UNIVERSITY OF OSLO
Department of Informatics

Separating the QoS Concern in QuA using Aspect Oriented Programming

Cand Scient thesis

Tore Engvig

23rd October 2005

Contents

1 Introduction **6**

1.1 QoS and Middleware . . . 6
1.2 Separation of Concerns . . . 7
1.3 Problem Statement . . . 7
1.4 Research Method . . . 7
1.5 Results . . . 8
1.6 Structure of this Document . . . 9

2 Background **10**

2.1 Introduction . . . 10
2.2 Quality of Service . . . 11
2.3 Middleware and Components . . . 12
2.4 Reflection and Middleware . . . 14
2.4.1 Behavioral Reflection . . . 16
2.4.2 Structural Reflection . . . 16
2.4.3 Architectural Reflection and Reflective Middleware . . . 16
2.5 Aspect-Oriented Programming . . . 17
2.5.1 Separation of Concerns . . . 17
2.5.2 Aspects . . . 18
2.5.3 Origins of AOP . . . 19
2.6 The QuA Project . . . 22
2.6.1 The QuA Component Model . . . 23
2.6.2 Service Planning . . . 24
2.7 Related Work . . . 24
2.7.1 COMQUAD . . . 25
2.7.2 Quality Objects . . . 26
2.7.3 DynamicTAO . . . 27
2.7.4 OpenCOM and the Lancaster Experience . . . 28
2.7.5 Enterprise JavaBeans . . . 30
2.7.6 IoC frameworks: The Spring Framework . . . 31
2.7.7 Other Middleware Approaches . . . 32
2.7.8 Separation of Concerns using AOP . . . 33

3 Tools and Techniques 34
3.1 The QuA Java Prototype . . . 34
3.1.1 Resources and QoS . . . 36
3.1.2 Service Planning . . . 37
3.1.3 Creating a QoS Aware Component . . . 37
3.1.4 Composing QoS Aware Services . . . 38
3.1.5 Actors and Roles in QuA . . . 39
3.2 AOP Frameworks . . . 40
3.2.1 An AOP Example . . . 40
3.2.2 Comparing AspectJ and AspectWerkz . . . 46

4 Analysis 48
4.1 Overview of the Problem . . . 48
4.1.1 Resources and Monitoring . . . 48
4.1.2 Configuration and Reconfiguration . . . 49
4.1.3 Cross-cutting Concerns . . . 50
4.1.4 Cases for Further Analysis . . . 50
4.2 Simple Case: Computing the Value of Pi . . . 51
4.2.1 Static QoS: QoS Aware Pi Components . . . 52
4.2.2 Monitoring and Resources . . . 53
4.2.3 Dynamic Behaviour: Adding a Cache . . . 57
4.2.4 Aspect Deployment . . . 58
4.2.5 Summary . . . 58
4.3 Complex Case: A Distributed Audio Player . . . 59
4.3.1 Audio Quality and Codecs . . . 60
4.3.2 Real-Time Transport Protocol . . . 62
4.3.3 Monitoring QoS . . . 63
4.3.4 Resource Management . . . 64
4.3.5 Adaptation . . . 66
4.3.6 Summary . . . 70
4.4 Experiments . . . 71
4.4.1 Criteria for Evaluating Results . . . 71

5 Experiments 82
5.1 Simple Compositions: Calculating Pi . . . 82
5.1.1 Separating Static QoS . . . 82
5.1.2 Resource Management . . . 88
5.1.3 Summary . . . 89
5.2 Complex Compositions: A Distributed Streaming Audio Player . . . 89
5.2.1 Implementing an Audio Service . . . 90
5.2.2 Creating a QoS Aware Player Application . . . 94
5.2.3 QoS Monitoring . . . 96
5.2.4 Adaptation . . . 97
5.2.5 Resource Management . . . 105
5.2.6 Testing the Adaptation Mechanism . . . 105
5.2.7 Testing with Remote Capsules . . . 107
5.2.8 Summary . . . 108

6 Evaluation **109**
6.1 Evaluating the Experiments 109
6.1.1 Modularization and Reuse 110
6.2 Aspect Components 114
6.3 Results 115

References **117**

Chapter 1

Introduction

This thesis is a study in the field of quality of service (QoS) aware component-based middleware and separation of concerns. QoS enabled middleware tries to preserve the safe-deployment property of components, but QoS is a *cross-cutting* concern and thus hard to manage by the middleware alone. Aspect oriented programming (AOP) is a new technique for modularizing cross-cutting concerns, and this thesis is a study in how AOP can be applied to QoS enabled middleware in order to separate concerns.

1.1 QoS and Middleware

Component-based middleware has been used with success for many years now. Component-based middleware enables safe-deployment and reuse of components in a well tested and known environment. Component-based middleware provides a runtime environment for components, but as the middleware itself consists of components, it can be tuned and configured to suit the applications' needs.

QoS properties such as timeliness and accuracy is poorly supported in existing component architectures. QoS aware applications usually share resources and they need to adapt to changes in the available resources to enforce their QoS constraints. This usually means that the QoS concern need to be considered in almost every part of the system, and the component implementations need to contain deployment specific knowledge.

Platform managed QoS means that the middleware must be able to reason about how end-to-end QoS depends on the quality of component services and thus adapt the component services to achieve the wanted QoS as resources vary.

Research in this area propose many techniques for making the middleware adapt to changes in its environment (McKinley et al., 2004). Reflection is central in many of those approaches. Reflective middleware is able to reason about itself, and can reason about and alter its own behaviour and structure during runtime. The idea is that the middleware should be able to reconfigure itself and the software that is deployed on it to handle changes in its execution environment.

1.2 Separation of Concerns

As the QoS concern needs to be considered in most parts of the system, it is a *cross-cutting* concern. Cross-cutting concerns are concerns that span multiple objects or components.

Cross-cutting concerns need to be separated and modularized to enable the components to work in different configurations without having to rewrite the code. If the code for handling such a concern is included in a component, it can make the component tied to a specific configuration. This code will typically be scattered all over the component implementation and tangled with other code in the component. Modularizing it will make it more robust for change, and separating it totally from the component implementation will save the component programmers from having to implement it.

Aspect oriented programming is a new method for modularizing cross-cutting concerns. By using AOP, concerns can be modularized in an aspect and later weaved into the code.

1.3 Problem Statement

To enforce QoS properties such as timeliness and accuracy in component services, we need to adapt to changes in the available resources. This adaptation involves cross-cutting concerns such as: Resource management, monitoring and reconfiguration of the services.

The QuA project at Simula Research Laboratory aims at providing a component-model and platform for QoS sensitive components.

Aspect oriented programming is supposed to help separating cross-cutting concerns. This thesis investigates how aspect oriented programming can be used to help separating the QoS concern in order to create an architecture for platform managed QoS.

The goal of this thesis is to see how those concerns can be separated, and whether the use of AOP contributes to modularizing those concerns.

The thesis is limited to investigating separation of concerns. The goal is neither to create a state-of-the art monitoring framework, nor is it to create a state-of-the art resource management or dynamic reconfiguration framework. Thus, details about distributed resource management, monitoring protocols and service planning are beyond the scope for this thesis.

1.4 Research Method

The hypothesis in this thesis is that the QoS concern can be separated with AOP in the QuA platform. The term “QoS” is a broad term and covers many QoS dimensions. Different applications need QoS in different ways. Thus, validating such a hypothesis for the general case is a formidable, and maybe even impossible, task. Instead, the hypothesis can be strengthened by investigating cases that cover relevant parts of the hypothesis.

Thus, an experimental approach with case analysis is chosen as method. By analysing concrete cases, the scope is narrowed and it is easier to achieve concrete results that can

be validated.

The cases should contribute to answering:

- How to separate the QoS concern in a static composition, i.e., how to separate the concern of choosing the "best" components when forming a service.

- How to achieve dynamic adaptation, i.e., how to change or reconfigure a running service. This includes how to dynamically adapt a service composed of more than one component.

- How to separate the interaction with resource managers and monitors.

QoS in the QuA project is limited to accuracy and timeliness. Thus, cases concerning accuracy and timeliness are selected.

Two prototypes are implemented. The first prototype separates static QoS from component implementations. The second prototype looks at monitoring and adaptation of a stream based audio service.

The resulting prototypes are then evaluated in terms of modularity and reusability in order to determine the degree of separation of concerns achieved. At the current time there exists no good metrics for measuring the degree of separation of concerns achieved by using AOP – or for measuring anything useful at all using AOP. The problem of how to evaluate the prototypes are discussed in section 4.4.1 based on the current metrics used in software engineering and other experiments involving AOP.

1.5 Results

Results of the analysis and the experiments conducted in this thesis show that:

- Architectural reflection can be enabled with the use of AOP. The experiments implement a simple meta-model for components and services which can be used for architectural reflection.

- Adaptation of components and complete services is implemented with aspects. Both compositional and parameter adaptation are implemented. Architectural reflection is used to adapte complete services, and for transferring bindings between components.

- Monitoring of some QoS dimensions are separated using AOP.

- Resource management is separated with the use of aspects.

- Static QoS is separated from component implementations using AOP.

The use of AOP to separate these concerns provided a modular implementation of the concerns. Most of the aspects used for separating the concerns are orthogonal to the components they act upon, and should be reusable for other services and components.

1.6 Structure of this Document

This thesis is structured as follows:

- Chapter 2 gives a background on reflective middleware, separation of concerns and related work.
- Chapter 3 describes details of the QuA Java prototype used in the experiments and a description of the most relevant AOP tools to use.
- Chapter 4 provides an in-depth analysis of the problem and describes how to conduct and evaluate experiments.
- Chapter 5 describes how the experiments are implemented.
- Chapter 6 evaluates the experiments, concludes and suggests potential further work.

Chapter 2

Background

2.1 Introduction

Many properties of a system can be termed Quality of Service (QoS). E.g., security, availability and a host of other -ilities. These different -ilities are usually called QoS dimensions. This thesis focuses on *timeliness* and *accuracy*.

Key to QoS is that resources are limited. In order to fulfill the requirements in one QoS dimension, we might have to sacrifice another QoS dimension. I.e., we have to prioritize. An example is a streaming audio service such as internet radio, an audio conference or IP-telephony. Continuous sound and low latency is important in such services. If the network bandwith is limited, we might have to reduce the sample rate or with other means reduce the amount of data sent over the wire. This will reduce the quality of the sound, but satisfy the timeliness constraint.

There is a lot of work involved in creating such a service. You need to create audio encoders, audio decoders and code for handling network transport of your audio data. If you create all this code once, you might want to use it in more than one system – and maybe in different configurations of the system. You probably want to *reuse* the code. Reuse does not only save you a lot of work, it also makes maintenance easier as you only have to fix bugs in the original code.

Making software components from your audio streaming code is a good approach to achieve this. Components are reusable units usually packaged in a binary format. Components specify which interfaces they provide, which interfaces they require, and they also make explicit any other possible context dependencies. I.e., if you create components from your audio encoder and decoder, you can use those components to compose your audio service.

If you have new versions of your components containing bug-fixes, or maybe even a completely new audio encoding algorithm, you only have to replace the original components. As the new component has to satisfy the old component's specification (i.e., it has to provide the same interfaces), the new component will fit in your service without a need to change any of the other components or code.

You have to deploy your components to something. I.e., they need a run-time environment that provides a component model – an understanding of what a component is and how to make it available to your program. This is called a component platform.

Component platforms are often middleware. Middleware acts as glue between applications and has traditionally been concerned with distribution, i.e., the network code in your audio streaming application.

If you program your audio streaming service using low-level network calls, you will write a lot of code that someone else already has written. If you in addition would like failover support or a lookup service for locating audio sources in your application, there is a considerable amount of code to write. Traditionally this has been handled by middleware.

By using component-based middleware, all that network code could be components provided by the middleware. Component-based middleware also allows you to switch the components in the middleware itself, making it possible to fine-tune the middleware to suit different configurations.

Your audio service does probably not run on dedicated computers using dedicated leased-lines. This means that resources such as network bandwidth and CPU power might change while your service runs. To fulfill the QoS constraints of your audio service, it has to adapt to changes in available resources. This may imply changing the sample rate or switching to a less CPU intensive audio encoding algorithm.

Adapting to changes could be handled by the code in your components. This will require that they run on a middleware platform that does not hide too many details from the components, as knowledge of those details might be necessary to determine when and how to adapt to changes. Another serious drawback of letting your components handle the adaptation is that they might need knowledge about the whole component composition. This will make your components less reusable and hence less suitable for other configurations.

This adaptation could be handled by the middleware. This kind of middleware is called *adaptive middleware*. One approach for making middleware adaptive is to use *reflection*, and create *reflective middleware*. Reflective middleware is able to reason about itself and the services that are deployed on it. Ideally, this makes the middleware capable of detecting that it has to adapt, and then reconfigure or recompose the audio service to fulfill its QoS constraints.

Adaptation requires knowledge of both *when* and *how* to adapt. Code for handling this must be included in most of the middleware code. There is also the possiblity that your components need dedicated code to handle this. Aspect Oriented Programming (AOP) is a technique for separating this code into modules, and at a later stage weave it back in again.

The rest of this chapter elaborates on the topics presented here, and provides an overview of the most relevant research in the field.

2.2 Quality of Service

There are many definitions and interpretations of what Quality of Service (QoS) is. Aagedal (2001) gives an overview of some of the different interpretations of QoS. In this work, the understanding of QoS is based on the definitions from ISO (ISO CD15935; ISO 13236) and OMG's UML profile for QoS (ptc/2004-09-01).

QoS is about the *performance* of the system. QoS is also about quantification and

prioritization of QoS *characteristics* in order to meet the total quality requirements for a service.

Some terms and definitions need to be clarified:

QoS characteristic A quantifiable aspect of QoS, which is defined independently of the means by which it is represented or controlled. Latency, throughput, capacity and scalability are examples of QoS characteristics.

QoS dimension Dimensions for the quantification of QoS characteristics. QoS characteristics can be quantified in different ways. A QoS dimension is one such way. E.g., the latency of a system function can be quantified as the end-to-end delay of the function, the mean time of all executions or the variance in time delay. The term QoS dimension is often used instead of QoS characteristic.

QoS requirement QoS information that expresses a part of, or all of, a requirement to manage one or more QoS characteristics, e.g., a maximum value, a target, or a threshold. When conveyed between entities, a QoS requirement is expressed in terms of QoS parameters.

QoS monitoring The use of QoS measures to estimate the values of a set of QoS characteristics actually achieved for some system's activity.

QoS characteristics can be grouped into *QoS categories*. Performance and dependability are examples of QoS categories that include many characteristics.

This thesis is part of the QuA project, and the QuA project focus on QoS in terms of *timeliness* and *accuracy*. Timeliness is related to how the system is able to meet the constraints on request/response delays. Some characteristics related to timeliness are:

Latency/delay The time it takes to send a message from A to B. E.g., a geostationary satellite orbits at about 36.000 km above earth. When using satellite communication a packet would have to travel 72.000 km, thus introducing a delay of about 0,24 seconds.

Jitter Variability in transfer delay.

A characteristic related to accuracy is *error probability*.

2.3 Middleware and Components

Middleware is software that connects applications. Middleware is neither dedicated to the operation of a specific system, nor the functionality of a specific application – middleware serves as glue between applications.

Schmidt (2002) decomposes middleware into the four layers in table 2.1:

- Host-infrastructure middleware resides on top of the operating system and provides a high-level API that abstracts away the heterogenity of hardware devices, operating systems, and network protocols.

Table 2.1 Four layers of middleware.

Applications
Domain-specific middleware services
Common middleware services
Distribution middleware services
Host infrastructure middleware services
Operating system and protocols
Hardware devices

- Distribution middleware defines higher-level distributed programming models whose reusable APIs and mechanisms automate and extend the native operating system's network programming capabilities encapsulated by host-infrastructure middleware. CORBA, DCOM, and Java RMI all fit in this layer.

- Common middleware services augment distribution middleware by defining higher-level domain-independent APIs that allow application developers to focus on programming application logic instead of having to write the "plumbing" code needed to develop distributed applications. This include fault tolerance, security, persistence, and transactions.

  Message-oriented middleware (MOM) – such as IBM MQ Series, Microsoft MQ services and message brokers – falls into this category. So does most of the middleware discussed in the related works section later in this chapter.

- Domain-specific middleware services are tailored to match a particular class of applications, such as avionics mission computing, online financial trading and distributed process control. Unlike the previous three middleware layers, which all provide broadly reusable "horizontal" mechanisms and services, domain-specific middleware target vertical markets.

  Domain-specific middleware are the least mature of the middleware layers, partly due to the historical lack of distribution middleware and middleware service standards needed for a stable base on which to create domain-specific middleware.

In short, middleware is a broad term that includes almost anything but the application itself and the operating system. Middleware has gained popularity because it provides functionality that many applications need, but is hard or expensive to create. E.g., many applications need security, distribution and transactions, but implementing a scalable and fault tolerant system for distributed transactions is expensive.

By using middleware for such operations, reuse of the middleware software is achieved, which makes the software cheaper and less error-prone. The application can focus on handling the application's concerns while the middleware takes care of concerns such as transactions and remoting.

Component-based middleware provides a run-time environment for components in the middle tier. Components are *"units of composition with contractually specified interfaces and explicit context dependencies only"* (Szyperski et al., 2002). Examples of compo-

nents are Enterprise Java Beans, Microsoft COM Components or CORBA CCM components.

As stated in the definition, components are units of composition. That means that components can be composed to form an application, another (composite) component or a service. Context dependencies are specified by stating the required interfaces and the acceptable execution platform(s). With explicit context dependencies and contractually specified interfaces, it is easy to create several implementations of the same component. Different implementations might suit different needs, but they can still be deployed the same way, and in the same environment, as the other implementations of the same component.

The characteristic properties of a component are (ibid.):

- It is a unit of independent deployment.
- It is a unit of third-party composition.
- It has no (externally) visible state.

All this makes components *reusable* assets. This makes it possible to buy pre-fabricated components and combine them with your own components in a multitude of configurations – or *compositions*. The same component can be reused in many systems.

Component instances can be plugged into a *component framework*. Component frameworks support components confirming to certain standards and govern the interaction of components plugged into the framework. Component frameworks establish the environmental conditions for the component instances, and regulates their interaction.

Component frameworks can be components themselves, and they can be organized hierarchically, allowing component frameworks to be plugged into higher level component frameworks. Szyperski et al. (2002) uses EJB containers and COM+ contexts as examples of tier one components frameworks, work flow and integration servers as tier two, and federated flow control models at the interorganizational level as examples of tier three component frameworks.

The middleware that provides the run-time environment for components are often called the *component platform*.

Component-based middleware does not only provide a run-time environment for components. The middleware itself consists of components that can be changed and configured. This means that the middleware can be tuned to suit the applications' needs. OpenCOM, which is discussed in the related works sections, is a good example of such middleware.

2.4 Reflection and Middleware

Reflection is the ability for a system to reason about itself. This is achieved by creating a meta-representation of the system. The system and the meta-representation are *causally connected* such that changes in the meta-representation are reflected in the system and vice versa.

A major contribution in the research on reflection was given in Brian Smith's PhD thesis (Smith, 1982, 1984). Smith recognized that the base- and meta-level had to be causally connected[1]. His research resulted in 3-Lisp – a fully reflective Lisp implementation.

3-Lisp introduced the concept of *reflective towers*, which is a fundamental part of most reflective programming languages. Reflective towers are a solution to the problem that occurs when a program modifies the data structures used to run the program itself. Reflective towers stacks interpreters such that there is a distinction between the interpreter running the program, and the interpreter running the reflective code. The reflective code might itself contain more reflective code, which in turn runs in another interpreter – thus creating a stack of interpreters.

Reflection in object oriented programming languages was studied by Pattie Maes (Maes, 1987) when she implemented reflection in the KRS language as a part of her PhD thesis. The resulting language was called 3-KRS and contributed with clarifying and defining what reflection and reflective architecture is.

In 3-KRS there is a one-to-one relationship between objects in the language and meta-objects. Further, 3-KRS provides a complete and consistent self-representation where every entity in the system is an object, and thus has a meta-object. This self-representation can be modified at run-time.

Another important work on reflective object systems is the CLOS MOP (Kiczales et al., 1991). This is a metaobject protocol for CLOS (the Common Lisp Object System). Kiczales et al. define a MOP as:

> *Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the language's behaviour and implementation, as well as the ability to write programs within the language.*

Some terminology regarding reflection needs to be clarified:

Reification is the act of going from the base-representation to the meta-representation (Friedman and Wand, 1984).

Using Smith's reflective towers, reification is the same as moving up one level in the stack of interpreters.

Reflection although the whole concept of having a causally connected base- and meta-level is referred to as reflection (Maes, 1987), the term reflection is also used as meaning the inverse of reification. I.e., going from the meta-representation to the base-representation.

Using Smith's reflection towers, reflection is the same as moving down one level. I.e., the current base-level becomes the meta-level.

Introspection Reflection can be split into introspection and intercession (Demers and Malenfant, 1995). Introspection is the *observation* of state and structure.

Intercession is the *alteration* of behavior and structure. I.e., altering the meta-representation such that the modifications is reflected in the base-level.

[1] Smith did not only recognize the importance of a causal connection, he actually made quite a point of it: "*Without such a connection, the account would be useless – as disconnected as the words of a hapless drunk who carries on about the evils of inebriation, without realising that his story applies to himself*" (Smith, 1984).

