resources.

Appendix A

Generated Code

This appendix contains the QoS mapping code generated from the audio conference model in Chapter 6.

```
class DataSource(Client):

    def __init__(self):
        self._reservations = []

    def run(self):
        self._reserveResources()
        # do stuff
        self._releaseResources()

    def _reserveResources(self):
        constraints = []

        qoschar = qosrep.getQoSCharacteristic('CommunicationThroughput')
        amount = qoschar.makeValue(645)

        constraints.append(('Receive', QoSConstraint(amount, 'gt')))

        match = rtrader.lookup('Network', constraints)
        if not match:
            raise 'NoSuchResourceFound'

        qoschar = qosrep.getQoSCharacteristic('CommunicationThroughput')
        amount = qoschar.makeValue(645)
        service = match.getService('Receive')
        resmngr.reserve(self, match, service, amount)
        self._reservations.append((match, service, amount))

    def _releaseResources(self):
        for reservation in self._reservations:
            resource, service, amount = reservation
            resmngr.release(self, resource, service)

            if resource.isProtected():
                resmngr.free(self, resource)

class AudioEncoder(Client):

    def __init__(self):
        self._reservations = []

    def run(self):
        self._reserveResources()
        # do stuff
        self._releaseResources()

    def _reserveResources(self):
        constraints = []

        qoschar = qosrep.getQoSCharacteristic('MemoryAmount')
        amount = qoschar.makeValue(2048)

        constraints.append(('Store', QoSConstraint(amount, 'gt')))

        match = rtrader.lookup('Memory', constraints)
        if not match:
            raise 'NoSuchResourceFound'

        qoschar = qosrep.getQoSCharacteristic('MemoryAmount')
        amount = qoschar.makeValue(2048)

        service = match.getService('Store')
        resmngr.reserve(self, match, service, amount)
        self._reservations.append((match, service, amount))
        constraints = []

        qoschar = qosrep.getQoSCharacteristic('ProcessingThroughput')
        amount = qoschar.makeValue(20000)

        constraints.append(('Process', QoSConstraint(amount, 'ge')))

        match = rtrader.lookup('Processor', constraints)
        if not match:
            raise 'NoSuchResourceFound'

        qoschar = qosrep.getQoSCharacteristic('ProcessingThroughput')
        amount = qoschar.makeValue(20000)

        service = match.getService('Process')
        resmngr.reserve(self, match, service, amount)
        self._reservations.append((match, service, amount))

    def _releaseResources(self):
        for reservation in self._reservations:
            resource, service, amount = reservation
            resmngr.release(self, resource, service)

            if resource.isProtected():
                resmngr.free(self, resource)

class DataSink(Client):

    def __init__(self):
        self._reservations = []

    def run(self):
        self._reserveResources()
        # do stuff
        self._releaseResources()

    def _reserveResources(self):
        constraints = []

        qoschar = qosrep.getQoSCharacteristic('CommunicationThroughput')
        amount = qoschar.makeValue(645)

        constraints.append(('Send', QoSConstraint(amount, 'gt')))

        match = rtrader.lookup('Network', constraints)
        if not match:
                raise 'NoSuchResourceFound'

            qoschar = qosrep.getQoSCharacteristic('CommunicationThroughput')
            amount = qoschar.makeValue(645)

            service = match.getService('Send')
            resmngr.reserve(self, match, service, amount)
            self._reservations.append((match, service, amount))

    def _releaseResources(self):
        for reservation in self._reservations:
            resource, service, amount = reservation
            resmngr.release(self, resource, service)

            if resource.isProtected():
                resmngr.free(self, resource)


class ConferenceManager(Client):

    def __init__(self):
        self._reservations = []

class VoiceRecorder(Client):

    def __init__(self):
        self._reservations = []

class AudioPlayer(Client):

    def __init__(self):
        self._reservations = []

class AudioDecoder(Client):

    def __init__(self):
        self._reservations = []

    def run(self):
        self._reserveResources()
        # do stuff
        self._releaseResources()
    def _reserveResources(self):
        constraints = []

        qoschar = qosrep.getQoSCharacteristic('MemoryAmount')
        amount = qoschar.makeValue(400)

        constraints.append(('Store', QoSConstraint(amount, 'gt')))

        match = rtrader.lookup('Memory', constraints)
        if not match:
            raise 'NoSuchResourceFound'

        qoschar = qosrep.getQoSCharacteristic('MemoryAmount')
        amount = qoschar.makeValue(400)

        service = match.getService('Store')
        resmngr.reserve(self, match, service, amount)
        self._reservations.append((match, service, amount))
        constraints = []

        qoschar = qosrep.getQoSCharacteristic('ProcessingThroughput')
        amount = qoschar.makeValue(100000)

        constraints.append(('Process', QoSConstraint(amount, 'gt')))

        match = rtrader.lookup('Processor', constraints)
        if not match:
            raise 'NoSuchResourceFound'

        qoschar = qosrep.getQoSCharacteristic('ProcessingThroughput')
        amount = qoschar.makeValue(100000)

        service = match.getService('Process')
        resmngr.reserve(self, match, service, amount)
        self._reservations.append((match, service, amount))

    def _releaseResources(self):
        for reservation in self._reservations:
            resource, service, amount = reservation
            resmngr.release(self, resource, service)

            if resource.isProtected():
                resmngr.free(self, resource)
```

Bibliography

[1] Anders Andersen. *OOPP, A Reflective Middleware Platform including Quality of Service Management*. Dr. sci. thesis, Department of Computer Science, University of Tromsø, Tromsø, Norway, February 2002.