2.4.1 Behavioral Reflection

It is customary to distinguish between *structural* and *behavioral* reflection. Behavioral reflection is about reflection on the behavior of the system. In a tutorial on behavioral reflection given at the Reflection'96 conference (Malenfant et al., 1996), behavioral reflection is defined as:

> *The ability of the language to provide a complete reification of its own semantics and implementation (processor) as well as a complete reification of the data and implementation of the run-time system.*

We can distinguish between *dynamic* and *static* behavioral reflection. Static behavioral reflection occurs at compile-time, while dynamic behavioral reflection can occur at run-time.

We can also distinguish between *discrete* and *continuous* behavioral reflection. Discrete behavioral reflection is when the reflective computation is initiated at a discrete point by calling a reflective procedure. Continuous behavioral reflection is when the reflective computations have a continuous effect on the base level computation.

Implementing full support for behavioral reflection in a programming language is far from trivial. E.g., Java has no support for behavioral reflection, the only means to dynamically alter behavior in standard Java is to use *dynamic proxies*.

There has been several projects working with adding behavioral reflection to Java. Kava (Welch and Stroud, 2001) is a system that lets you intercept and possibly alter all object access (i.e., method call, field read/write, etc). This is done by modifying the Java byte code at class load-time in order to insert hooks that call the interceptor object. MetaXa (Golm and Kleinöder, 1998) use another approach, and modifies the Java virtual machine in order to add meta-objects.

A powerful tool for behavioral reflection in Java is Javassist (Chiba and Nishizawa, 2003). Javassist lets you modify all aspects of a Java class (add/remove/alter methods, fields, inheritance hierarchies, etc). Adding code to a method using Javassist is a simple task that does not require knowledge of Java bytecode. Javassist allows you to add strings containing Java code that will be compiled at run-time.

Javassist has been used in AOP frameworks such as AspectWerkz and JBoss AOP.

2.4.2 Structural Reflection

Structural reflection is about reflection on the structure of a system (Chiba, 2000). In a programming language this means introspection and intercession of the structure of a program, i.e., reflecting on class definitions, data types, etc.

In the Java language this is partly supported by the reflection API, but it only supports introspection. Kava, mentioned in the previous section, has some support for structural reflection in Java. Javassist, also mentioned in the previous section, has full support for structural reflection.

2.4.3 Architectural Reflection and Reflective Middleware

The use of reflection is not limited to programming languages. Reflection is about systems that reason about themselves. Thus, the terms structural and behavioral reflection

are useful for other aspects of a system.

Architectural reflection (Cazzola et al., 2001) is the computation performed by a software system about its own software architecture. Applying reflection on the software architecture level may help creating more flexible and dynamic applications.

This has been exploited to a great extent in reflective middleware (Kon et al., 2002). Middleware has traditionally followed the black box principle (Kiczales, 1996), but this can hide details that often is needed by the programs running on the middleware[2].

Reflective middleware opens up the implementation and lets the applications access and modify meta-models in the middleware in a controlled manner without breaking encapsulation.

Middleware may have many meta-models (Costa et al., 2000). Examples are the meta-models of OpenCOM (Coulson et al., 2002):

- *MetaInterception* provides a meta-model for the execution model of components.
- *MetaArchitecture* provides a meta-model of the composition of components.
- *MetaInterface* provides a meta-model of the component types.

Reflective middleware is described in more detail in the related work section (section 2.7 on page 24).

2.5 Aspect-Oriented Programming

AOP is a new method for modularizing cross-cutting concerns. When you design a system, you decompose your system into *something*. Depending on the level you are designing, you can decompose into components, sub-systems, modules, classes, objects, etc.

The goal of this decomposition is usually to *separate concerns*. However, not all concerns can be decomposed to the structures mentioned above – some concerns are *cross-cutting*. They span multiple other concerns. AOP is a means for modularizing those concerns and weave them into the system at a later point of time using some kind of *aspect-weaver*.

2.5.1 Separation of Concerns

A concern is *"those parts of software that are relevant to a particular concept, goal, or purpose"* (Ossher and Tarr, 2001). The IEEE recommended practice for architectural descriptions (IEEE 1471) defines a concern as:

> *Those interests which pertains to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders.*

[2]CORBA's use of location transparancy hides the fact that e.g., TCP/IP is used for communication

Examples of concerns are security, reliability, safety, QoS, etc. Concerns are the primary criteria for decomposing software into smaller, more managable and comprehensible parts.

The idea of separating concerns and hiding their complexity by abstractions is not new. Dijkstra (1968) argued for separate layers of functionality in operating systems in the late 60's. Parnas (Parnas et al., 1971; Parnas, 1972) described modules as "responsibility assignments" in the 70's, and argued for information hiding as a criteria for decomposing the system into modules.

Saltzer et al. (1984) argued that functions placed at low levels of a system may be redundant or of little value when compared to the cost of providing them at that low level. Those functions should instead be implemented in the applications. Examples include bit error recovery, security using encryption, duplicate message suppression, recovery from system crashes and delivery acknowledgment.

This are all examples of how to separate concerns by decomposing the system into smaller units that handles a specific concern. Separation of concerns has been an important part of software development for a long time. E.g., most of the patterns in the GoF book (Gamma et al., 1995) tries to separate concerns, role modeling (Reenskaug et al., 1996) strives to achieve separation of concerns, and pattern constructs such as adaptive object models (Yoder and Johnson, 2002) also tries to spearate concerns.

However, none of the methods mentioned above handles *cross-cutting* concerns very well. Cross-cutting concerns spans through other concerns resulting in code scattered around multiple classes and tangled with other code in those classes.

Typical examples of such concerns are: logging, transaction management and remoting. Even if you have separated your business objects and transaction management objects, the business objects needs to call the transaction management objects in order to begin and end transactions.

2.5.2 Aspects

Aspects are the construct used for capturing the cross-cutting concerns. A clear definition of what an aspect is, does not exist. There has been many attempts at separation of cross-cutting concerns, and most of them are some sort of "aspect-oriented programming" (AOP) – without having a notion of an "aspect".

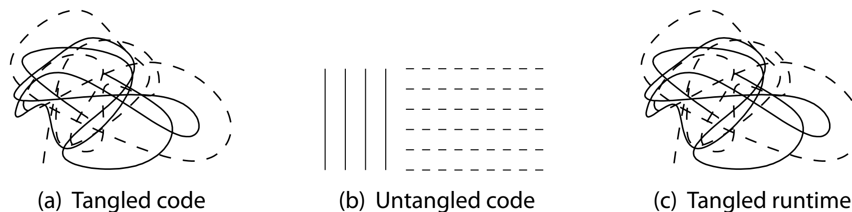
Today, AOP is more or less synonymous with the AspectJ (Kiczales et al., 2001) model for aspects. In AspectJ, an aspect is a first-class construct that contains the separated concern and the description of how to weave it into the code.

Common for all the aspect frameworks are the elements they use to separate cross-cutting concerns. Those elements are (Elrad et al., 2001):

1. A *join point model* describing the hooks where enhancements may be added.
2. A means of *identifying join points*.
3. A means of *specifying behavior at join points*.
4. *Encapsulated units* combining join point specifications and behavior enhancements.

5. A method of attachment of units to a program.

Figure 2.1 Cross-cutting concerns are separated at the code level and weaved in at byte-code level.



(a) Tangled code (b) Untangled code (c) Tangled runtime

Aspects modularizes cross-cutting concerns at the code level and are weaved into the system at runtime[3]. Figure 2.1 illustrates this. The solid lines represent code that handles a concern, and the dotted lines represent code that handles cross-cutting concerns. Figure 2.1(a) shows how cross-cutting concerns are handled with tangled code scattered around the code-base. Figure 2.1(b) shows all concerns separated at the code-level and finally, figure 2.1(c) shows the running system where the aspect weaver has weaved in the cross-cutting concerns.

Some code samples to further illustrate this is given in section 3.2.

2.5.3 Origins of AOP

Most of the research that led to aspect oriented programming was done during the early 90s. Some of this research is presented here to give an historical background to the subject. While all of these approaches may be called "aspect oriented" in the sense that they all support separation of cross-cutting concerns in some way, only the AspectJ approach is associated with the term AOP today.

2.5.3.1 Composition Filters

Composition filters (Aksit et al., 1992) was the first framework that could be considered aspect oriented. Composition Filters was developed at the University of Twente as a part of the Sina language.

Composition filters are specified as functions that manipulate messages received and sent by objects, i.e., they work as interceptors for messages between objects. These filters are specified independent of a specific programming language. Cross-cutting concerns are captured in the filters.

Filters can be composed, and by using superimposition (Bergmans and Aksit, 2001), filters can be imposed on one or more objects.

There exists implementations of composition filters for Sina, Java and .Net[4].

[3] Aspects can be weaved at compile-time, load-time or run-time. Most aspect weavers work at compile-time or load-time.

[4] See the composition filters Wiki for more information about the implementations: http://janus.cs.utwente.nl:8000/twiki/bin/view/Composer/WebHome

2.5.3.2 Adaptive Programming and the Law of Demeter

Adaptive programming (Lieberherr, 1996) is a technique developed by the Demeter group at the Northeastern University.

Adaptive programming is highly influenced by the Law of Demeter (LoD). LoD is a style rule for designing object-oriented systems stating that you should *"Only talk to your immediate friends"*. Or more general:

> *Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.*

Here, a "unit" is a method, and "closely related" means methods in the same class as the unit, or methods of argument classes or part classes. This is a specialization of the low coupling principle.

Operations in object-oriented programs often involves a set of cooperating classes. One can either localize this operation in one class, or split the operation over the set of associated classes.

The problem with the former option is that too much information about the structure of the classes (is-a and has-a relationships) needs to be tangled into each such method. This makes it difficult to adapt to changes in the class structure, and it violates the Law of Demeter. The latter option scatters the operation across multiple classes, making evolution difficult.

Adaptive programming uses traversal strategies and the visitor pattern to help enforcing LoD. A traversal strategy defines how to traverse a graph of has-a relationships. E.g., a very simple has-a relationship is a Company that has Employees. The traversal strategy defines how to traverse this graph, and behaviour may be added at the nodes by using a visitor pattern. This style of programming is "structure-shy", i.e., it separates structure from behaviour.

Adaptive programs are transformed from traversal strategy descriptions into regular object oriented programs using the Demeter tools. E.g., Demeter/J or DJ for Java and Demeter/C++ for C++.

LoD has later been extended to LoDC: the Law of Demeter for Concerns (Lieberherr, 2004). LoDC restricts what your friends are, and states that:

> *Talk only to your friends who contribute to your concerns or that share your concerns.*

Adaptive programming has been taken one step further with the JAsCo AOP framework (Vanderperren et al., 2005). JAsCo is an AOP framework created at the Vrije University tailored for component-based development. Adaptive programming are supported in JAsCo through the use of adaptive visitors and traversal connectors.

2.5.3.3 Subject Oriented Programming and Hyper/J

Subject Oriented Programming (Harrison and Ossher, 1993) is a method for decomposing concerns developed at IBM Watson Research Center. In subject oriented programming (SOP) a concern is captured in a *subject*. A subject is composed of classes and other subjects. A subject has a distinct *view* of a class, and that class may be shared with other subjects that might have other views of it.

An activated subject instantiates the classes, and those instances can also be shared with other subjects. I.e., different subjects can see the same object as having different properties and behavior.

The ideas for SOP has been further extended in Hyper/J (Ossher and Tarr, 2000; Tarr et al., 1999). Hyper/J is a tool for weaving *hyperslices* into *hypermodules* to form a *hyperspace*.

A hyperslice captures a concern, and is a set of conventional modules. A hyperslice is based on the old method of program slicing (Weiser, 1981) where a slice is a subset of a program's behaviour reduced to a minimal form that still produces that behaviour. Just as with subjects in SOP, hyperslices might overlap.

A hypermodule contains a set of hyperslices, which is composed using composition rules. Hyper/J is a tool for Java that weaves the hyperslices into hypermodules using the composition rules.

Some of the ideas from SOP and hyperspaces are used to create the Concern Manipulation Environment (CME). CME (Harrison et al., 2004) is a plugin to the Eclipse IDE[5] that helps you identify, extract and compose concerns.

2.5.3.4 AOP and AspectJ

The term "Aspect Oriented Programming" was coined by Gregor Kiczales and his team at Parc Xerox in the mid-90s. AOP and AspectJ (Kiczales et al., 1997, 2001) is inspired by the work with meta-object protocols and the CLOS MOP (Kiczales et al., 1991) and the ideas of open implementations (Kiczales, 1996; Maeda et al., 1997).

In AOP, aspects capture the cross-cutting concerns. AOP follows the elements used for separation of cross-cutting concerns listed in section 2.5.2 quite closely.

An aspect is a separate unit that contains the join point identifications and the behaviour to attach to them. Join point specifications contains regular expressions that matches classes, methods or attributes. Join points can also be combined using boolean logic. Behaviour at those join points can be altered, and the code that is attached is called *advice*[6].

AspectJ also allows to structurally change classes by using intertype declarations or changing the inheritance hierachy. This is close to structural and behavioral reflection. Discussing if it *is* reflection is not really interesting, as most AspectJ-like tools *uses* structural and behavioral reflection to implement AOP. E.g., AspectWerkz uses Javassist and byte-code modification tools to implement AOP.

Aspects are later weaved into the code using an aspect weaver such as AspectJ. A more detailed description of AspectJ with code samples is given in section 3.2. The semantics for describing aspects defined in AspectJ has become quite popular, and has

[5]See http://www.eclipse.org/cme/ for more updated information about CME.

[6]A similar method of adding behavior at arbitrary points in the code was discussed a long time ago in quite a different context. In 1984, the April edition of Communications of ACM contained an April fools article (Clark, 1984) that describes the *ComeFrom* statement as a linguistic approach to solving the goto controversy of the 70s. *ComeFrom* is the inverse of *GoTo*. In AOP terms the code after the *ComeFrom* statement is an advice, and the label or line number you come from is the join point. This novel feature has been implemented in the esoteric programming language Intercal, and according to rumors on the Portland Pattern Repository's Wiki (http://c2.com/cgi/wiki?ComeFrom), this feature has been used as a joke among assembler programmers in the late 60s and early 70s.

spawned many related projects[7].

2.5.3.5 Other Approaches

There exists many other approaches to achieve separation of concerns. Meta programming and reflective techniques has already been described, and is also covered in the section that describes related works.

Interceptors (Narasimhan et al., 1999) is a pattern that allows services to be added transparently to a framework and triggered automatically when certain events occur. Interceptors are similar to composition filters, only less sophisticated, as they cannot be composed or weaved in any way. The concept of intercepting the program flow is fundamental to many of the aspect oriented techniques, and variations of this is used in many different systems. E.g., mode hooks in the emacs text editor, servlet filters in the J2EE servlet specification and they are also part of the CORBA specification (orbos-02-06-57).

MDA (Frankel, 2003) expresses concerns in UML models. The UML model describes all the concerns in the system, e.g., by using OCL constraints, stereotypes or action languages. This model can then be run in a UML virtual machine or through a code generator that generates running code for the model. QoS concerns and MDA are still in an early research phase.

Generative programming (Czarnecki and Eisenecke, 2000) is a method that uses code generation to generate code from descriptions such as domain specific languages.

There are no distinct boundaries between these methods. E.g., generative programming can generate aspect oriented source code. It can also interpret the descriptions during run-time instead of generating the code, and if the description it uses for code-generation is a UML model, it is getting close to MDA. COMQUAD and QuO is good examples of how all this mix together. COMQUAD and QuO is described in the related works section (section 2.7 on page 24).

It is also worth mentioning the resemblance between AOP and active databases (Dittrich et al., 1995). Much of the research on AOP and active databases happened at the same time (the mid 90s), and when comparing the event-condition-action model of active databases with the join point model of AOP, one finds many similarities[8]. These similarities are explored by Cilia et al. (2003) who suggests a convergence between AOP and active databases to form a new kind of distributed service-oriented systems.

2.6 The QuA Project

The QuA project at Simula Research Laboratory is investigating how a component architecture can preserve the safe deployment property for QoS sensitive applications (Staehli et al., 2004). QuA focuses on the timeliness and accuracy QoS properties.

QuA defines a component platform that is able to *plan and execute QoS sensitive services*. The QuA architecture defines only the minimal services needed to discover

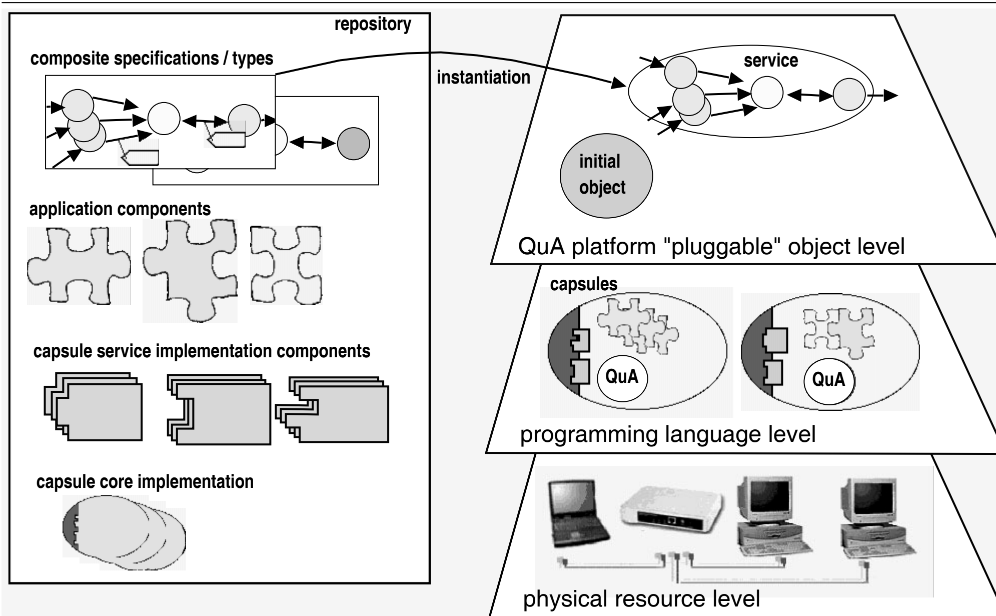
[7] See http://aosd.net/technology/ for an exhaustive list of AOP frameworks.

[8] According to Parnas (1996) this is a good thing, as *"The wheel is reinvented so often because it is a very good idea; I've learned to worry more about the soundness of ideas that were invented only once"*.

and execute such plans. All other services, including remote protocols and specialized service planners, may be provided as plugin components.

QuA is component based middleware. It defines a small core on which components can be deployed on. The QuA middleware itself consists of components that can be changed and replaced to suit a specific configuration.

Figure 2.2 A conceptual view of the QuA architecture[a].



a) Figure 2.2 is adopted from (Staehli and Eliassen, 2002).

QuA also aims to be *adaptive* middleware, i.e., the QuA platform will be able to detect changes in the system's environment, and adapt services running on the platform to those changes.

Figure 2.2 shows an early conceptual view of the QuA architecture. Components are instantiated from a repository to become part of a service running in the QuA platform. The QuA platform supports execution of, and communication between, QuA objects, and may span multiple distributed capsules. A capsule is similar to what is called a container in other middleware architectures, and provides an object representing the QuA platform to support platform managed service instantiation and binding.

2.6.1 The QuA Component Model

QuA defines a component model for the platform, and the most important terms in this model are:

Component platform is a virtual machine for manufacturing, composing, and executing objects from software components.

Software component is the runtime objects manufactured from blueprints. It is interpreted by a component platform to manufacture objects.

Blueprint is a persistent and immutable value encoding how to implement a component.

Binding is a mechanism enabling object communication.

Blueprints are sometimes called "component templates" in other component models, and software components – or only components – are often called component instances. In this document, the terms defined by QuA will be used. I.e., a component is an instantiated blueprint.

QuA blueprints are discovered and instantiated through *capsules*. Capsules have access to repositories containing the blueprints. Blueprints is looked up based on a *QuA-Name*, which is the logical repository name described with a URI like syntax.

2.6.2 Service Planning

QuA defines the QoS semantics (Staehli and Eliassen, 2004) needed to provide QoS management support in the platform. Important concepts from the QoS semantics are the notion of *utility functions* and *error models*. A utility function is a function describing how "good" a set of QoS parameters are for a user.

An ideal output trace is generated when a service executes completely and correctly on an infinite fast platform. An error is the devitation between the actual output trace and the ideal output trace. An error model is a vector of error functions capturing this difference. An *error predictor* function is a function that can predict this error for a component.

Central in the QuA architecture is the concept of *plans* and *planners*. A service planner plans a service based on a service specification and a quality specification. The service specification describes the service in terms of the component types (actually QuA-Names) involved. The quality specification contains an *error model* and a *utility function* that describes how "good" a set of QoS parameters is. The service planner then tries to create a service that maximize the utility.

For more details on the QuA platform, see section 3.1, which describes the QuA Java prototype used in this thesis.

2.7 Related Work

There has been a lot of research on the topic of middleware and separation of concerns. This section presents some of this work with an emphasis on component-based middleware and QoS. Most of this is research work from academia, but an example of state-of-the art commercial middleware (Enterprise Java beans) and open-source middleware (the Spring framework) is also presented.