[2] Arnor Solberg, Jon Oldevik, Jan Øyvind Aagedal. A Framework for QoS-Aware Model Transformation, Using a Pattern-Based Approach . 2004.

[3] Arnor Solberg, Knut Eilif Husa, Jan Øyvind Aagedal, Espen Abrahamsen. QoS-aware MDA, 2003.

[4] Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003.

[5] Lothar Baum and Thorsten Kramp. Towards a uniform modeling technique for resource-usage scenarios. In *PDPTA*, pages 1324–1329, 1999.

[6] P. Desfray. White paper on the profile mechanism. OMG Document ad/99-04-07, April 1999.

[7] Hector A. Duran-Limon, Gordon S. Blair, and Geoff Coulson. Adaptive resource management in middleware: A survey. *IEEE Distributed Systems Online*, 5(7), 2004.

[8] Marie-Pierre Gervais. Towards an mda-oriented methodology. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, pages 265–270. IEEE Computer Society, 2002.

[9] Object Management Group. Meta object facility (mof) specification, version 1.4. OMG Specification formal/02-04-03, Object Management Group, 2002.

[10] Steffen Göbel, Christoph Pohl, Simone Röttger, and Steffen Zschaler. The comquad component model: Enabling dynamic selection of implementations by weaving non-functional aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 74–82, New York, NY, USA, 2004. ACM Press.

[11] Hector A. Duran-Limon. *A Resource Management Framework for Reflective Multimedia Middleware*. PhD thesis, University of Lancaster", 2001.

[12] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman To Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[13] ISO/IEC. QoS – basic framework. ISO Report ISO/IEC JTC1/SC21 N9309, ISO/IEC, 1995.

[14] ISO/IEC. *ISO/IEC 10746-2 Reference Model for Open Distributed Processing - Part 2: Foundations*, 1996.

[15] ISO/IEC. Working draft for open distributed processing reference model, quality of service. Approved meeting output, editor's draft 6.4, ISO/IEC, June 1998.

[16] Jan Øyvind Aagedal. *Quality of Service Support in Development og Distributed Systems*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2001.

[17] Luis Ferreira Pires João Paulo Almeida, Marten van Sinderen and Maarten Wegdam. Handling qos in mda: A discussion on availability and dynamic reconfiguration. 2003.

[18] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.

[19] Antonio Vallecillo Lidia Fuentes. An introduction to uml profiles, 2004.

[20] Lars Sveen Lundby. Qos monitoring framework, nov 2004.

[21] Joaquin Miller and Jishnu Mukerji. *MDA Guide*. Object Management Group, Inc., June 2003. Version 1.0.1.

[22] Object Management Group. MOF Model to Text Transformation Language - Request for Proposal.

[23] Object Management Group. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, Revised submission August 18 2003, www.omg.org (members only).

[24] Object Management Group. UML Profile for Schedulability, Performance and Time Specification, March 2002.

[25] Object Management Group, Inc. *Unified Modeling Language Specification*, March 2003. Version 1.5 (formal/03-03-01).

[26] Object Management Group, Inc. *XML Metadata Interchange (XMI) Specification*, May 2003. Version 2.0 (formal/03-05-02).

[27] Object Management Group, Inc. *UML 2.0 Superstructure Specification*, October 2004. FTF convenience document (ptc/04-10-02).

[28] Roney Pignaton, Víctor A. Villagrá, Juan I. Asensio, and Julio Berrocal. Developing qos-aware component-based applications using mda principles. In *EDOC*, pages 172–183, 2004.

[29] Richard Staehli, Frank Eliassen. QuA: A QoS-Aware Component Architecture. Technical Report Simula 2002-12, Simula Research Laboratory, 2002.

[30] Simone Röttger and Steffen Zschaler. CQML$^+$: Enhancements to CQML. In J.-M. Bruel, editor, *Proc. 1st Int'l Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France*, pages 43–56. Cépaduès-Éditions, June 2003.

[31] Simone Röttger and Steffen Zschaler. A software development process supporting non-functional properties. In *Proc. IASTED Int'l Conf. on Software Engineering (IASTED SE 2004)*. ACTA Press, 2004.

[32] Ed Seidewitz. What models mean. *IEEE Softw.*, 20(5):26–32, 2003.

[33] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, 2003.