There is also a section on related work in the field of AOP showing that AOP actually does help separating cross-cutting concerns in real world case-studies.

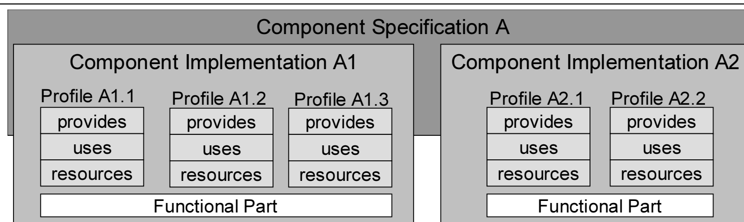
2.7.1 COMQUAD

The COMQUAD project (Göbel et al., 2004b,a) works with the issues involved with supporting non-functional properties in component-based systems.

COMQUAD has a component model and a container that separates the non-functional concerns from the component implementations. This is achieved by making the container responsible for instantiation of component specifications based on a client's non-functional requirements.

The COMQUAD component model is based on Szyperski's definition of a component (Szyperski et al., 2002) with a few additions: The concept of *Home Interfaces* are borrowed from Enterprise JavaBeans (EJB), and the concepts of *facets* and *receptacles* are borrowed from the CORBA Component Model (CCM). Another addition is *streaming interfaces*.

Figure 2.3 COMQUAD Component specifications, implementations and NFP Profiles[a].



a) Figure 2.3 is adopted from (Göbel et al., 2004b).

COMQUAD distinguishes between *active* and *passive* components. Active components are components that require a thread of their own, such as an audio streaming service. Passive components are plain request-response components running in the same thread as the caller.

A component consists of a specification, a set of non-functional profiles and a set of component implementations (see figure 2.3). The non-functional profiles are specified using CQML$^+$ (Röttger and Zschaler, 2003). CQML$^+$ is an extension to CQML (Aagedal, 2001) that includes support for modelling resource demands.

The component specification in figure 2.3 contains the functional specification. In QuA, this resembles the QuA type. Component implementation A1 and A2 is the same as QuA blueprints, and the different profiles are different QoS configurations.

A component based application in COMQUAD is represented by an assembly of component specifications. Assemblies are specified with assembly descriptors. An assembly descriptor contains many other descriptors.

Most notably it contains a *component network template*. A component network template is a description of how the different components are connected. E.g., the composition of components that together form a video player, is described in a component network template.

The COMQUAD container is reponsible for instantiating the component network template, thus forming a *component network*. This instantiation is based on QoS requirements and available resources.

The container is split in a real-time part, and a non real-time part. The real-time part

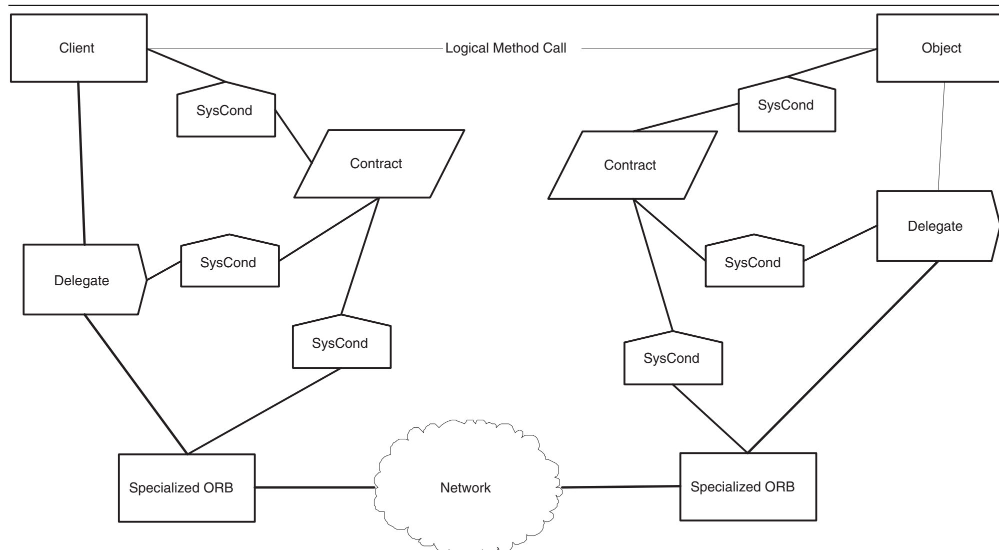
is written in C++ and use the DROPS real-time operating system for resource reservation. The non real-time part is written in Java and is based on the JBoss application server. This part handles all operations that does not have real-time requirements, such as deployment and negotiation of component contracts.

The current version of COMQUAD is not able to adapt the component network at runtime, but adherence to component contracts are enforced by a contract manager that intercepts all calls between components.

2.7.2 Quality Objects

QuO (Loyall et al., 1998; Duzan et al., 2004) is a framework for including QoS in distributed object applications. QuO supports the specification of QoS contracts between client and service providers, runtime monitoring of contracts and adaptation to changing system conditions.

Figure 2.4 Overview of a remote method invocation in a QuO application[b].



[b] Figure 2.4 is adopted from (Loyall et al., 1998).

Contracts are specified in a Quality Description Language (QDL). QDL consists of:

Contract Description Language (CDL) CDL describes the QoS contract between a client and an object. This includes QoS that the client desires from the object, the QoS that the object expects to provide, regions of possible levels of QoS, system conditions that need to be monitored, and behavior to invoke when client desires, object expectations or actual QoS conditions change.

Structure Description Language (SDL) SDL describes the internal structure of remote objects' implementations, such as implementation alternatives and the adaptive behavior of object delegates.

Resource Description Language (RDL) RDL describes the resources available in the system and their status.

Aspect Specification Language (ASL) ASL describes adaptive behaviour of objects and delegates.

Figure 2.4 shows a remote method invocation in a QuO application. The system condition objects (SysCond) interface between the contract and resources, mechanisms, objects and ORBs in the system. These are used to measure and control QoS.

A CDL contract consists of *regions* of QoS. When QoS parameters change, a *transition* to another region might occur. Those transitions, and which client callback methods that should be called, are specified in the CDL contract.

Adaptive behaviour of objects and delegates are specified in an Aspect Specification Language (ASL). ASL consists of pointcut definitions and code to execute at those pointcuts.

Contracts, sysconds, callbacks and adaptive behaviour can be packaged in a *quosket* (Schantz et al., 2002). Quoskets expose an interface and can be instantiated. Quoskets are units of encapsulation and can be reused as a component in applications.

2.7.3 DynamicTAO

DynamicTAO (Kon et al., 2000) is a CORBA compliant reflective ORB that supports dynamic reconfiguration. DynamicTAO is based on TAO, which is a flexible ORB that uses the strategy pattern (Gamma et al., 1995) to support static configuration of implementation strategies.

DynamicTAO adds support for dynamic configuration and reconfiguration by adding interfaces for transferring components across the distributed system, loading and unloading modules into the ORB runtime and inspecting and modifying the ORB configuration state.

Reification is achieved through the use of component configurators. The component configurators know the dependencies between certain components and system components. Each instance of the ORB contains a customized component configurator called TAOConfigurator that contains hooks where you can attach implementations of strategies.

Example: A DynamicTAO component can be the TAO ORB itself. This component has strategies for dispatching, connection, concurrency, etc. Each of those strategies may have different implementations that can be dynamically changed by a *DynamicConfigurator*.

The DynamicConfigurator is a CORBA object that has methods for loading, unloading, configuring and “hooking” implementations. I.e., change strategy implementations at run-time. Changing strategy implementations at run-time can cause consistency problems as the running implementations might be in a state where they cannot be stopped, or they need to pass the state to the new implementation. Passing of state to new strategy implementations uses the memento pattern (Gamma et al., 1995).

DynamicTAO also contains a monitoring service. This service is also a component that can be loaded and attached during run-time. The monitoring service uses request interceptors to monitor method calls, and to store statistics about the calls to a storage server.

DynamicTAO does not have any mechanism for doing the actual adaptation, it only provides a framework for creating such a mechanism. Creators of suchs mechanisms

may query the storage server for statistics to determine when to adapt, and use the DynamicConfigurator to perform the actual adaptation.

2.7.4 OpenCOM and the Lancaster Experience

The University of Lancaster has a been working with reflective middleware for many years. Blair et al. (2004) give a brief description of their experience and results. The rest of this section will go a little deeper and describe the parts of their research that is of particular relevance to this thesis.

2.7.4.1 ADAPT

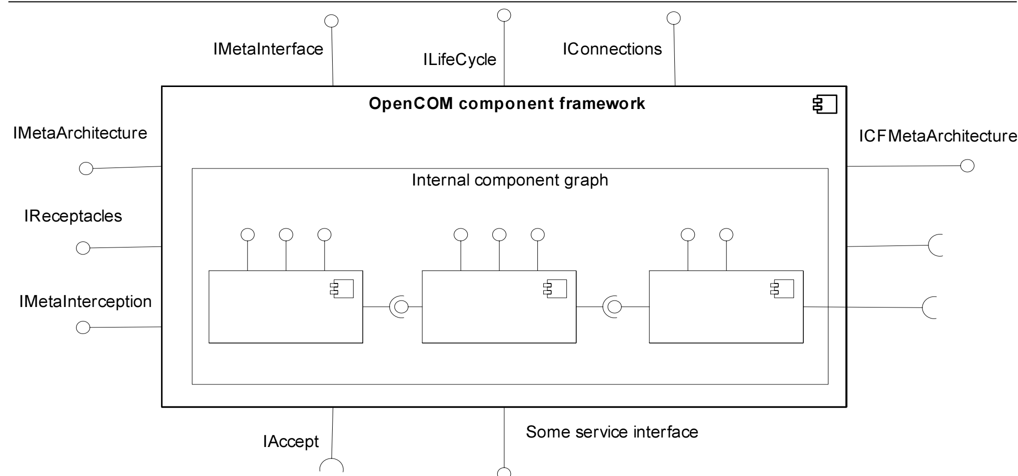
The ADAPT project (Fitzpatrick and Blair, 1998) is one of the early experiments with reflective middleware at Lancaster. ADAPT is a distributed middleware based on CORBA. ADAPT introduced the concept of *open bindings*. Open bindings provide a meta-interface that gives access to an object graph representing the underlying end-to-end communications path. This component graph is reflective, i.e., you can inspect and modify it at runtime. The reflective capability is used for dynamic reconfiguration and adaptation to achieve QoS. Monitoring resources to determine when to adapt is done with an interceptor-like event mechanism. In addition to the open bindings and reflective capabilities, it extends CORBA with streaming interfaces.

2.7.4.2 OpenCOM and OpenORB 2

ADAPT was succeded by the Open ORB project, which again was succeded by the OpenCOM and OpenORB 2 projects.

OpenCOM (Coulson et al., 2002; Clarke et al., 2001) is a reflective component platform based on a subset of Microsoft COM. Key concepts in OpenCOM are components, component frameworks and reflection.

Figure 2.5 OpenCOM component frameworks.



Fundamental concepts of the OpenCOM component model are interfaces, recepta-

cles and connections. Connections are bindings between interfaces and receptacles.

OpenCOM's reflective capabilities allows you to introspect and change the binding of components, thus altering the component graph. This is used for dynamic reconfiguration and adaptation.

Composite components are realized with *component frameworks*. A component framework is a collection of rules and contracts that govern the interaction of a set of components. A component framework is a component itself. It contains an internal structure of components that implements the service functionality provided by the component framework. Component frameworks can be nested, thus creating a hierarchical composition.

Figure 2.5 shows a component framework in OpenCOM. The meta interfaces and connections interfaces allows for introspection and structural reflection. To ensure integrity when altering the component graph, the *IAccept* interface is used. This interface can veto changes in the configuration. Monitoring code can be injected as interceptors to determine when to adapt.

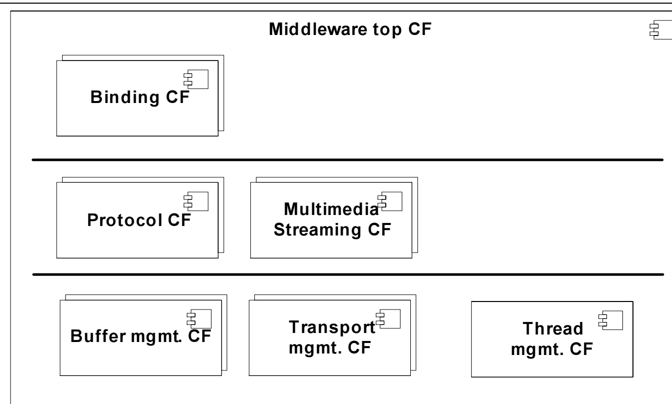
OpenCOM provides three reflective meta-models:

Interface meta-model supports dynamic discovery and invocation of the set of interfaces defined on a component.

Architecture meta-model enables discovery and adaptation of an underlying component framework.

Interception meta-model supports dynamic interception of method calls on interfaces.

Figure 2.6 Open ORB architecture.



OpenORB 2 (Blair et al., 2001) takes the research done in the OpenORB project further. OpenORB was implemented as a prototype in the Python programming language. OpenORB 2 is implemented in C++ with OpenCOM.

Figure 2.6 shows the architecture in OpenORB 2. OpenORB 2 has a layered architecture where each layer has a set of component frameworks. The three layers are: binding, communication and resource layer.

Each component framework in a layer is only allowed to access interfaces provided by components in the same or in a lower layer. The top level OpenORB component

framework is responsible for managing lifecycle and dependecies between component frameworks.

As OpenORB 2 is implemented with OpenCOM, it supports dynamic reconfiguration. However, dynamic reconfiguration of established bindings is not directly supported through the binding framework. This is a responsibility of the individual binding types.

ReMMoC (Grace et al., 2003) is a later project at Lancaster that utilizes OpenORB 2 component frameworks and OpenCOM. ReMMoC is a reflective middleware platform that adapts its binding and discovery protocol to allow interoperation with heterogeneous services. ReMMoC is aimed at mobile computing, thus it has only two component frameworks to reduce the size of the implementation. Those component frameworks are: binding framework and service discovery framework. Both these frameworks can be dynamically reconfigured.

2.7.5 Enterprise JavaBeans

EJB (EJB 2.1) is a component architecture that supports separation of some extra-functional concerns. Those concerns include security, clustering and failover, remoting[9] and declarative transaction demarcation.

To achieve this, the EJB specification defines a rigid contract that component implementors must fulfill. Class- and method names are semantically important, you are not allowed to use static variables or use synchronization primitives etc.

EJB has three kinds of components:

Session Beans usually acts as façades to a system. In a service oriented architeture they will usually expose the service interface. Session beans can be stateless or stateful.

Entity Beans represents persistent entities. Entity beans' persistence can either be container managed (CMP) or bean managed (BMP).

Message Driven Beans are asynchronous beans that listen to a Java Message Service queue or topic. JMS is the J2EE specification for message queues.

Sun has published a catalog of blueprints[10] containing guidelines and patterns for how to use EJB in enterprise applications.

To create an EJB component, you first have to define the interface it provides. This interface is the most important part of the contract that specifies the component. This interface is specified in the components *remote* or *local* interface. You have to specify both a remote and local interface if you support both local and remote invocation.

You also have to create a *home* interface. The home interface defines create-methods for instantiating a component. The home interface is also separated in a remote and local interface.

Finally, you must implement the component itself. This is called the *bean* class and contains implementation of the create methods defined in the home interface and the

[9] Remoting is only a partially separated concern. When you access an EJB component you must either access its *local* or *remote* interface. This shortcoming is possible to overcome, but you have to implement it yourself.

[10] See http://java.sun.com/blueprints/enterprise/index.html

business methods defined in the remote interface. It must also implement some lifecycle management methods (activation, passivation and removal).

The contract between those classes are informal. I.e., you have to follow naming conventions to make it work. A *createFoo()* method defined in a component's home interface must have a corresponding *ejbCreateFoo()* method in the bean class. There are also no formal contracts between the local- and remote interface and the bean implementation class. Adherence to the contract are discovered during runtime and results in exceptions if a client calls a method declared in a remote or local interface that is not implemented in the bean implementation class.

Concerns such as security and transaction demarcation are defined in a deployment descriptor, and the EJB framework is responsible for realizing those concerns. To realize this, the EJB framework intercepts the incoming method calls to the components and handles security checks and transaction demarcation in the interceptors.

2.7.5.1 JBoss AOP

JBoss[11] is an open source implementation of a J2EE container.

JBoss has taken the separation of concerns one step further than the standard J2EE containers. They have defined an AOP framework with reusable aspects that handles the concerns normally taken care of by a J2EE container. Thus, they make it possible to separate those concerns without having to create EJB components. The framework may be used without the rest of the JBoss J2EE implementation and can be downloaded separately.

The JBoss AOP framework comes bundled with some reusable aspects. This includes aspects for transaction demarcation, remoting, clustered remoting and caching.

2.7.6 IoC frameworks: The Spring Framework

The J2EE specification and especially the EJB specification has been criticized for being too heavy-weight. There are popping up open-source component frameworks that are trying to make it easier and more lightweight to create enterprise applications. The Spring Framework[12] is one such framework.

The Spring Framework is based on the *Inversion of Control* pattern, or *Dependency Injection* (DIP) as it is called now. DIP is also called "The Hollywood Principle" – don't call us, we call you.

The DIP pattern is detailed on Martin Fowlers homepage[13]. The idea behind DIP is that you use design by interface. When objects collaborate, the collaborators are interfaces. Determination of which concrete interface implementation to use is handled by the framework. The object that needs collaborators thus have mutator metods (e.g., *setCollaborator(ICollaborator coll)* for the collaborators, or collaborators as part of the constructor. The framework is also responsible for instantiating objects. The composition of collaborating objects can either be declared programatically, or in a configuration file outside the code.

[11] http://www.jboss.org/
[12] http://www.springframework.org/
[13] http://martinfowler.com/articles/injection.html

Spring provides a simple AOP framework. This is mainly a means of inserting hooks into the composition. However, Spring is designed to be used with other AOP frameworks such as AspectWerkz and AspectJ. It has well defined points and flows for your pointcuts.

Even though the AOP framework is simple, it still enables Spring to separate some cross-cutting concerns. This includes remoting and declarative transactions. Neither the component implementation nor the user of the component need to know if it is a local Java object, remote SOAP service or EJB session bean. This is set up in the configuration file.

Spring has some commonalities with the Fractal component model (described below). A Spring component type is defined in its interface, and the *required* interfaces are specified as *set* methods on the interface. Which component implementation to use, and the bindings between components, are specified in a configuration file similar to Fractal's Architecture Description Language.

2.7.7 Other Middleware Approaches

The Fractal project (Bruneton et al., 2002a) has a component model that supports composite components through a recursive model (a component can have other components as content) and sharing of components. A Fractal component has a controller part and a content part. The controller part contains interceptors and controllers and supports structural reflection. The content part contains component implementation or other components.

The Fractal core contains binding, content and life-cycle controllers. Configuration is done programatically with the binding controller, or declaratively with the Architecture Description Language, which is expressed in XML. Dynamic reconfiguration is supported through the use of reflection and life-cycle management.

The DREAM component architecture (Leclercq et al., 2004) is part of the Sardes project at INRIA and is based on the Fractal component model. DREAM focuses on asynchronous middleware and resource management. Asynchronous messages are published through a component's output interface and received in an input interface. Messages pass through message managers. This makes it possible to control the amount of messages sent to an input interface based on the available resources.

OpenCorba (Ledoux, 1999) is a reflective CORBA ORB written in NeoClasstalk. NeoClasstalk is a Smalltalk implementation with explicit metaclasses. OpenCorba uses this to provide explicit metaclasses for parts of the broker. This reification expose some of the details of the broker that otherwise would be hidden for the users. Explicit metaclasses are also used to add dynamic adaptability in the invocation mechanism.

Other work in the field include the Arctic Beans project at the University of Tromsø (Andersen et al., 2001), which focuses on separating transactions and security concerns in middleware, and the FAMOUS (Hallsteinsen et al., 2005) project at SINTEF, which focuses on adaptive middleware for mobile devices.

2.7.8 Separation of Concerns using AOP

There has been a lot of research on how AOP can help separating cross-cutting concerns. E.g., Nordberg (2001) shows how to reduce coupling in the visitor and observer patterns with aspects. Hannemann and Kiczales (2002) takes this a step further, and goes through all the GoF (Gamma et al., 1995) patterns. Their study shows that 17 of the 23 patterns improve in terms of modularity when they are rewritten to use aspects. This experiment has later been repeated and verified in a quantitative study using other metrics (Garcia et al., 2005).

There has been few larger case-studies showing that AOP actually does help with separation of cross-cutting concerns in real world projects. I.e., projects that involve more than a few lines of code or theoretical analysis only. Some of the studies that have applied AOP in larger case-studies are presented here.

Coady and Kiczales (2003) used AspectC to separate concerns in the BSD operating system. Four concerns were separated:

- Page daemon wakeup.
- Prefetching for mapped files.
- Disk quota.
- Device blocking.

Their study shows that by using AspectC they achieved localized changeability, explicit configurability, reduced redundancy and subsequent modular extensibility with a minimum loss of performance.

Papapetrou and Papadopoulos (2004) did a case-study where they created two versions of a component-based web-crawling system. One using conventional object oriented techniques, and one using aspect oriented techniques. Their study showed that the aspect oriented version was developed faster and had more modular code. More modular code promoted reuse of the original components, made the code less error-prone and easier to debug. It also made configuration of the system easier by making it possible to enable or disable extra-functional concerns.

Zhang and Jacobsen (2003) did a case study where they applied aspect oriented refactoring to CORBA. They created an aspect mining tool to analyze source code for scattered code. The mining results were used to refactor the ORBacus CORBA implementation using AspectJ. Their study shows that the refactored implementation had reduced coupling and improved modularity. Based on this work, they have developed some principles for horizontal decomposition which are applied to middleware (Zhang and Jacobsen, 2004).

Colyer and Clement (2004) have applied AOP in large-scale environments. They applied aspect oriented techniques to analyze the IBM Websphere Application Server. By using AspectJ and the aspect oriented tool support in the Eclipse IDE, they were able to identify and completely separate the EJB concern from Websphere.

Chapter 3

Tools and Techniques

This chapter provides an overview of the most important tools used in this thesis. An overview of the QuA Java prototype and some of the most important concepts in the QuA architecture are presented together with a description of how to use them. There is also an overview of the two most mature AOP frameworks together with some simple code examples to give a better understanding of how AOP works.

3.1 The QuA Java Prototype

The QuA Java prototype used in this work is implemented by Øyvind Matheson Wergeland as a part of his master's thesis (Wergeland, 2005). This section contains a brief overview of Wergeland's prototype implementation and how it can be used. See (Wergeland, 2005) for details about the implementation[1].

The goal in Wergeland's thesis is to investigate *service planning* in QuA. This means that most of the parts of QuA that are not strictly necessary for service planning are implemented on an ad-hoc basis.

Figure 3.1[2] shows a UML static structure diagram of the QuA Java prototype. The main entrance to the system is the *QuA* object[3]. The QuA object contains static methods for initializing the *JavaCapsule*, instantiating blueprints, creating service compositions, etc.

The intention is that all services run in a separate *ServiceContext*, but the current implementation run all the services in the same service context.

The ServiceContext knows which *planners* to use. Planners are specified with Java interfaces, and what implementation to use is determined by a configuration file.

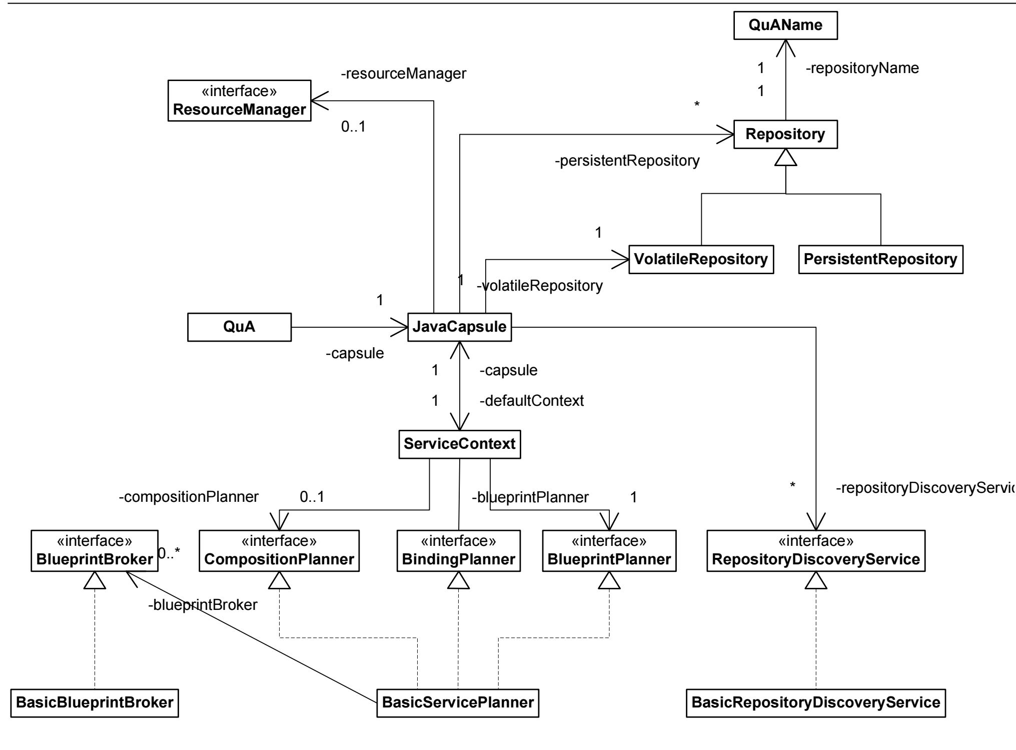
CompositionPlanner is responsible for creating services, i.e. composing components based on a service specification and a quality specification to form a service.

[1]The QuA architecture has changed after Wergeland's implementation. See the QuA architecture description at the QuA documentation pages (http://www.simula.no:8888/QuA/55) for a description of the current architecture. There also exists an updated Java prototype that reflects the architecture updates. However, it is Wergeland's implementation that is used in this work.

[2]Figure 3.1 is adopted from (Wergeland, 2005).

[3]In more recent versions of QuA this is called the *QuAMOP*.

Figure 3.1 The QuA Java prototype.



BindingPlanner is responsible for resolving QuANames (e.g., resolving remote blueprints) and creating bindings.

BlueprintPlanner is responsible for finding the best blueprint for a given quality specification. The name *ImplementationPlanner* is also used for this planner in different versions of the QuA prototype.

Blueprints are contained in repositories where they can be discovered by the repository discovery service. Repositories are usually zip files (the persistent repository) or cached in memory (the volatile repository), but it is also possible to load blueprints from remote repositories using QuA Remote Access Protocol (QRAP).

Blueprints are identified by a *QuAName*. QuANames identify all platform objects in the distributed object namespace and have an URI like syntax:

QuAName ::= repositoryPath["/"shortName[":"version"."fixlevel]]

The QuAName "*/qua/types/QoSAware:1.0*" identifies the QoSAware platform object in the /qua/types repository, which maps to the Java interface *qua.types.QoSAware*. The /qua/types repository is reserved for types.

When blueprints are instantiated as components, they are instantiated in separate classloaders. This is to avoid possible name clashes.

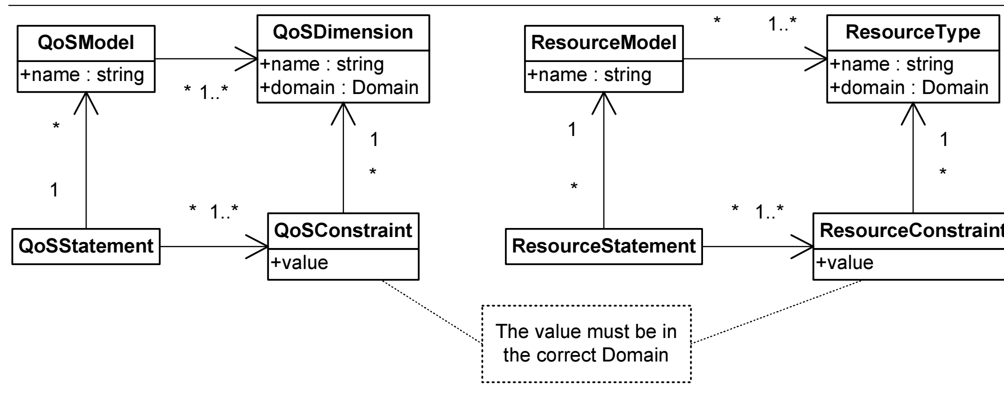
3.1.1 Resources and QoS

The model for QoS and resources are based on CQML (Aagedal, 2001). Creating a resource model for QuA is the topic for a separate master's thesis (Abrahamsen, 2005). A simplified resource model and manager based on an early version of Abrahamsens master's thesis is used in the prototype. Abrahamsens final work on resource models differs significantly from the version used here.

A resource is usually a physical resource that can be shared. Resources are described in *models*. There are different resource models for the different resources that need to be described (e.g., a PC resource model, LAN- or PDA resource models). A resource model contains a set of *resource types*. Examples of resource types are bandwidth and CPU power.

Resource types belong to a *domain*. Examples of domains are real and natural numbers. Resource types can be constrained by a *resource constraint*, which is a value for the resource type.

Figure 3.2 Resource and QoS model.



Finally, a *resource statement* is a collection of resource constraints for a given resource model. QoS aware components return resource statements when they are asked for the resources they need to deliver a certain QoS.

Examples of resource constraints are: CPU = 20% and memory = 2kB. A resource statement can contain both those resource constraints.

QoS are modeled the same way as resources. A QoS model contains *QoS dimensions* (the QoS equivalent of a resource type) that can be constrained by a QoS constraint. A *QoS statement* is a collection of QoS constraints for a given QoS model. Examples of QoS dimensions for a QoS model for sound are sample rate or simply "CD-quality", "FM stereo quality", etc. Examples of QoS constraints for sound are: Sample rate = 44.100 kHz and number of channels = 2. A QoS statement can contain both those QoS constraints.

Figure 3.2 shows a UML model of the Resource- and QoS models. The figure is adopted from (Wergeland, 2005).

The resource manager in the prototype contains the bare minimum needed to perform service planning. I.e., it knows how much resources that are available, and resources can be reserved or released. Reserved resources are local to the running service (on the local host). There is no interaction with the operating system or use of reserva-

tion protocols such as RSVP.

3.1.2 Service Planning

A service is a component composition that does something, or more formally:

> *"A service is a subset of output messages and causally related inputs to some composition of objects."* (Staehli and Eliassen, 2004).

E.g., an audio streaming service consists of a composition of components that represents an audio source, an audio sink, and some components in between to encode, decode and stream the audio data.

The QuA service planner tries to form the *best* such composition based on a service specification and a quality specification. The service planner of Wergeland (2005) is based on Q-RAM (Rajkumar et al., 1997), which is a model for QoS-based resource allocation.

The service specification contains a set of *types* described as QuANames, and a specification of the bindings between the types. Blueprints contains implementations of the types, which are later instantiated as components.

The quality specification contains the QoS boundaries (maximum and minimum values), and a *utility function*. The utility function takes a *QoSStatement* as input, and returns a normalized number between 0 and 1 denoting how "good" the given QoSStatement is. QoS statements are a collection of QoS constraints based on CQML (Aagedal, 2001). The QuA architecture also includes error models and error predictors. This is not implemented in this prototype.

When a QoS aware service planner is asked to compose a service, it tries to maximize the utility within the QoS boundary given the resources available.

A service planner consists of many components, e.g., a composition planner, a binding planner and an implementation planner. The task of finding components that maximize the utility is a responsibility of the implementation planner.

The QuA core requires a basic set of components to always be available. E.g., a basic set of components to compose services must be present. Thus, a *basic service planner* without QoS aware capabilities are present as a default service planner. Wergeland's QuA prototype contains a QoS aware *generic implementation planner*. This implementation planner can replace the default basic implementation planner at run-time, which will make the service planner QoS aware.

3.1.3 Creating a QoS Aware Component

To give a better understanding of how this works, a simple walkthrough on how to create and use a QoS aware component is presented.

The simplest component to create is a *HelloWorld* component. First we have to create the type for the component. This is done by creating a Java interface, e.g., *qua.types.HelloWorld* containing only one method: *String* sayHello().

Then we must create a blueprint that contains an implementation of the type. First, we must create a Java class that implements the HelloWorld interface, e.g., *qua.hello.HelloWorld*.

If the blueprint should be part of any QoS aware service planning, it has to implement the *QoSAware* type. This means adding a method that takes a *QoSStatement* as input, and returns a *ResourceStatement* describing the resources necessary to satisfy the QoS constraints in the QoS statement.

When this is done, the Java class implementing HelloWorld and QoSAware must be packaged as a blueprint. First, all the classes involved in the component are packaged in a jar file that contains a description of what the main-class is, then the jar file is packaged as a blueprint.

Blueprints contain additional meta-information:

Shortname is a short name for the blueprint that can be used as a reference for it, e.g., "HelloWorld".

Version describes the version number of the blueprint, e.g., "1.0".

Platform describes which platform this blueprint can be instantiated on, e.g., "Java".

Implements is a list of QuA types this blueprint implements, e.g., "/qua/types/HelloWorld, /qua/types/QoSAware". Note that in the Java prototype, QuA names maps to Java interface names.

We can create more blueprints for the HelloWorld type, e.g., *FastHelloWorld* and *ResourceIntensiveHelloWorld*, with different implementations of the *QoSAware* type.

After the blueprint is placed in a repository where QuA can find it (usually a zipfile), we can start using it.

3.1.4 Composing QoS Aware Services

To compose a QoS aware service, we must create a service specification. An example of a service is a composition of two components where one component produces something that the other component consumes. Assume that the QuAName "/qua/types/Producer" denotes the type for the producer component, and that the QuAName "/qua/types/-Consumer" denotes the type for the consumer component. Further, assume that the Producer type has a method, *setConsumer()*.

The following code will create a service specification for the service:

```
ServiceSpec spec = new ServiceSpec();
spec.addInstanceOf(QuA.name("/qua/types/Producer"), "producer");
spec.addInstanceOf(QuA.name("/qua/types/Consumer"), "consumer");

spec.addBinding("producer", "setConsumer", "consumer");
```

The service specification requires two components; one component implementing the type "/qua/types/Producer", and one component implementing the type "/qua/types/Consumer". The producer component is given the *role name* "producer", and the consumer component is given the role name "consumer".

The binding between the components is specified based on role names. The "producer" role has a collaborator playing "consumer" role, and the "setConsumer" method must be called on the producer (with the consumer as parameter) in order to bind the components.

In order to create a QoS Aware service, we also need a *quality specification*. A quality specification denotes the QoS requirements of the user of the service and consists of an implementation of the QualitySpec interface. This interface has three methods:

maxQoS This is a QoS statement returning the maximum QoS required by the user of the service. An example of maxQoS is “delay=20ms”. If it is possible to achieve better delay, it is of no importance to the user of the service.

minQoS This is a QoS statement returning the minimum QoS required by the user of the service. An example of minQoS is “delay=100ms”. If it is not possible to achieve a delay less than 100ms, the service is of no use to the user of the service.

utility This method takes a QoS statement as input, and returns a number between 0 and 1 describing how “good” the QoS statement is for the user of the service. A return value of 1 denotes the best QoS – i.e., maximum QoS is reached – and a return value of 0 denotes minimum QoS.

To compose the service, the *compose* method in the QuA service context is used. The service context delegates this call to the *planners* configured for the context. Figure 3.1 on page 34 shows a UML static structure diagram of the QuA prototype. This figure shows that the service planner consists of a composition planner, a binding planner and a blueprint planner.

The service planner will compose a service that *maximize* QoS for the given service specification. This means that it will select implementations for the producer and consumer components that *maximize the utility* function in the quality specification. The resources required by the components will not exceed the available resources in the system. Further, the service planner will instantiate the component blueprints, and bind the according to the bindings described in the service specification.

3.1.5 Actors and Roles in QuA

The *platform independent model* (PIM) for QuA contains use cases for tasks related to QuA[4]. Those use cases contain actors that are responsible for the different tasks. The following actors are described in the PIM:

Component Developer This is a person that develops component blueprints. Examples of such component blueprints are audio encoders and decoders.

Platform Developer This is a person that develops components for the *QuA platform*. An example of such a component is a new specialized service planner.

Application Developer This is a person that develops applications that use QuA.

Platform Deployer This is a person that deploys applications on QuA. This person is responsible for configuring the QuA middleware to suit the applications’ needs, and for deploying the application on QuA.

QuA Client This is any software that uses QuA, e.g., an application that uses QuA components.

[4]The PIM is available from the QuA documentation pages: http://home.simula.no:8888/QuA/55

Service User This is the person that uses applications, or services, deployed on QuA.

The actors describe the different *roles* a person can play. The same person can both be a component developer and a platform developer.

In addition to the roles described above, it is desirable to separate *QoS Expert* as a distinct role. It should be possible for a *component developer* to develop component blueprints without having to also be a QoS expert. The QoS expert provides implementations of the *QoSAware* interface used for describing the mapping between QoS requirements and resource demands for component blueprints, and implementations of the *QualitySpec* used when composing QoS aware services.

The role of QoS expert resembles the *qoskateer* role desribed in QuO (Schantz et al., 2002).

3.2 AOP Frameworks

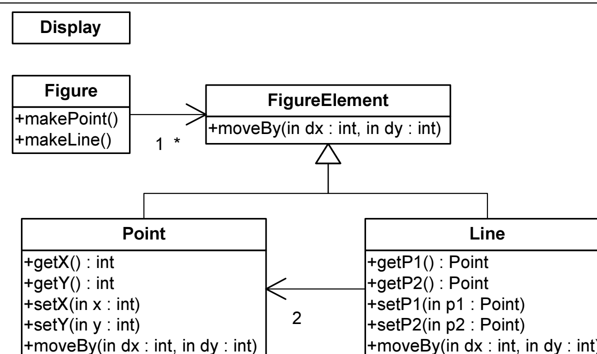
After AspectJ started to gain momentum, there has been released many AOP frameworks. Many of them are special purpose frameworks for research or for use in other frameworks, some are only proof-of-concept implementations[5]. Two of the most mature general purpose AOP frameworks are AspectJ (Kiczales et al., 2001) and AspectWerkz (Bonér, 2004).

This section will give a brief presentation of those two frameworks and point out the most important features that separates them[6].

3.2.1 An AOP Example

To discuss the two frameworks, a simple example will be used. The example is of no particular relevance to this thesis, but it is a much used example and is part of the AspectJ documentation. It has also been used in many papers and magazine articles.

Figure 3.3 A simple figure editor.



[5] See http://www.aosd.net/technology/ for a comprehensive overview of AOP frameworks.

[6] For more up-to-date information, see the AspectJ homepage (http://www.eclipse.org/aspectj/) or the AspectWerkz homepage (http://aspectwerkz.codehaus.org/).

Figure 3.3 shows a UML diagram of a simple figure editor. A *Figure* consists of *FigureElements* that can be either *Points* or *Lines*.

Consider the concern of detecting when the display needs to be updated. The display needs to be updated if a point or line moves, i.e., if a set or moveBy operation is called on a Point or Line object. This is a *cross-cutting* concern as it spans multiple objects.

Assuming that the Display object has a static method, *needsUpdating()*, a straightforward implementation of this concern would be to add *Display.needsUpdating()* to all the mutator methods in Line and Point. This means that the implementation of the concern would be *scattered*, i.e. spread over multiple objects – and it would be *tangled* with the other code in the mutator methods.

AOP tries to modularize the cross-cutting concern, and the next two sections looks at how this can be done with AspectJ and AspectWerkz.

3.2.1.1 Example Implementation with AspectJ

In AspectJ, aspects are first-class entities. An aspect is declared in much the same way as a class, and it has to be compiled with a specialized compiler.

Common for all aspect frameworks is the elements they use to separate cross-cutting concerns. Those elements are (Elrad et al., 2001):

1. A join point model describing the hooks where enhancements may be added.
2. A means of identifying join points.
3. A means of specifying behaviour at join points.
4. Encapsulated units combining join point specifications and behavior enhancements.
5. A method of attachment of units to a program.

Example 3.2.1 shows how this is utilized in AspectJ. Line 1 declares the aspect in much the same way as you would declare a class. This is the encapsulated unit combining join point specifications and behaviour enhancements.

Example 3.2.1 An aspect for updating the display.

```
1  aspect DisplayUpdate {
2    pointcut moved(): call (void Point.setX(int)) ||
3                      call (void Point.setY(int)) ||
4                      call (void Line.setP1(Point)) ||
5                      call (void Line.setP2(Point));
6
7    after (): moved() {
8      Display.needsUpdating();
9    }
10 }
```

Line 2 declares a named pointcut, which provides a means for identifying join points. The join points here are whenever a call to Point.setX *or* a call to one of the other mutator

methods is issued. Pointcuts can be combined using boolean logic: *and* (&&), *or* (||) and *not* (!).

Line 7 adds an *advice* at the join points. Advice is a means of specifying behaviour at join points. This particular advice will run after any join point identified by the *moved* pointcut.

There are three kinds of advice: *before*, *around* and *after*. *Before* is executed before the pointcut, *after* is executed after the pointcut. Before and after advice can both be expressed with an *around* advice. An around advice runs instead of the code at the pointcut. This allows you to alter the behaviour at the join point.

When using around advice, a special method called *proceed* is available. Proceed does exactly that: it proceeds the execution to the pointcut, and its return value is the same at the return value in the pointcut. Proceed takes the same parameters as the methods defined in the pointcut, thus allowing you to alter the values of the parameters – or you can choose not to proceed at all. The following is an example of using an around advice to do boundary checking on points:

```
void around(Point p, int x): call(void Point.setX(int)) && args(x)
                             && target(p) {
   if (Display.checkXBoundary(x)) {
     proceed(p, x);
   } else {
     proceed(p, Display.getValidXBoundary(x));
   }
}
```

The example only checks the x value of the Point (the code would be similar for the y value). The idea is that if the x value is within the display's x boundary, then proceed as usual. If it is not inside the boundary, proceed with a value that is.

The AspectJ join point model is quite powerful. You can match when a method is called using a *call* pointcut designator or when the actual method is executed by using an *execution* pointcut designator[7].

You can also use dynamic pointcut designators that match based to the control flow of the program using *cflow* and *cflowbelow*, or you can match by argument types, field access, exception throwing, object construction, etc. See the AspectJ documentation at http://www.eclipse.org/aspectj/ for further documentation.

The pointcut designators use type or method patterns. E.g., the *call* pointcut designator on line 2 in example 3.2.1 uses the method pattern: "*void Point.setX(int)*". The patterns can contain wildcards to make matching easier. E.g., the pattern "`call (public * *.set*(..))`" matches any public method whose name starts with *set*. It is also possible to expose the arguments to the methods:

```
pointcut publicMutator(String str): call(public * *.set*(String)) &&
                                    args(str);
```

Makes it possible to access the *str* argument in an advice. Some of these features are used in example 3.2.2.

[7]The difference of a *call* and *execution* pointcut designator might not seem important at first, but there are some significant differences. E.g., if you want to trap all calls to *System.out.println()* you cannot use an execution designator as that would mean that the java.lang.System class would have to be altered by the aspect weaver – which is not allowed. There are other important differences, but it is beyond the scope of

Example 3.2.2 Improved aspect for updating the display.

```
1  aspect DisplayUpdate {
2    pointcut movedBy(): call  (void FigureElement+.moveBy(int,int ));
3    pointcut changedElement(): ( call  (*  Point . set  *(..))    ||
4                                 call (*  Line . set  *(..))    &&
5                                !cflow(movedBy());
6    pointcut moved(): movedBy() || changedElement();
7
8    after (): moved() {
9      Display . needsUpdating();
10   }
11 }
```

Example 3.2.1 does not catch all the cases when the display needs updating. If the *moveby()* operation is implemented without using the public set methods, a call to moveBy would go unnoticed by the DisplayUpdate aspect.

This is fixed in example 3.2.2. Line 2 declares the *movedBy()* pointcut and this pointcut affects all implementations of the *FigureElement.moveBy(int, int)* method, i.e., the Point.moveBy(int, int) and Line.moveBy(int, int) methods. This is achieved by using the "+" wildcard notation on FigureElement. The "+" wildcard affects all subtypes of FigureElement.

Line 3 is a rewritten version of the *moved()* pointcut in example 3.2.1. Instead of declaring the full signature of all methods, wildcards are used. A *cflow* pointcut designator is added to prevent duplicate trigging of the advice. Duplicate trigging of the advice can happen if the Point.moveBy() method is implemented by calling the Point.setX() and Point.setY() methods. If this was the case, the advice would be triggered three times: one time for the moveBy() method, and one each for the setX() and setY() methods.

The *cflow* designator simply states that calls to mutator methods in Point or Line should be part of the pointcut if the control flow is within the moveBy operation[8].

Other Important Features

An important feature in AspectJ are the concept of *inter-type declarations*. Inter-type declarations allows you to declare fields and methods for arbitrary types in an aspect. E.g., the *DisplayUpdate* aspect could declare that the Line class should have a method called "getLength()" and also provide the implementation of the method.

Inter-type declarations also allows you to change the inheritance hierarchy of classes. The *DisplayUpdate* aspect could declare that the Point class should be a subclass of java.awt.Component. It also allows you to introduce interfaces, and the interface implementation, to classes. An aspect could declare that the Point class should implement the java.lang.Comparable interface and also provide an implementation of the required compareTo(Object) method.

---

this chapter to discuss them.

[8]Note that this is the case for the Line.setPoint methods too. A call to Line.setPoint1() might result in Point.setX() and Point.setY() being called – cflow designators for this case is omitted for brevity. If we assume that the Display.needsUpdating() operation is idempotent, none of the cflow designators is needed.

The possibility of adding interfaces to classes makes it possible to use mixin-based inheritance[9]. Mixins (Bracha and Cook, 1990) is an old technique that is used for composition of classes.

In programming languages that support multiple inheritance, such as CLOS and C++, you can declare an abstract class, "MyBigServiceClass", as a subtype of many small classes – e.g., "SubServiceA", "SubServiceB" and "SubServiceC". By creating concrete implementations of the small classes, you can compose a concrete implementation of "MyBigServiceClass" simply by extending the concrete implementations. See (Szyperski et al., 2002, pp. 113–115) or (Bracha and Cook, 1990) for examples.

Introducing interfaces with aspects can be seen as a way of adding multiple inheritance to Java.

Another important feature is precedence declaration[10]. You can declare that aspect A has precedence over aspect B. This is useful in cases where the order of advice execution is important. E.g., if an encryption aspect should be combined with a password validation aspect, the order of aspect execution is important.

AspectJ Weaving

There are three stages where an aspect can be weaved:

1. Compile-time.
2. Class load-time.
3. Run-time.

AspectJ supports compile-time and load-time[11] weaving. AspectJ requires the use of a specialized compiler that supports the aspect language constructs.

When using compile-time weaving it is enough to compile your aspects and Java source (including possible jar files that might change due to weaving) using the aspect compiler. The aspect compiler will then change the byte-code in existing and new classes to weave in the aspect code. To run the weaved code you do not need any special tools, you only need to provide a small runtime library.

Load-time weaving works much the same, but the Java system class-loader must be replaced with a weaving classloader. The AspectJ weaver must also be included at run-time. Load-time weaving makes it possible to weave unmodified byte-code when it is loaded by the class-loader. This can be useful for component containers that need to weave in cross-cutting concerns as the components are deployed.

3.2.1.2 Example Implementation with AspectWerkz

AspectWerkz is similar to AspectJ in many ways. The semantics for specifying pointcuts and advice are much the same. However, there are some important differences. AspectWerkz does not define a language that needs a specific compiler for adding aspects. Aspects can be specified using plain Java classes with pointcut and advice specified either using XML or annotations.

[9] In some aspect languages this is called *introduction*, and in other languages it is called *mixin*.
[10] This feature was introduced in AspectJ 1.1, which was released in June 2003.
[11] Load-time weaving was introduced in AspectJ 1.2, which was released in May 2004.

Example 3.2.3 AspectWerkz XML definition for the DisplayUpdate aspect.

```
1  <pointcut name="movedBy"
2              type="method"
3              pattern="void FigureElement+.moveBy(int,int)"/>
4
5  <pointcut name="changedElement"
6              type="method"
7              pattern="(* Point.set*(..) OR * Line.set*(..)) AND
8                       !cflow(movedBy)"/>
9
10 <pointcut name="moved"
11             type="method"
12             pattern="movedBy OR changedElement"/>
13
14 <advice type="after" name="updateDisplay"
15           class="DisplayUpdate"
16           bind-to="moved"
17           deployment-model="perJVM"/>

// The corresponding Java code
public class DisplayUpdate {
  public Object updateDisplay() {
      Display.needsUpdating();
  }
}
```

Example 3.2.3 shows the XML declaration and Java code for the DisplayUpdate aspect.

The class attribute in the advice declaration (line 15) in example 3.2.3 points to the DisplayUpdate class, and the name attribute (line 14) points to the updateDisplay method. As can be seen, the pointcut model are much the same as in AspectJ.

AspectWerkz can also use Java annotations to declare pointcuts and advice. Example 3.2.4 shows the DisplayUpdate aspect using Java 5 annotations. If annotations are used, an XML deployment descriptor must be provided. The deployment descriptor states which classes are aspects.

To use annotations, you must use Java 5. It is possible to use annotations as comments in Java 1.4 and below, but then the code must be post-compiled using an annotation compiler.

AspectWerkz Weaving

AspectWerkz supports all kinds of weaving. It supports compile-time, load-time and run-time weaving[12]. AspectWerkz' primary way of weaving has always been load-time weaving with compile-time weaving as an option if you have no control of the Java virtual machine startup process.

There are many ways to use load-time weaving in AspectWerkz. Common for all

[12] Run-time weaving has been supported since March 2004 and has been a supported part of AspectWerkz since version 2.0RC1, which was released in November 2004.

Example 3.2.4 Using annotations to declare pointcuts and advice.

```
@Aspect("perJVM")
public class DisplayUpdate {
  @Expression(" call  void FigureElement+.moveBy(int,int) " )
  Pointcut movedBy;

  @Expression(" (* Point . set *(..)  || * Line . set *(..))  &&
                !cflow (movedBy)");
  Pointcut changedElement;

  @Expression("movedBy || changedElement")
  Pointcut moved;

  @After( "moved")
  public Object updateDisplay()  {
      Display . needsUpdating();
  }
}

<!-- The corresponding XML deployment descriptor -->
<aspectwerkz>
    <system id="tests ">
        <aspect class="DisplayUpdate"/>
    </system>
</aspectwerkz>
```

of them is that they use a customized weaving classloader. Load-time weaving is very flexible in AspectWerkz. If a class has already been weaved (i.e., the Java byte-code has been modified), it is possible to switch the advice or mixin implementation during run-time.

AspectWerkz' support for run-time weaving allows you to declare new aspects and weave them during run-time. AspectWerkz has the capability to introspect classes and identify aspects during run-time, it also allows you to deploy and undeploy aspects during run-time.

AspectWerkz uses Java HotSwap for run-time weaving, and HotSwap has some limitations. HotSwap does not allow you to do any changes that would imply changes to the class' meta-class. I.e., changing the class signature or adding fields or methods. This implies that you cannot add mixins with HotSwap as this would imply changes to the class signature.

3.2.2 Comparing AspectJ and AspectWerkz

AspectJ and AspectWerkz are similar in many ways. The most important differences are that AspectJ has a richer set of pointcut designators, better support for inter-type declarations and that it needs a special compiler, while AspectWerkz uses XML or annotations to specify pointcuts and has strong support for dynamic aspects by using run-time weaving.

To sum it up, the strengths of AspectJ are:

- Rich set of pointcut designators.
- Mature project.
- Good tool-support.

And the strengths of AspectWerkz are:

- Good support for dynamic aspect through the use of run-time weaving.
- No need for a specialized compiler.

It was recently announced[13] that AspectWerkz and AspectJ shall join forces and that the next version of AspectJ (AspectJ 5) will contain some of the unique features of AspectWerkz. Hopefully this will result in an AOP system with AspectWerkz' simplicity and support for dynamic aspects combined with the strengths of AspectJ.

[13] See the announcement at http://www.theserverside.com/news/thread.tss?thread_id=31244

Chapter 4

Analysis

4.1 Overview of the Problem

The problem with QoS is that resources are scarce and shared. If we always had enough resources, there would not be any need for taking special action in order to achieve the desired QoS.

As resources are scarce, we have to prioritize. E.g., in an audio streaming service we might have to sacrifice stereo sound to get the desired sample size. As resources are shared, the resource availability might change while the service is running. This implies that we have to adapt to the changes in resource availability to get the desired QoS.

The following issues are important when creating a QoS aware service:

Monitoring To know *when* to adapt, the resources and achieved QoS must be monitored.

Resource management In order to plan a service, we must know and keep track of the available resources.

Configuration An initial configuration of the service must be created. This includes selecting and configuring the components that are part of the service.

Reconfiguration If the resources change at runtime, the service must be reconfigured to fit the currently available resources.

Some of these issues have a cross-cutting nature. The rest of this section will elaborate on these topics and present cases for further analysis.

4.1.1 Resources and Monitoring

Lack of resources are the key issue with QoS. To handle resources computationally, they must be represented in some sort of resource model, and the availability and allocation of resources must be handled by some sort of resource manager.

There are many ways to manage resources. One approach is to *reserve* resources in some way, e.g., by using reservation protocols such as RSVP (RFC 2205). This is the

approach used by COMQUAD (Göbel et al., 2004b). QuO (Schantz et al., 2002) also supports this approach. When using resource reservation, you must ensure that you do not use more resources than you have reserved.

Another approach is to not reserve resources at all, but instead to know how much resources you need and how much resources that are available at all times – and then deal with the changes in the available resources. If you are not running a real-time operating system that allows you to reserve resources (e.g., CPU power) or are not using a network where RSVP is available – this is the most viable approach. For more information about resource management, see Abrahamsen (2005).

Monitoring is important in both approaches. Either you must monitor resource usage to ensure that it does not exceed the reserved resources, or you must monitor the available resources and achieved QoS so you can take action when the resource availablity, and thus the achived QoS, changes.

4.1.2 Configuration and Reconfiguration

A service must have an initial configuration. This initial configuration is created by the QuA service planner. The service planner of Wergeland (2005) is based on Q-RAM (Rajkumar et al., 1997), which is a model for QoS-based resource allocation. The service planner will compose a service of one or more components that maximize the service utility – i.e., it will create the "best" composition of components. This initial composition is also the initial configuration of the service.

When the resource availability change during the lifetime of the service, the service must be *reconfigured* to fit QoS requirements within the available resources. The service needs to *adapt* to the change in available resources.

In general, there are two approaches to adaptation (McKinley et al., 2004):

Parameter adaptation change program variables that alter the service's behaviour. E.g., some audio codecs support adjustable frame rate and compression level. Changing this will change the need for network bandwidth.

Compositional adaptation change algorithms or the structure of the component composition. E.g., switching to a different audio codec will alter the component composition.

Parameter adaptation only lets you tune parameters. If you want to change strategies or algorithms, compositional adaptation must be used.

For a running service, compositional adaptation might require that you transfer state from the old service to the new adapted service. In the audio example described in the introduction to chapter 2 (page 10), you want a *smooth transition* between the old and new codec. A smooth transition means that the sound does not pause or repeat itself during the adaptation.

Compositional adaptation also has to be *consistent*. If you change the audio *encoder* at the server side, you might have to change the audio *decoder* at the client side accordingly.

To decide what parameters to tune or which components to adapt, the service planner may be used. However, the QuA service planner is not designed to replan a running service, but to plan a new service.

4.1.3 Cross-cutting Concerns

Monitoring, resource management and reconfiguration are all cross-cutting concerns. Those concerns cut through each other, but they also cut through the components and the affected services.

A common method for separating concerns from the components and into the platform, is to reify method calls. Method calls between component boundaries are reified and inspected by the platform the components run on. The separated concerns are handled by acting in some way based on the reified calls. This can be handled by the platform itself, or by programs that registers as listeners on such method calls.

Most of the frameworks described in chapter 2.7 (page 24) use this approach. OpenCOM, COMQUAD and QuO all introduce a layer of indirection where they can handle concerns. To reify a method call, a meta-model for the execution model must be created. Creating a meta-model that provides enough information and capabilities to handle a big set of concerns – even concerns that was not imagined at the time of designing the meta-model – is a big and difficult task.

Example: CORBA reifies method calls as an IIOP stream with some meta data attached. Interceptors can inspect this reified call, but has limited abilities to alter the method call. Because reifying method calls as IIOP streams affects performance, this can in some cases be skipped when calling *local* CORBA objects. A method call must be reified if it should be handled by an interceptor, thus determining whether a method call should be reified or not is not trivial, and adds complexity to the ORB.

Java provides a limited reflection API, i.e., introspection only. AOP provides a means for intercepting method calls at arbitrary points of execution, and for altering behaviour at those points. With Java's introspection capabilities and AOP's intercepting capabilities, the requirements for a meta-model for the execution model is relaxed. There should be no need for the platform to provide method reification and intercepting capabilities by itself. If the platform specifies a well defined flow for method calls, and points where one can add pointcuts that AOP can add advice to, this should be sufficient to handle most of the concerns normally handled by reifying method calls.

The goal of this thesis is neither to create a state-of-the art monitoring framework, nor is it to create a state-of-the art resource management or dynamic reconfiguration framework. The goal is to see how those concerns can be separated, and whether the use of AOP contributes to modularizing those concerns. Separating those concerns allows the various roles in QuA to focus on their tasks. The component developer can focus on developing components, the application developer focus on developing applications, the QoS expert add the QoS specifications and the deployer installs the components and configures the middleware to suit the application's needs.

4.1.4 Cases for Further Analysis

The hypothesis in this thesis is that the QoS concern can be separated with AOP in the QuA platform. The term "QoS" is a broad term and covers many QoS dimensions. Different applications need QoS in different ways. Thus, validating such a hypothesis for the general case is a formidable, and maybe even impossible, task. Instead, the hypothesis can be strengthened by investigating cases that cover relevant parts of the hypothesis.

Thus, an experimental approach with case analysis is chosen as method. By analysing

concrete cases, the scope is narrowed and it is easier to achieve concrete results that can be validated.

The cases should contribute to answering:

- How to separate the QoS concern in a static composition, i.e., how to separate the concern of choosing the "best" components when forming a service.
- How to achieve dynamic adaptation. I.e., how to change or reconfigure a running service. This includes how to dynamically adapt a service composed of more than one component.
- How to separate the interaction with resource managers and monitors.

QoS in the QuA project is limited to accuracy and timeliness. Thus, cases concerning accuracy and timeliness are selected.

Interaction with resource managers and monitors are relevant for both static and dynamic QoS. In other respects, static and dynamic QoS propose different challenges. Thus, two cases are selected for analysis: One simple case with only one local component, and one complex case with a distributed composition of many components.

The simple case focuses on static QoS, and the complex case focuses on dynamic QoS and reconfiguration of an entire composition of components.

The QuA Java prototype of Wergeland (2005) is used as a basis for the case analysis.

4.2 Simple Case: Computing the Value of Pi

Choosing components for computing the value of π has been discussed as an example of service planning in internal QuA project meetings. This is a simple example for service planning as the service composition consists of only one local component.

There exists many algorithms for computing the value of π. Bailey et al. (1997) describe some of those algorithms. The different algorithms have different characteristics. Algorithm A might compute π faster than algorithm B to a precision of 20 digits, but algorithm B might compute π faster than algorithm A to a precision of 10000 digits.

It is unlikely that anyone actually need the value of π to a substantial amount of digits. According to Bailey et al. (1997) a value of π to a precision of 40 digits would be more than enough to compute the circumference of the Milky Way galaxy to an error less than the size of a proton. Nevertheless, it is a simple and suitable case for analysis.

There are two topics to analyse in this case:

1. Static QoS – this means that we have components that compute the value of π, and want to make them QoS aware – i.e., suitable for service planning.
2. Interaction with resource managers and monitors.

In the next sections, these topics will be further analysed.

4.2.1 Static QoS: QoS Aware Pi Components

Static QoS is directly supported in QuA. In this case, the user would need to specify a service specification containing the type of the π component and the required QoS. The QuA service planner will then find the best component that fulfills the QoS requirements[1].

The service planner needs to know the resources required by a component to optimize QoS. In the QuA Java prototype this implies that a component has to implement the *QoSAware* interface and declare support for the QoSAware type.

So, all a component has to do to support static QoS, is to implement the QoSAware type. This could be separated from the component implementation. Separating it from the component implementation makes it easier to experiment with different settings for the resource requirements without having to recompile and deploy the component blueprint.

This might not have big advantages in a production environment. The resource requirements for a component blueprint are usually tightly coupled to its implementation – so if the resource requirements change, it is very likely that the implementation also has changed.

There are some cases when resource requirements change independently of the implementation: If the implementation run in an environment that it has not been tested on, resource requirements might change. A new Java VM or CPU that has better performance for mathematics is an example of such an environment.

Separating the QoSAware interface is an advantage when developing and testing components. In such occasions, it might come in handy to be able to change the resource requirements without having to recompile and deploy the component blueprint. In the QuA architecture, the QoSAware interface contains more than a mapping between QoS requirements and resource requirements. It also contains *error predictors*. If we improve the implementation of the error predictor, it ought to be possible to deploy the new implementation without having to create a new component blueprint. Error predictors are not implemented in the QuA prototype used in this analysis.

Separating the QoSAware interface from the component blueprint implementation makes it easier for a *QoS expert* to handle tasks related to the QoS concern, while the *component developer* can focus on implementing component blueprints. In many cases, a QoS expert can create an implementation of the QoSAware interface without having to examine the source code of the component blueprint. Instead, measuring the component's resource demands with different quality configurations might suffice.

The QoS concern is not necessarily part of the "compute the value of π" concern, so separating those concerns can be advantageous for the sake of modularity. The QoSAware interface is usually implemented with hard-coded values in the QuA Java prototype. By separating the interface, it should be possible to switch to using external descriptor files such as the CQML$^+$ XML descriptors used in COMQUAD[2], without recompiling or redeploying the blueprints.

There are many ways to handle this. AOP is not needed to model resource demands outside the blueprint. However, the QuA Java prototype assumes that the component

[1] See chapter 3.1.3 on page 37 for an example of creating QoS aware components with the QuA Java prototype.

[2] See section 2.7.1 on page 25 for a brief description of CQML$^+$ and COMQUAD.

knows the resource demands through access to the QoSAware interface. Separating the QoSAware interface from the component would demand big changes to the prototype.

A less intrusive solution can be created with AOP. The QoSAware interface can be added to the component with an *introduction*. Introductions, or mixins, are supported by most AOP frameworks. The component blueprint also needs to provide some meta-information to the QuA core. This meta-information could be added by an *advice* in the QuA core's loading of blueprints.

A drawback with Wergeland's QuA implementation is that the service planner must instantiate the component blueprints to access the QoSAware type. This implies that all the blueprints that implement a type that is part of a service will be instantiated in the service planning process. Separating the QoSAware interface from the blueprint implementation with AOP as described above, will not remove this drawback.

4.2.2 Monitoring and Resources

Even in a simple case with static QoS only, resources must be managed in some way.

The most important resource needed for most mathematical operations, such as calculating π, is CPU power. Resources like memory or disk space are relevant for some special computations. Different characteristics of the CPU, or the Java virtual machine, might also be relevant. However, in this analysis the resource demands are simplified, and CPU power is the only resource considered.

The most relevant QoS dimensions to specify for planning the service is *precision* and *delay*. In this case, precision is the number of digits required, and delay is the timeframe allowed for computation. The components must then specify how much CPU power they need to satisfy the requirements.

4.2.2.1 Resource Management

Resource management can be as simple as a resource manager that keeps tracks of resources – i.e., it knows what resources are available at all times.

Resource management can also mean that resources are explicitly reserved and released. To do this properly for CPU, special schedulers are needed. Standard Java does not have such schedulers, but there exists a specification for real-time Java[3] that adds such capabilities. Real-time Java is not considered here.

How resources are managed is a concern that should be separated. Neither the service planner nor the components should need to know how resources are managed. Resource management should be separated from the QuA platform components, the application and the components used by the application.

In the simplest case of resource management, the concern is already well separated in the QuA Java prototype. The components need to know how much resources they require to fulfill a given QoS requirement – they do not need explicit knowledge about the resource manager. The service planner is the only part of the system that needs to know about the resource manager. However, it only needs to ask the resource manager how much resources are available.

If the resource manager explicitly reserve and allocate resources, it would be a little

[3] See https://rtsj.dev.java.net/ for more information about real-time Java.

more complicated. Then, the service planner will have to reserve resources for services it plans, and the resources will also have to be released. The resource reservation concern can be separated. This will make the service planner independent of how the resource manager manages resources.

If resources are not properly released, resources may leak. This is analogous to the problem with memory leaks, experienced in systems that have explicit memory management. The C and C++ languages are examples where this might occur.

There are various strategies to prevent memory leaks in those programming langauges. The simplest solution is to use a garbage collector. A garbage collector releases memory automatically when it no longer can be accessed. The concern of releasing reserved memory can be handled orthogonally to the binary program executables. E.g., the Boehm-Demers-Weiser conservative garbage collector[4] for C and C++ utilizes intercepting techniques similar to those used by Narasimhan et al. (1999) to override the *malloc* and *free* system calls. In modern languages, such as Java and C♯, garbage collection is part of the runtime system for the language.

The garbage collection analogy suggests that resource management could be separated from the application logic and from the blueprint implementations. By making the QuA platform components (e.g., the different planners used by the QuA core) responsible for handling resource management, both the *application developer* and the *component developer* would be relived the task on managing resources. Instead, the *platform developer* would have to implement resource management as part of the platform components. Making the platform components responsible for managing resources will tie the platform components to a specific resource manager. The platform developer should also be relieved the task of managing resources, and the platform components should not be tied to a specific resource manager.

Resource reservation is a concern that is easy to separate in the QuA Java prototype of Wergeland (2005). The service planner knows how much resources a planned service needs. Adding an aspect that extends the service planner to also reserve resources is trivial, as shown in example 4.2.1.

Example 4.2.1 shows how an *after advice* can reserve the resources required by each component instantiated as a part of the service planning process.

When reserving resources, we also have to *release* resources. The QuA Java prototype does not have any life-cycle management. If a service is created, there are no methods for stopping or terminating the service. Thus, there are no natural places to release resources reserved by a service.

If resources are reserved, the resources used by components would have to be monitored to make sure that they do not exceed the reserved amount. For the π components, this requires monitoring of the CPU used by the component. This is not possible with standard Java and is not considered here.

Resource Management and Transactions

When the service planner is planning a service, it asks the resource manager for the available resources and compares them with the resources required by the different component blueprints.

[4]See http://www.hpl.hp.com/personal/Hans_Boehm/gc/ for more information about the Boehm-Demers-Weiser conservative garbage collector.

Example 4.2.1 An aspect for reserving resources.

```
aspect ReserveResources {
  pointcut serviceIsPlanned (ServiceContext ctx,
                             ServiceSpec serviceSpec,
                             QualitySpec qualitySpec ):
        call  (ServiceContext CompositionPlanner+.compose(ServiceContext,
                                                          ServiceSpec,
                                                          QualitySpec) &&
             args(ctx, serviceSpec, qualitySpec );

  after  (ServiceContext ctx, ServiceSpec serviceSpec,
          QualitySpec qualitySpec) returning:
        serviceIsPlanned (ctx, serviceSpec, qualitySpec) {
      ResourceManager mgr = ctx.getResourceManager();
      // Pseudo-code
      foreach  (component in serviceSpec) {
          mgr.reserveResources ((( QoSAware)component).getConfiguredResources());
      }
  }
}
```

The resource information is used to compose a service. After the service is composed, an aspect, or the service planner itself, can reserve those resources. If another thread is using the resource manager there may not be enough resources left after the planning process is finished, and we must plan the service once more. E.g., the aspect in example 4.2.1 does not account for the situation that the resource manager might not have enough resources left.

Thus, access to the resource manager should be *transactional*. From the point where the service planner starts to plan a service, to the point where it has finished planning, it should have exclusive access to the resource manager.

Adding synchronization primitives to achieve this is not complicated. AOP has a long tradition for resolving synchronization issues. This goes back to early work with Compositon Filters and the Sina language (Aksit et al., 1992).

Example 4.2.2 shows how such an aspect could be implemented. It starts by serializing access to all methods on all objects that implements the ResourceManager interface. This is done by adding a lock using the Java *synchronized* primitive on the resource manager object. Then, transactional access for the service planning process is ensured by aquiring the same lock during service planning.

This aspect could also be enhanced to support handling of error conditions in the planning process. E.g., if the service planner throws an exception, the aspect could restore the state of the resource manager to the state it had before the planning process began.

To summarize, resource management is not a cross-cutting concern in QuA unless resources are reserved. When resources are reserved, the concern is tangled with the service planner. This can be untangled and separated with the use of AOP, making the service planner and resource manager independent of each other.

Transactional resource reservation is not considered in the QuA platform, but can

Example 4.2.2 Transactional resource reservation.

```
aspect TransactionalResourceReservation {
  pointcut resourceMgrAccess(ResourceManager mgr):
        call (* ResourceManager+.(..)) &&
        target(mgr);

  /* Synchronize all access to the resource manager */
  Object around(ResourceManager mgr): resourceMgrAccess(mgr) {
      synchronized(mgr) {
          return proceed();
      }
  }

  /* Lock all other access to the resource manager
   * pointcut serviceIsPlanned is defined in the previous example */
  Object around (ServiceContext ctx, ServiceSpec serviceSpec,
         QualitySpec qualitySpec ):
        serviceIsPlanned (ctx, serviceSpec, qualitySpec) {
      ResourceManager mgr = ctx.getResourceManager();
      synchronized(mgr) {
          return proceed();
      }
  }
}
```

be handled by an aspect.

4.2.2.2 Monitoring

There are two aspects of monitoring: *Resource usage* and *achieved QoS*. Monitoring resource usage is part of resource management. Monitoring achieved QoS is a separate task.

Computing the value of π is a discrete operation. This means that monitoring achieved QoS while the component is computing π is non-trivial.

The relevant QoS dimensions in this case are *delay* and *precision*. If delay is 10 seconds, the only way to monitor this would be to check if the component is finished with computing π 10 seconds after it started. If the components should be monitored continuously, they would have to be able to report progress and estimates on how much time they need to complete.

Monitoring precision is practically impossible. E.g., if precision is 10 decimals, a QoS monitor can inspect the returned result. If the result is 3,14159265 – the monitor must have knowledge of the correct answer to conclude that the returned precision is off by 2 digits. It can inspect the result to determine if precision is too high, but detecting that the QoS is too good is of little value.

In this case, monitoring is of little value. What actions should be taken based on the results of QoS monitoring? In continuous services, such as a streaming audio service, the results can be used to determine if the service needs to adapt. In a discrete service, the results is only useful to determine whether the service finished within its QoS con-

straints.

Even if it in this case is close to useless, monitoring delay can be done with AOP. An aspect can intercept the component method for computing the value of π, measure the time used, and report whether it was within its QoS constraints to a QoS monitor.

A more sophisticated aspect could start a thread when the component was invoked, and report violation of the QoS constraints if the component is not finished before it should. E.g., if the constraint on delay is 10 seconds, the aspect could report to a QoS monitor if the component is not finished computing the value of π after 10 seconds.

Even though monitoring QoS is of little value in this case, the example of monitoring delay suggests that AOP can be used to separate this concern.

4.2.3 Dynamic Behaviour: Adding a Cache

Dynamic QoS is not within the scope of this case. As seen in the previous section on monitoring, it would also be a poor case for dynamic QoS as computing the value of π is a discrete operation.

However, there is one kind of dynamic behaviour that is relevant for this case. If the value of π is computed to a given precision once, it is no need to use any additional resources to compute it once more. Thus, the computed value should be cached.

Where should the cache be implemented? If each of the blueprints implemented it, the value of π would be cached for each invocation on the same component. If the value of π is needed in another service, possibly used by the same application, a new blueprint would be instantiated – with no knowledge of the cached value.

If caching is implemented in the blueprints, the cache would not influence subsequent service planning and thus be of little value. Also, if caching is implemented as part of the blueprint, every blueprint will have to implement it.

Caching could be implemented as a part of the application, but that would lead to scattered code with more complexity.

A cache is a typical cross-cutting concern. It is neither natural to handle it in the application, nor in the blueprints. Caching is a cross-cutting concern that should be separated.

Caching the value of π cuts through more concerns than those implemented by the blueprints and the application. In a QoS aware environment, caching also affects resources and service planning.

If resources are reserved, the resources reserved by a π service must be released if the outcome is cached. If a new service that needs a component for calculation of π is needed, the cache should be considered in service planning.

Implementing a cache for π components in the QuA Java prototype without the use of AOP is not trivial. Code for handling it will have to be added to the blueprints and the service planner – in addition to implementing the cache itself. One would also have to make sure that the cache implementation for π components does not interfere with other components.

Adding a special cache for π components to the service planner is not desirable. If a cache should be added at all, it must be a general cache that can be used for caching more than the value of π.

With the use of AOP, adding a cache is simple. The cache does not need to be gen-

eral, it can be a special cache whose only purpose is to cache the value of π to a certain precision. Implementing a special cache is a much simpler task than implementing a general cache.

An aspect implementing a cache would have to:

- Intercept invocations to π components to get the value it should cache.
- Release resources allocated with the resource manager if resource allocation is used.
- Make sure that the service planner knows that a cached value of π exists – i.e., that a "component" can provide the value of π up to a certain precision for free. This can be achieved by advertising a component blueprint that interacts with the cache.

This assumes that caching π is "free" regarding resources. If π is needed with great enough precision, say a billion digits, the cache itself would need resources such as memory or disk space.

4.2.4 Aspect Deployment

One of the goals of the QuA project is to enable *safe deployment* of QoS-sensitive applications. Assuming that QuA guarantees safe deployment of components, how can this property be guaranteed with aspects?

Also, how should the aspects be deployed? QuA package a component blueprint as a binary file. Can aspects also be packaged as binary entities, e.g., as a *quosket* (Schantz et al., 2002) as seen in QuO?

Aspect components is a topic for ongoing research. E.g, the JAsCo (Vanderperren et al., 2005) AOP framework supports aspect components. Aspects and components are also a research topic at the Northeastern University (Lieberherr, 2004).

Although an interesting and important topic, aspect deployment is beyond the scope for this thesis.

4.2.5 Summary

The analysis suggests that aspect oriented programming is most useful for resource management in this case with static QoS and components for calculating the value of π:

Static QoS AOP might be used to add QoS awareness to components that are not QoS aware – i.e., it might be used to separate the QoS aware implementation details from the component blueprints.

In the QuA Java prototype used in this thesis (Wergeland, 2005), AOP is a non-intrusive way to achieve this separation. However, there exists other QuA implementations where this separation is achieved without the use of AOP.

Resource Management Resource management is only a cross-cutting concern when resource allocation is used. In this case, it can be separated with the use of AOP. AOP can also be used to add transactional behaviour to the resource manager.

Monitoring As computing the value of π is a discrete operation, monitoring QoS is of little value in this case. The analysis suggests that AOP can be used to separate monitoring of the delay QoS dimension, but it is not clear what use there is for this information. One possible use is to make historical information avaliable in later service planning.

In addition, caching π is a natural candidate for aspects. Caching is a typical cross-cutting concern that has been part of AOP tutorials and subject to discussion in Internet forums. In environments that use resource reservation and service planning, the cross-cutting nature of caching increases.

4.3 Complex Case: A Distributed Audio Player

The case with components for calculating the value of π is a simple case as it only involves one local component. A more complex case is a distributed audio player.

An audio conference has been discussed as a case in internal QuA meetings. An audio conference can be simplified to a distributed real-time audio stream. If QuA can handle that, it should also be able to handle an audio conference system.

Figure 4.1 A component composition for an audio service.

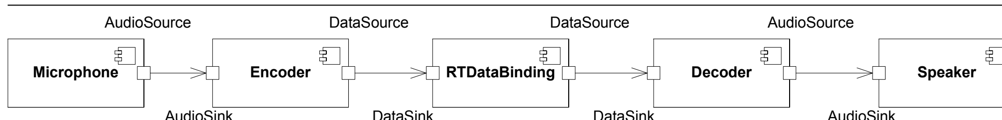


Figure 4.1 shows a component composition for an audio service. At the left is an audio source, in this case this is a microphone. The audio source is connected to an encoder component. The encoder provides an audio sink interface that the audio source can connect to.

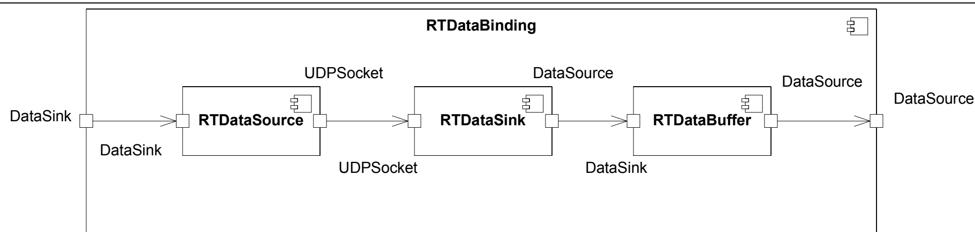
An audio encoder encodes raw audio data into another data format. Raw audio data usually takes a lot of space, if the audio is transferred over a network this implies that it will consume much bandwidth. An audio encoder compresses audio data, usually by reducing quality. MP3 and OGG Vorbis are examples of audio codecs for music, while GSM, G.723 and G.729 are examples of audio codecs for speech.

The encoder is connected to an audio decoder through a binding component. The decoder decodes the encoded data into raw audio data. The decoder is connected to a speaker component through its audio sink interface.

The RTDataBinding component in figure 4.1 is a composite component. Figure 4.2 shows a decomposition of the component. This binding consists of a real time data source that sends packets using UDP to a real time data sink. A buffer is added to the composition to absorb bursts of data and smooth out jitter. This binding is an example of a streaming binding.

The binding is the main reason for introducing QoS awareness to the audio service. If the binding is remote, the network resource is limited. With limited bandwidth, we have

Figure 4.2 A stream binding for packet data.



to compress the audio data using a codec – i.e., audio encoder and decoder. The audio encoding and decoding process reduce audio quality, and also consume CPU resources.

In addition, resources might change during runtime. Network bandwidth can decrease or increase in a shared network. CPU availability can also change. All this requires that we might have to reconfigure the audio service while it is running.

4.3.1 Audio Quality and Codecs

When sound is digitally sampled, the quality is determined by the *sample rate* and *sample size* (Tanenbaum, 2003).

Sample rate is the number of samples pr second. The Nyquist sampling theorem states that the sampling rate must be greater than twice the signal bandwidth. In most cases, the signal bandwidth is the same as the highest frequency.

Sample size is the size of the sample in bits. Typical values for sample size are 8 or 16 bits. The sample size determines the quantization errors. A sample size of 16 bits allow 65536 different values for the sample. The error introduced by the finite number of values is called the quantization error.

The human ear can hear frequencies between 20 Hz and 20.000 Hz. This means that sound must be sampled with a sample rate of more than 40.000 Hz according to the Nyquist theorem. This is roughly the same as the sample rate of audio CDs. Audio CDs use a sample rate of 44.100 samples/sec and a sample size of 16 bits.

Thus, transferring uncompressed audio CDs over a network requires a bandwidth of 44.100×16 bits/sec. Actually, it requires twice the bandwidth. Audio CDs are usually in stereo; stereo requires two channels. Thus, the required bandwidth is $2 \times 44.100 \times 16$, which equals about 1.4Mbps.

When sound is encoded, it is packetized into a *frame*. A frame contains a number of samples for all the channels.

4.3.1.1 Codecs

All digital sound is coded in some way. In section 4.3 (p. 59) the term "raw audio data" was used. Raw audio data is an imprecise term. What is meant by raw audio data is usually pulse code modulation (PCM) encoded audio. PCM is a method for encoding information on a carrier signal and is used to create digital representation of an analog sound signal. Other modulation techniques are: frequency modulation, amplitude modulation and wavelet modulation.

PCM is the modulation technique used to create digital representation of analog

sound signals. When the term "unencoded sound" is used, it usually means PCM.

PCM sound samples use a lot of space. E.g., in the previous section the bandwidth for transferring sound from an audio CD was calculated to 1.4Mbps. This implies that the storage needed for 1 hour of music is 630MB.

To reduce the need for storage space or bandwidth, music and sound are usually further encoded. A *codec* is a set of algorithms to encode and decode sound. Popular codecs for music are MP3 and OGG Vorbis. Those codecs are able to reduce the bandwidth requirement from 1.4Mbps to about 192kbps. To achieve this, the codecs are lossy. This means that the sound sample that is encoded, and then decoded, using an MP3 codec, is not the same as the initial sample. The encoding process loose some of the precision in the original sample and thus reduces the quality.

MP3 and OGG Vorbis are CPU intensive algorithms designed to encode music. Human speech tends to be in the 600 Hz – 6000 Hz range (Tanenbaum, 2003). Thus, a sample rate of 12000 is enough. There exists special codecs designed to encode human speech. Those codecs are good at reproducing speech, but give poor results when reproducing music. ITU-T has standardized a set of codecs for speech. Some of those codecs are used for digital phones and audio conferencing systems.

One of the simplest codecs are G.711. It compresses a 12 or 16 bit PCM sample to 8 bit using a logarithmic algorithm. Two versions exits: a-law (used in Europe) or μ-law (used in USA and Japan). G.711 uses a sample rate of 8000 Hz. Thus it requires a bitrate of 64kbps. G.711 samples are packetized into frames of 240 bytes. Thus, a frame contains 30 milliseconds of sound.

Another efficient speech codec is the GSM codec used in mobile phones. GSM also uses a sample rate of 8000 Hz. GSM encodes 160 samples with a sample size of 13 bits to 264 bits – or 33 octets. Thus, a frame of GSM encoded sound contains 160 samples encoded as 33 octets. This implies that a frame of GSM sound contains $\frac{160}{8000} =$ 20milliseconds of sound and requires a bandwidth of 13200bps.

As a frame of G.711 encoded sound contains 30 milliseconds of sound and a GSM frame contains 20 milliseconds of sound, it implies that the minimum latency for those codecs are 30 and 20 milliseconds.

ITU-T has also defined more complex speech codecs that has lower latency and better sample rate. Some of the complex codecs support features that reduce the bandwidth requirements:

VBR Variable bitrate allows a codec to change its bitrate dynamically to adapt to the complexity of the audio being encoded. This feature is also present in the MP3 and OGG Vorbis music codecs. VBR has the advantage that you can encode speech with the same quality as using constant bitrate, but with lower bandwidth requirements. The drawback is that you do not know the bandwidth requirements.

ABR Average bitrate solves the problem with VBR. ABR adjusts VBR quality to meet a constant bitrate requirement. The drawback is that you do not have constant quality.

VAD Voice activity detection detects silence or background noise. Silent periods can be encoded very compact. Advanced codecs also has support for "comfort noise generation" to reproduce the background noice in the silent periods.

Features such as VBR and ABR can reduce the need for bandwidth, but they require more CPU power.

4.3.1.2 Audio Quality

Sample rate and sample size is important to determine audio quality. When a speech codec is used, sample rate and sample size is not the primary factors for quality. The codec itself and the configuration of the codec is more important.

The telecommunication industry needs quantitative measures for audio quality. Thus, the ITU has specified how to quantify the subjective audio experience for speech (ITU-T P.800). This specification introduces mean opinion score (MOS) as a means for quantifying the quality. MOS evaluates audio quality on a scale from 1 to 5, where 1 is bad and 5 is excellent.

4.3.2 Real-Time Transport Protocol

For transport of real-time data, RTP (RFC 3550) is a common choice. RTP provides services for time reconstruction, loss detection, content identification, etc. RTP works in conjunction with a control protocol, RTCP, which monitors QoS and provides session control.

RTP is designed to work with datagrams, i.e., it is designed to work with unreliable protocols that can loose packets. An RTP packet contains a fixed header and a payload. When streaming real-time audio, the payload will typically be an audio packet from a codec, e.g., a frame with GSM data.

The most important RTP header fields relevant for this case are: the payload type (i.e., which codec is used), sequence number and timestamp. By including payload type in each packet, RTP makes it possible to smoothly change codec during run-time. Sequence numbers makes it easier to detect packet loss or packets that arrive out of order. The timestamp denotes the sampling instance of the first octet in the payload and can be used to calculate jitter.

In real-time applications, such as audio conferences or VoIP, latency is a very important QoS dimension. Monitoring latency is not trivial, as it requires synchronized clocks. The timestamp in an RTP packet starts with a random number that increases monotonically and can not be used for latency calculations.

RTCP provides a means for QoS monitoring and congestion control. This includes monitoring of latency and bandwidth. In an RTP session, the participants periodically send RTCP packets to each other. These packets include network time protocol (NTP) timestamps, delay and other information that can be used to calculate latency and other QoS dimensions.

Both RTP and RTCP is designed to be extensible. RTP allows additional codec information to be sent in a packet using different RTP profiles, and RTCP allows application specific packets to be sent. Although RTP encourage the use of RSVP (RFC 2205) to *reserve* bandwidth, the application specific packets in RTCP, together with the NTP timestamp and latency information, may be used to calculate bandwidth.

4.3.3 Monitoring QoS

For monitoring achieved QoS, the most relevant QoS dimensions are latency, jitter, MOS, packet loss, sample size and sample rate. Monitoring subjective quality dimensions, such as MOS, usually requires feedback from the user that listens to the sound. However, under optimal conditions, a given audio codec will have a theoretical MOS value. E.g., GSM encoded audio has a theoretical MOS value of 3.5 (Dai, 2000).

Assuming that we have the component composition shown in figure 4.1 and figure 4.2 on page 59, the above mentioned quality dimensions can be monitored as follows:

Latency This is the delay from sound enters the microphone to it is played in the speaker. Latency is not trivial to monitor in a distributed environment, as it requires synchronized clocks. RTCP, or a similar protocol, is needed to monitor latency. An aspect can intercept the planning of the audio service, and start an RTCP service in both the server and client QuA capsules. This would require that the aspect recognize the planned service as an audio service, and would be an aspect specific for audio services.

The information needed by an RTCP service is the header fields from the transport protocol. An aspect can intercept the RTDataBinding component and send the received packets to the RTCP service. If a protocol similar to RTP is used, we also need the timestamp for when the Encoder component received the first octet in a frame. An aspect could wrap the Encoder component, set a timestamp, and include this timestamp in the transport protocol.

The latency calculated by RTCP is the latency from the sound enters the microphone to it reaches the receiving RTDataSink component. To calculate the experienced latency, we also have to add the time spent in the buffer, and the time used to decode the audio packet. This can be achieved by adding aspects to the RTDataBuffer and the Decoder components. Those aspects would have to measure the time an audio packet has spent in the respective components.

Jitter This is the variance in latency. The network jitter can be monitored by intercepting insertions of data packets in the RTDataBuffer component, which is part of the RTDataBinding. Experienced jitter in the audio playback can be monitored by intercepting the data flow from the Decoder component to the Speaker component.

Packet loss This is part of the monitoring of the binding. This requires that the data packets contain a sequence number as seen in protocols such as RTP (RFC 3550). Packet loss can be monitored by intercepting packet receival in the RTDataSink component, and then check if the newly arrived packet has a sequence number that is one greater than the sequence number of the last arrived packet.

Packets migh arrive out of order. The buffer can reorder the packets before they are sent to the decoder. If we include packets that arrive too late as lost packets, monitoring packet loss by intercepting the flow from the buffer to the decoder will give a more correct measure.

Sample rate If the transport protocol contains information about sample rate and sample size, this can be monitored by intercepting the binding, e.g., the RTDataSink or the RTDataBuffer components. If not, the only place this can be monitored with certainty, is after the decoding process. Thus, we can monitor sample rate and sample size by intercepting the flow from the decoder to the speaker.

Sample rate and sample size can also be monitored on the server side of the binding by intercepting the flow from the microphone to the encoder. However, monitoring this on the server side is of little interest.

MOS To calculate theoretical values for MOS, we need to know which codec is used. Asking the Decoder component which codec it uses might seem as a sensible approach. However, the codec used might change during runtime. In RTP (RFC 3550), each packet contains a description of the codec used. Thus, monitoring which codec is used can be monitored by intercepting the flow from the buffer to the decoder.

Most of the quality dimensions above can be monitored by intercepting the program flow at one point in one component for each quality dimension. This is suitable for aspects, as we can add advice at those points. The exception is latency. Monitoring latency requires interception of the program flow at multiple points and the initiation of an RTCP like service. Investigating the details of RTCP is beyond the scope of this thesis.

It is possible to monitor these quality dimensions without the use of AOP in QuA, but not with a clear separation of concerns. Some of the quality dimensions could be monitored by the binding, but this would tangle the monitoring concern with the binding concern.

We could also use connector components (Aksit and Choukair, 2003) – a kind of "glue" components that are placed between the other components. Those components can receive data, inspect the data, and pass it on to the next component. For example, the behaviour added in an aspect's advice could be added by a connector component. As all the components that are part of a service are specified in a service specification, this would require the connector components to be part of the service specification, and thus tangle the monitoring concern with the service description.

It might be possible to achieve a clear separation of concerns without AOP if QuA had full support for reflection. E.g., the meta-models of OpenCOM[5] could be used to achieve this. The QuA Java prototype does not support reflection, and the point of using AOP is that we can achieve this without having to create a reflection framework.

4.3.4 Resource Management

Resource management in this case is in many ways similar to resource management in the simple case discussed in section 4.2.2.1 on page 53. The aspects for separating resource reservation also applies in this case, i.e., resource reservation and transactional access to the resource manager can be separated using AOP.

[5]See section 2.7.4 on page 28 for a description of OpenCOM.

There is one important difference between the cases; in this case, the service spans over two capsules in a distributed system. The binding that connects the two capsules uses a shared network resource. This indicates that the two capsules should use a shared resource manager in a common distributed address space. Alternatively, hierarchical resource managers might be used. Ecklund et al. (2002) propose hierarchical QoS managers to manage resources in distributed services.

The issue of distributed resource management is beyond the scope for this thesis. To simplify the analysis, we will regard the service as one service with two sub-services where each sub-service has separate resource management.

There are two resources to monitor in this service: CPU and bandwidth. The audio codecs consume CPU power, and the binding consumes bandwidth.

4.3.4.1 Monitoring CPU Usage

Monitoring CPU usage was discussed in the analysis of the previous case (section 4.2.2.1 on page 53). The same limitations are valid here; standard Java does not have capabilities for proper reservation and monitoring of CPU usage.

Instead of monitoring *how much* CPU power is used, maybe we can monitor if the current CPU power is *sufficient*. If an audio encoder manages to encode x seconds of audio in less than x seconds, we have enough CPU power. The same goes for the decoder, if it manages to decode x seconds of audio in less than x seconds, we have enough CPU power.

If an encoder or decoder use more than x seconds to encode or decode x seconds of sound, they use too much time and the sound will not be smooth. I.e., the sound will contain breaks between each packet of audio data. This indicates that the codec does not have enough CPU resources available.

If lack of CPU power is detected, we can adapt to a less CPU intensive audio codec. If the lack of CPU power is temporary, and we have adapted to a less CPU intensive audio codec to handle the situation, we have no method for detecting that we later regain CPU power and thus can adapt back to the original codec.

Detecting whether the codecs use too much time in the encoding or decoding process can be implemented as an aspect. By adding an around advice to the encoder's *encode* method and the decoder's *decode* method, we can measure the time used and compare it with the duration of the audio data sent to the codec.

This can be implemented directly in the component blueprints, but would tangle the resource monitoring concern with the blueprint implementations and tie the implementation to a specific resource manager. Implementing this with AOP will separate the CPU monitoring concern.

There is a problem with this approach. Detecting that a codec component do not have enough CPU resources might be caused by another component using too much CPU. If this is the case, it is the other component that violates the resource constraints, not the codec. Thus, this approach is unreliable and imprecise.

4.3.4.2 Monitoring Bandwidth

Bandwidth can be monitored using RTCP the same way as monitoring latency, as described in section 4.3.3.

RTCP packets are sent periodically. The NTP timestamp, latency calculations and size of the packet can be used to calculate bandwidth. The RTP specification (RFC 3550) recommends that a maximum of 5% of the reserved bandwith should be used for RTCP packets. This ensures that we do not interfere too much with the bandwidth used by the audio service.

4.3.5 Adaptation

To adapt the audio service means to reconfigure the service while it is running in order to fulfill the QoS requirements as resource availability changes. There are at least three issues related to this:

When to adapt? QoS monitoring might detect that the service do not fulfill the QoS requirements. To maximize QoS we also need to detect when we have gained resources and can adapt to a more resource intensive composition.

Where should adaptation occur? I.e., where should the adaptive behavior be inserted?

How to adapt? This regards both what the new composition should look like (e.g., which parameters should change and which components should be replaced), and how to perform the actual adaptation.

The main objectives for this thesis is *where* to adapt and *how* to perform the actual adaptation.

4.3.5.1 Where and When to Adapt

Regarding *where* to adapt the service, we can add a layer of indirection between the application and each of the components in the service. Adding a layer of indirection in order to adapt components is a common approach (McKinley et al., 2004). Other frameworks adds this layer of indirection by reifying method calls using some sort of interceptor technology. The reified method call can then be evaluated and possibly redirected to another component if necessary.

This layer of indirection can also be added using AOP. All components in the service can be wrapped in an aspect that:

1. Check if the component needs to be adapted.
2. Performs the adaptation, either by reconfiguring the existing component (parameter adaptation) or by replacing the component (compositional adaptation).

Example 4.3.1 shows an example of how such an aspect may be implemented. The example relies on an adaptation manager that knows if adaptation should occur, and how to adapt a component.

This approach postpones the adaptation until a method on the component is executed, i.e., the adaptation manager is passive. This approach resembles the *delegates* used in QuO (Schantz et al., 2002). A QuO delegate acts as a local proxy for an object and adds locally adaptive behavior.

Example 4.3.1 An aspect for adapting a component.

```
aspect AdaptComponent {
  AdaptationManager adaptationManager;
  Object targetComponent = null;
  // Pointcut that matches execution of any QoS aware component
  pointcut componentInvocation: execution (* qua.types.QoSAware+(..));

  Object around(Object currentTarget): componentInvocation &&
                                       this (currentTarget) {
    if (targetComponent == null)
      targetComponent = currentTarget;

    if (adaptationManager.needsAdaptation(targetComponent)) {
      // Adaptation manager will either return a new component,
      // or the same component with reconfigured parameters
      targetComponent = adaptationManager.adaptComponent(targetComponent);
    }
    // Execute the method on the component
    return proceed(targetComponent);
  }
}
```

Adapting a component can be costly in terms of CPU cycles. Thus, an active adaptation manager that can adapt a service when the service is idle might be preferred in some occasions.

Example 4.3.2 shows how this might be implemented. In this example the adaptation manager adapts components actively. The aspect only acts as a level on indirection, or proxy, for the component. In a streaming audio service there will be no idle time. Thus, only the former approach is considered here.

Example 4.3.2 Another aspect for adapting a component.

```
aspect AdaptComponent {
  AdaptationManager adaptationManager;
  // Pointcut that matches execution of any QoS aware component
  pointcut componentInvocation: execution (* qua.types.QoSAware+(..));

  Object around(Object currentTarget): componentInvocation &&
                                       this (currentTarget) {
    // Ask the adaptation manager for the component to use
    // and execute the method on the component
    return proceed(adaptationManager.getComponent(currentTarget));
  }
}
```

It is beyond the scope of this thesis to give a full analysis of *when* to adapt. In example 4.3.1 adaptation occurs when the adaptation manager has decided that it should occur.

The simplest scheme for deciding when to adapt is to ask the component how much resources it needs when a method is invoked on the component, and then compare this with the available resources in the resource manager. If there are too few resources available, we must adapt the component.

This simple scheme does not catch the situation where *more* resources become available. If we first adapt the service to use poor but tolerable audio quality due to a decrease of bandwidth, we can adapt the component to use better audio quality as bandwidth increase. The simple scheme does not detect this situation.

Poladian et al. (2004) have created a model for dynamic configuration of resource-aware services. Their approach to service planning is very close to the Q-RAM (Rajkumar et al., 1997) model used in the QuA prototype of Wergeland (2005). In their model, adaptation occurs when the *observed* utility of the service is lower than the utility from the best computed service configuration.

Strategic management (Berset, 2004) can also be used to determine when to adapt. Strategic management impose a coordinator of adaptive behavior and requires an active adaptation manager. If services are allowed to autonomously decide when to adapt, we risk that all services in a system adapts at the same time when resource availability changes. If all services adapt, the resource availability will change again, and the services might also adapt again. This can cause an unstable oscillating system. Strategic management tries to prevent this by coordinating adaptive behavior. Strategic management is also discussed by Ecklund et al. (2002), who propose hierarchical QoS managers for distributed systems.

4.3.5.2 How to Adapt

Example 4.3.1 showed an example of *where* to adapt. That example is also a simple example of *how* one might adapt a component. The aspect in the example asks an adaptation manager whether adaptation should occur. If it should adapt, it asks the adaptation manager for a new component that replaces the original.

An adaptation manager would need to know which component the original component should be replaced with, or if parameter adaptation is sufficient; how should the component be reconfigured. The ideal solution would be to ask the service planner to replan the service, but the service planner does not support reconfiguration.

To solve this without reconfiguration support in the service planner, we can ask the service planner to instantiate a component with the same type as the component we want to adapt. To make it possible for the service planner to find a component that maximize QoS, we would have to temporarily free the resources reserved by the original component. This is because the service planner finds a component based on what resources are available.

If we use this approach, we would always use compositional adaptation. If the same component blueprint is used before and after adaptation, it is a different component even if the only difference is the configuration. Compositional adaptation can be more costly than parameter adaptation. This is because we have to instantiate a component blueprint, and also transfer state and bindings from the original component to the new component.

An alternative approach is to not replace the component if the new component uses the same component blueprint as the original, e.g., if both components represents the

same codec. It might be cheaper, in terms of time and CPU usage, to reconfigure the original component with the configuration used in the new component. For this to be possible, the component must support reconfiguration – e.g., by implementing a type dedicated to reconfiguration. An implementation of a type dedicated to reconfiguration could be introduced with an aspect.

Both the approaches described above, require that we known which component that needs to adapt. The only way to detect that a component needs to adapt, is if it uses too much resources. If the service does not fulfill its QoS constraints, we do not know which component needs adaptation. Thus, we must adapt the whole service.

To adapt the whole service, we can use the approach for compositional and parameter adaptation described above. We can ask the service planner to plan a new service using the original service specification. We can then iterate through the components in the service, and adapt each component in the running service to match the component in the new service. This approach requires that the new and old service are isomorph – i.e., that the new service uses the same service specification and do not need to add or remove components to the specification.

Example: The original service uses a G.711 codec. The QoS monitor detects that the service do not fulfill its latency constraint. We then ask the service planner for a new service, this service uses a GSM codec. By comparing the two services, we detect that the codec in the original service must be adapted to use GSM components.

State Transfer and Compositional Adaptation

When compositional adaptation is used, all state and bindings from the original component must be transferred to the new component.

To transfer state between components, the memento pattern (Gamma et al., 1995) can be used. This is used by dynamicTAO (Kon et al., 2000) to transfer state between ORB strategies. However, this requires that the components are aware of the memento pattern.

The *RTDataBuffer* component is the only component in the audio service with a state that should be transferred in case of compositional adaptation. For all the other components, only the bindings need to be transferred.

To transfer bindings from one component to another, we need to know which bindings the original component has, and also which other components are bound to the component. All the bindings in a service is added to a *ServiceSpec* before the service is planned by the service planner.

In the current implementation of the QuA Java prototype, those binding specifications are not kept after the service is planned. To make the bindings available for an adaptation manger, this will require a change to the QuA implementation.

Another approach is to add some simple reflection capabilities to the component, with introspection support only. We can collect the binding specifications with an aspect as the service is planned. The service specification used to compose a service contains a description of the bindings and composition of the service Those bindings can be made available by introducing an interface implementation to the components using AOP. E.g., add an interface, *QuAService*, with a method, *getServiceSpec()*, to all component blueprints using the AOP mixin, or introduction, mechanism.

Ensuring a Valid Composition

If we adapt a component in a service, we must ensure that the service has a valid composition after adaptation has occured. In theory, this should be enforced by the type system; a component that implements a type can be replaced by another component implementing the same type. However, there may be other dependencies between components.

The encoder and decoder components in the audio service is an example of such a dependency. If we replace a GSM encoder component with a G.729 encoder component, the composition is still valid according to the service specification – but if we do not replace the GSM decoder component with a G.729 decoder component, the service will be invalid.

Blair et al. (2001) discuss a similar case in the context of OpenORB 2. They use *architectural constraints* to ensure integrity. The type system is one level of constraint, and an explicit encoding is proposed for other architectural constraints.

If explicit architectural constraints are not supported, the dependency between the encoder and decoder can be regarded as a special case of component dependencies, requiring a special solution.

As described in the section on monitoring (section 4.3.3), the transport protocol contains a description of which codec is used in each packet transfered. The client side must monitor the binding and adapt the decoder component if the codec changes.

There might be other dependencies between components. E.g., the codec components might require some pre- or post-processing components. If the codec changes, but the pre- or post-processing component do not change, the service may be invalid. To capture such dependencies, composite components should be used. E.g. an audio decoder component that requires pre- or post-processing components should create a new composite component consisting of the encoder and the pre- or post-processing component.

If composite components were used for this, we would adapt the composite as any other component, and the dependency would be solved automatically. Composite components are not supported in the QuA Java prototype.

While composite components can be used for solving dependencies between adjacent components in the service composition, they can not be used for solving the dependency between the encoder and the decoder. A special solution using aspects that monitor a change of codec was proposed for solving this dependency above. A general solution requires that all such dependencies are explicitly expressed in some way. *Architectural style* (Garlan et al., 2004) is one way expressing such dependencies. Explicitly stating all dependencies is also in line with Szyperski's component definition (Szyperski et al., 2002), which states that components should have "*explicit context dependencies only*".

4.3.6 Summary

The analysis suggests that aspect oriented programming can prove valuable for separating the QoS concern in distributed audio services.

QoS Monitoring Most of the quality dimensions that are relevant for monitoring in this case should be simple to handle with AOP. The exception is latency, which requires

a RTCP like service and more complex aspects.

Resource Management AOP seems useful for separating the resource monitoring concern. The CPU resource are not trivial to monitor using standard Java 1.4 whether AOP is used or not. Monitoring bandwidth requires the same RTCP like service as monitoring of latency.

Adaptation AOP seems useful for adaptation of components. AOP can be used to intercept invocations on components. The invocation can be redirected to other components, or the existing component can be reconfigured. If parameter adaptation is used, the components that should adapt must support reconfiguration. If compositional adaptation is used, the components must be able to transfer state. A reflective model can be used to transfer bindings when compositional adaptation is used, and AOP might be used to create this reflective model.

Key to all these topics is the need for an extra layer of indirection. AOP is particularly suited for adding such a layer.

4.4 Experiments

To validate the conclusion of the analysis, an experimental approach using prototyping is chosen as method. The analysis suggests that the QoS concern to a high degree can be separated using AOP with the QuA Java prototype of Wergeland (2005). This should be validated, and prototyping is an efficient way of doing that.

The analysis uses two cases. A simple case with static QoS, and a complex case with a streaming distributed audio service. To validate the analysis, the following should be implemented:

Simple Case Two or more components that calculate the value of π. Making the components QoS aware should be separated as aspects. Resource reservation should also be separated as an aspect.

Complex Case A streaming audio service with support for at least two different codecs. Monitoring of some of the QoS dimensions described in section 4.3.3 should be separated with aspects. Implementing a control protocol like RTCP is beyond the scope of this thesis, this implies that monitoring of latency and bandwidth will not be included. Also, the method for monitoring CPU usage is imprecise and unreliable and will not be implemented.

The most important issue is to handle parameter and compositional adaptation.

A successful implementation will strengthen the hypothesis that AOP can be used to separate the QoS concern in QuA.

4.4.1 Criteria for Evaluating Results

There must be some criteria for evaluating whether the prototypes are successful. E.g., how well does it separate the concerns? It is often claimed that use of AOP produce

more modular code. If so, how well is the cross-cutting concerns modularized, and can this modularization of cross-cutting concerns be reused for other components in other services?

The rest of this chapter will discuss criteria for evaluating the prototypes.

4.4.1.1 Reuse

Software reuse has been one of the Holy Grails of software engineering for decades. Standish (1984) refers to visions of software reuse back in the 60's. Work to promote reuse has continued ever since and is one of the main reasons to introduce components and component-based software engineering (Szyperski et al., 2002).

When it comes to reuse, history has showed us that "the proof of the pudding is in the eating". There has been many attempts at creating architectures or frameworks that enable or promote reuse. However, software is not reused until it actually *is* reused. The world is full of reusable components or frameworks that never has been reused. Thus, trying to prove that AOP leads to better reuse is a dubious task.

No matter if the software is reused or not, there are some properties of components that makes them more suitable for reuse. Evaluating how the use of AOP affects those properties should be part of evaluation of the prototypes.

Shades of Gray: Black Boxes and Encapsulation

There are many kinds of reuse. Szyperski et al. (2002) discuss some of them:

Blackbox reuse Reusing the component solely on the basis of its interfaces and their contractual specification.

Glassbox reuse Allows for inspection of the implementation of the component, but not to modify it.

Whitebox reuse Same as glassbox reuse, but also allow for modification of the implementation.

Graybox reuse Part of the component's implementation is opened for inspection and modification via inheritance.

All those shades of gray indicates the level of encapsulation. A black box has the advantage that you can easily switch the implementation of the component without having to modify the rest of your program. If the program depends on the internals of a component, switching component implementation can be problematic. If we cannot switch component implementation, evolvability of the component (e.g., providing new versions with bug fixes or better algorithms) is harder. It will also be harder for a QoS-aware application to replace components as resource availability change.

Black boxes are not always the best solution. Black boxes abstract away the implementation details, but sometimes the *user* of the black box knows best what implementation strategy to follow. Kiczales (1996) discuss this problem, and argues for an *open implementation* that allows the user of the component some degree of control over its implementation. AOP and reflective middleware are both technologies that utilize this idea.

Black, white or gray boxes – regarding the prototypes we can conclude that the less the aspects need to know about a component's implementation, the better. I.e., if the aspects only need to know about the component specification, both the component and the aspects themself are more likely to be reusable.

Aspects and Fragile Base Classes

Encapsulation is regarded as a key property of object-oriented programming. Removing this encapsulation can lead to problems. Even if an object is properly encapsulated and only exposes a controlled set of methods, they might be modified by the use of inheritance as is the case with graybox reuse.

Inheritance can be a subtle way of breaking the encapsulation. This is know as the *"fragile base class problem"* (Szyperski et al., 2002). The fragile base class problem has two interpretations, the *semantic* and the *syntactic* fragile base class problem:

Semantic Subclasses might become invalid when the semantics of the base class change.

Syntactic Compiled classes are not always binary compatible with new binary releases of superclasses.

The syntactic fragile base class problem is not related to breaking of encapsulation. The syntactic fragile base class problem is about binary compatibility as the base class evolve. This problem is easiest to spot for programming languages such as C++ where the function pointer tables of a class changes as the class changes. This breaks binary compatibility, and the subclasses needs to be recompiled to work. It is also relevant for other object-oriented languages, such as Java. If the new base class introduces a private method that is already declared in a sub-class, the application will break, and recompiling the program will not help as overriding private methods is not allowed.

AOP and the syntactic fragile base class problem are only remotely related. The problem can arise in situations where static weaving is used. If a new version of the base class is provided, one might have to re-weave all the aspects in the system. There are also some cases when using introductions or mixins can lead to the syntactic fragile base class problem for aspects.

The semantic fragile base class problem occurs when the semantics of the base class changes. In such situations, subclasses might become invalid. This problem occurs because the subclass has broken the superclass' encapsulation with the use of inheritance. Szyperski et al. (2002) have a thorough discussion of this problem.

The fragile base class problem makes reuse and evolution of software problematic, and is one of the reasons for introducing components.

Even with the introduction of components, the fragile base class problem can occur due to the use of aspects. Components usually expose an interface. To change or extend this interface in any way, ordinary subclassing is not enough. You have to make a new component. Aspects can change and alter this interface in the same way as subclasses in object-oriented programming, thus making the components fragile.

Examples:

> The Norwegian equivalent of a social security number is called a birth number. This is composed of 6 digits denoting the birthdate, and 5 digits that

are a person number. 02027212312 is the birth number for a person born on 2. February 1972 with person number 12312.

Consider an interface for a *Person* component with the following methods: *String getBirthNumber()* and *Date getBirthDate()*.

We could add an aspect for security that limits access to the birth number by creating an advice for the getBirthNumber() method. In a new version of the component the implementation of getBirthDate() might have changed to deduce the birth date from the birth number. If not carefully written, the aspect would then add security to the getBirthDate() method, which could break the application.

We could also consider an aspect for validating the birth date of a person, i.e., to make sure that the getBirthDate() method and the getBirthNumber() method is consistent regarding the birth date. We can create an advice for the getBirthDate() method that ensures that the birth date is the same as the date you can deduce from the birth number, and do the same for the getBirthNumber() method. If a new version of the component either use getBirthDate() to create a birth number, or use getBirthNumber() to return a birth date – then the aspect would lead to infinite recursion[6].

Finally, if the Person component did not have a getBirthDate() method to begin with, we could introduce it with an aspect. By using AOP we could introduce a new interface, BornPerson, on all classes that implement the Person interface. The BornPerson interface contains one method, getBirthDate(), which deduce the date from the birth number. If a new version of the component has a getBirthDate() method, the behaviour would be undefined and the application would break. This is an example of the syntactic fragile base class problem.

As aspects can extend the behaviour of components, they can also introduce the fragile base class problem to components. In an architecture such as QuA, where the platform decides which blueprint to instantiate by using service planning, this problem is important to be aware of. This means that a new component implementation is used quite often.

For the prototypes, this implies that aspects should try to rely on the contractual specification of the component only. They should also assume nothing about not contractually specified parts of the semantics of the implementations.

It is easy to quantify how much the aspects adheres to the contractual specification of components. How much the aspects assume about the semantics that are not contractually specified, is not, and has to be part of a qualitative analysis.

[6]One can argue that using a *!cflowbelow(Person)* pointcut designator in these examples would remove the problem as this would prevent the advice from triggering when for example the getBirthDate() method is called from within the component. However, this would remove the functionality added by aspects for an unknown amount of methods in the component. Such behaviour is not always wanted.

4.4.1.2 Orthogonal Aspects

Orthogonality is regarded as an advantageous property for software, and a property that promotes reuse. An intuitive understanding of orthogonal aspects would be that they are independent of other aspects or classes.

What exactly does independent or orthogonal mean, what should the aspects be orthogonal to? Other aspects, components, or maybe concerns?

Bergmans and Aksit (2001) interpret orthogonality as aspects that are orthogonal to other aspects. In their Composition Filters approach this means that filter specifications do not refer to the specification of other filters. This, along with the enhanced modularity achieved by Composition Filters, should lead to better reusability and adaptability of concerns.

Szyperski et al. (2002, p. 428) shares this understanding of orthogonality to some degree. Szyperski claims that *"No two aspects, as we know and understand them today, are truly orthogonal to each other"*. Szyperski uses encryption as an example of this. If one aspect encrypts messages, and another aspect traces method calls, the ordering of aspects is important as encrypted messages would lead to garbled log messages. Thus, the aspects are not independent and hence not orthogonal.

Claiming that no aspects are orthogonal on the basis of one observed example is a bold statement. Any *concern* that alters data (e.g., encryption or compression) need to be executed in a specific order. Thus, Szyperski's example is not a case that is special for aspects, it is general for concerns. Claiming that no two concerns are truly orthogonal is at best questionable.

Colyer et al. (2004) has another approach to orthogonality. Their aim is to find principles that help to create flexible, configurable aspect-oriented systems that scale to larger systems and larger concern implementations.

Colyer et al. understands orthogonal aspects as aspects whose inclusion is optional. This optional inclusion is related to substitutability where orthogonality requires that the behaviour of the program is unchanged if the aspect is removed or substituted with another aspect. This understanding is based on the discussion of subtypes and substitutability in object-oriented programming by Liskov (1987). Her definition of a subtype is known as the *Liskov substitution principle*:

> If for each object O_1 of type S there is an object O_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O_1 is substituted for O_2 then S is a subtype of T.

Colyer et al. (2004) define a *principle of orthogonal aspects* as a corollary to Liskov's substitution principle:

> Let $A(T)$ represent the behavior of [the type] T in the presence of aspect A.
>
> If for each object O_1 of type $A(T)$, there is an object O_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when O_1 is substituted for O_2 *and*
>
> If for each object O_1 of type T, there is an object O_2 of type $A(T)$ such that for all programs P defined in terms of $A(T)$, the behaviour of P is unchanged when O_1 is substituted for O_2,
> then A is an orthogonal aspect with respect to T.

This principle states that:

1. The behaviour of all programs P defined in terms of the type T should be unchanged when the aspect A is present in the system, i.e., it is safe to add the aspect A to the system from the perspective of T.
2. The behaviour of all programs P defined in terms of A(T) should be unchanged when the aspect A is not present in the system, i.e., it is safe to remove the aspect from the perspective of T.

The aspect A is restricted to introducing behavioural changes outside the specification of T. For the prototypes, this means that the audio stream should be able to stream audio and the components for calculating the value of π should be able to do this with or without the presence of the aspects.

This principle of orthogonality can in some cases be too strict. Thus, Colyer et al. introduce the *principle of weakly orthogonal aspects*:

> Let A(T) represent the behaviour of [the type] T in the presence of aspect A.
>
> If for each object O_1 of type A(T), there is an object O_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when O_1 is substituted for O_2,
>
> then aspect A is a weakly-orthogonal aspect with respect to T.

Weakly orthogonal aspects allow other types in the same concern as A to depend on the behaviour of A(T).

The principles of orthogonality provides good design guidelines for the prototypes, but they do not provide a method for quantitative measures of the orthogonality of a program. However, the principles of orthogonality provides valuable input to a qualitative evaluation of the prototypes.

4.4.1.3 Software Metrics

To determine if AOP reduces complexity, provides a better separation of concerns, or in other ways improve the quality of the code, qualitative measures are preferred. Evaluating the prototypes with qualitative methods should give exact feedback on the quality. Software metrics is a means of achieving qualitative measures.

Software metrics are a well know tool for evaluating a design or a software implementation. There is no general agreement of what "good design is", but there exists some design metrics that have evolved over time and that are generally regarded as sound.

Sommerville (1995) suggests the following metrics for evaluating a design:

Cohesion How closely are the parts of a component related?

Coupling How independent is a component?

Understandability How easy is it to understand the function of a component?

Adaptability How easy is it to change a component?

The principle of high cohesion and low coupling was proposed by researchers at IBM in the early 70's (Fenton and Pfleeger, 1997) and is a well known principle of good design.

Understandability and adaptability cannot be measured directly, but they are related to the *complexity* of a component. Sommerville suggests cyclomatic complexity as a measure for complexity. Cyclomatic complexity is a measure of the number of independent paths in a program:

$$CC(G) = \#\text{edges} - \#\text{nodes} + 1$$

Where $CC(G)$ is the cyclomatic complexity for a program flow graph. For programs without goto statements, this is equal to the number of conditions in the program.

Other metrics for complexity utilizes fan-in/fan-out structural measures. Fan-in is a measure of how many other components that call the component, and fan-out measures the number of other components called by the component.

Complexity metrics can also be used for measuring coupling. High fan-in values for a component might indicate high coupling, and high fan-out values might indicate that the complexity of the calling component is high.

Cohesion is not that simple to measure, as it requires an understanding of the component's purpose. Fenton and Pfleeger (1997) suggests the "lack of cohesion metric" (LCOM) for object-oriented programs. This measures how closely the local methods are related to local instance variables in a class.

Fenton and Pfleeger warns against using complexity metrics uncritically. There is a great appeal in generating a single number that express the complexity of a program. They argue that such a measure often will address conflicting goals, thus ending up as useless in practice. They also gives examples showing that some of the properties that a complexity metric must satisfy are contradictory.

Metrics such as coupling, fan-in/fan-out and cyclomatic complexity can be computed by tools. There exists numerous plugins for the Eclipse IDE that compute such values.

Metrics used by Others

Although coupling, cohesion and complexity are relevant for aspect oriented programs, the methods for measuring it is not that simple.

Consider a tracing aspect that pointcut every method call in the system. The weaved code for this aspect would give a very high fan-in value, thus indicating high complexity. The whole point with AOP is that this complexity is separated in a module by its own, thus measuring the weaved code is pointless. However, aspects might be coupled with other components in the system, so how can this be measured?

Kiczales et al. (1997) do not use complexity or coupling for measuring the success of applying aspects to a program. Instead, lines of code is used:

$$\text{reduction in code bloat due to tangling} = \frac{\text{tangled code size} - \text{component program size}}{\text{sum of aspect program sizes}}$$

Lines of code can give some indications of the complexity of a program, or how "bloated" it is. However, it is not a very good measure[7]. Lines of code is a measure of program

[7] Looking at the winner contributions to the last years "Obfuscated Perl Contest" gives a brief indication on why lines of code is not always a good measure for software complexity.

size. Fenton and Pfleeger (1997) suggest that lines of code is not the optimal measure of program size for object-oriented programs. Instead, counting the number of classes, methods and class interactions is a better metric.

The approach used by Kiczales et al. (1997) requires that you have two implementations that you can compare. One implemented without aspects, and one that use AOP. To create an additional implementation of a QoS-aware adaptive audio streaming service without AOP is a huge task that is beyond the scope of this thesis.

Papapetrou and Papadopoulos (2004) conducted a case study to measure the usefulness and usability of AOP. They implemented two versions of a web crawling system. One with AOP and one without. To evalute their study, they used metrics such as learning curve, stability and time used to create the implementations. They also used lines of code and comparison of code tangling as metrics.

To measure code tangling, they measured how many places code was added to add a concern. E.g., adding logging resulted in adding code 73 places without AOP, and only one place by using AOP.

A similar approach is followed in the study by Coady and Kiczales (2003). Their study evaluated the use of AOP in the BSD operating system. They measured how many places code for handling concerns such as page daemon wakeup and disk quota exists, then they measured how they could reduce this number by using AOP.

Zhang and Jacobsen (2003) use classical software metrics such as cyclomatic complexity, lines of code, number of methods pr class and coupling. In addition, they use scattering as a metric. Scattering is measured the same way as code tangling in the studies mentioned above.

Their measure of cyclomatic complexity ignores the complexity in their aspects. To gather values for cyclomatic complexity they have used the JavaNCSS[8] tool. This tool measures cyclomatic complexity simply by counting the number of *if*, *for*, *while*, *case* and *catch* statements in the Java code.

The study of Zhang and Jacobsen (2003) looks at how aspect oriented refactoring can improve CORBA implementations. They refactored the ORBacus ORB by using AspectJ to see if they could reduce complexity and increase modularity in the ORB. However, their structural metrics only take the Java code in account, the aspect code and the coupling between aspects and Java code is not measured.

An aspect oriented refactoring might create more modular, or less scattered, code. It might also reduce the complexity in the Java code, but if you do not take the complexity imposed by the aspects in account, you do not really know if the total system complexity has changed.

New Metrics for AOP

The problems with using classical software metrics for aspect oriented programs suggests that we need new metrics for AOP. This is also suggested by Coady and Kiczales (2003) who states that the area of AOP and metrics needs further research.

AOP is not the only field that suffers from this. This problem is evident in all systems that extracts handling of concerns out of the code. COMQUAD, QuO, EJB and Spring are all examples of systems that modularize concerns in external configuration files, deployment descriptors or domain specific languages. Classical software metrics

[8]http://kclee.com/clemens/java/javancss/

ignores the coupling between the code and the external files, and also the complexity within those external files. Complexity does not necessarily vanish if you move it from code to external configuration files.

Table 4.1 Metrics used by Garcia et al. (2005).

Attributes	Metrics	Definitions
Separation of Concerns	Concern Diffusion over Components (CDC)	Counts the number of classes and aspects whose main purpose is to contribute to the implementation of a concern and the number of other classes and aspects that access them.
	Concern Diffusion over Operations (CDO)	Counts the number of methods and advices whose main purpose is to contribute to the implementation of a concern and the number of other methods and advices that access them.
	Concern Diffusions over LOC (CDLOC)	Counts the number of transition points for each concern through the lines of code. Transition points are points in the code where there is a "concern switch".
Coupling	Coupling Between Components (CBC)	Counts the number of other classes and aspects to which a class or an aspect is coupled.
	Depth Inheritance Tree (DIT)	Counts how far down in the inheritance hierarchy a class or aspect is declared.
Cohesion	Lack of Cohesion in Operations (LCOO)	Measures the lack of cohesion of a class or an aspect in terms of the amount of method and advice pairs that do not access the same instance variable.
Size	Lines of Code (LOC)	Counts the lines of code.
	Number of Attributes (NOA)	Counts the number of attributes of each class or aspect.
	Weighted Operations per Component (WOC)	Counts the number of methods and advices of each class or aspect and the number of its parameters.

Garcia et al. (2005) have tried to refine the classical metrics to make them more suitable for AOP. Table 4.1 shows their refined metrics.

Those metrics were used to replicate the study of Hannemann and Kiczales (2002) in order to get a quantitative assessment of AOP implementations of the 23 GoF patterns.

Their metrics are an improvement over the earlier metrics presented here. Their metrics for coupling, cohesion and size takes the aspect oriented nature in account.

However, the metric for coupling would give very high values for some typical use of AOP. E.g., a tracing aspect would be coupled with every other class in the system.

The metrics for separation of concerns is a good start at developing such metrics. As they are new metrics, there does not exist much empirical data to evaluate whether a value for one of the metrics is good or bad. At present time they are mostly suited to compare how separation of concerns improves with different implementations of the same problem.

Lopes and Bajracharya (2005) takes a different approach when evaluating modularity in aspect oriented design. They use the design structure matrix (DSM)[9] and net options value (NOV) from the theory of modular design in their analysis of modularity. DSM is used as a modeling tool, and NOV provide a model for quantitative measures.

DSM captures more of the complexity than classical software metrics as DSM takes the dependencies of all parts of the system in account, i.e., dependencies between code, external libraries, design rules, configuration and deployment descriptors, etc.

DSM and NOV are promising tools for evaluating design decisions and evolution. E.g., for comparing a design without aspects with a design that uses aspects. NOV analysis calculates the "value" of the system based on a number of parameters. These parameters are based on assumptions and lack formal or empirical verification. Standard techniques to estimate parameters for NOV and richer models for NOV for software are issues of further research (Lopes and Bajracharya, 2005).

Thus, DSM and NOV are not suitable for evaluating the prototypes in this thesis. Getting a NOV value of 42 would not say anything meaningful about the system without having another NOV value to compare it with.

4.4.1.4 Summary

The goal of the experiments is to see if it is possible to separate some of the most relevant concerns for QoS by using AOP. Separating concerns at the code level only is an achievement, but if this separation introduce coupling and complexity that is not directly reflected in the code, it will be a drawback.

A problem with the experiments is that there are no existing data to compare with, i.e., there does not exist a QoS-aware application for QuA that adapts to changes in the available resources that do not use AOP.

The lack of data to compare with makes it problematic to use quantitative methods for measuring the success of the experiment. E.g., the metrics for code tangling and some of the metrics used by Garcia et al. (2005) is interesting but of no use if there is no reference implementation available for comparison.

It exists recommended numbers for the classical software metrics. E.g., SEI CMU gives recommendations for cyclomatic complexity[10]. Classical software metrics are of little use for AOP. Thus, new metrics are being developed. Those metrics are still very new, and it does not exist enough empirical data to give recommended threshold values for those metrics.

The metrics and tools available for quantitative analysis that exists today are not sufficient. Instead, a qualitative anlysis of the prototypes should be used.

[9] See http://dsmweb.org/ for more information about DSM.

[10] See http://www.sei.cmu.edu/str/descriptions/cyclomatic.html

The prototypes should be evaluated based on:

Encapsulation This is related to how much the aspects need to know about the components, i.e., the components' shade of gray. The darker shade, the better.

Modularity How well are the concerns separated, are the aspects likely to introduce fragile base class problems, and are the modules orthogonal?

Reusability This is related to encapsulation and modularity. It is also related to how much the aspects need to known about the components and the service they run in – e.g., do they assume a specific service composition?

Thus, the main success criteria for the prototypes are whether they manage to separate QoS concerns, and the degree of reusability.

Chapter 5

Experiments

This chapter describes the experiments performed in order to get empirical data that can be used to validate the analysis in the previous chapter. The experiments consist of prototype implementations. Two prototypes are implemented. One prototype investigates static QoS in simple compositions, and the other investigates dynamic QoS in complex compositions.

5.1 Simple Compositions: Calculating Pi

This experiment looks at simple compositions, i.e., compositions containing only one component. The main goal of the experiment is to separate the QoS concern from the component implementation when the service planner chooses which blueprint to instantiate.

See the analysis in section 4.2 on page 51 for a rationale for the experiment. The analysis also suggests that resource reservation and transactional access to the resource manager can be separated using aspects.

5.1.1 Separating Static QoS

Static QoS is directly supported in QuA. To support QoS, blueprints must provide the *QoSAware* type. In the Java implementation, this means implementing the *QoSAware* interface and adding the QoSAware type to the blueprint's descriptor.

The QoSAware interface contains one method that takes a *QoSStatement* as input, and produce a *ResourceStatement* as output as shown in listing 5.1.1.

Listing 5.1.1 The QoSAware Java interface.

```
public interface QoSAware {
    public ResourceStatement requiredResources(QoSStatement qosStatement)
        throws QuAException;
}
```

This is used by the QuA service planner to choose which blueprint to instantiate in a composition[1].

5.1.1.1 Components for Calculation of Pi

As an example, three components to calculate the value of π is created. The first component uses the Chudnovsky method with binary splitting (Haible and Papanikolaou, 1998), the second method uses the Borwein 4th-order convergent algorithm (Bailey et al., 1997), and the third method uses Machin's algorithm. These methods are chosen because they have different characteristics.

The Chudnovsky method:

$$\frac{1}{\pi} = 12 \sum_{n=0}^{\infty} \frac{(-1)^n (6n)! (13591409 + 545140134n)}{(3n)!(n!)^3 \, 640320^{3n+3/2}}$$

The Chudnovsky method adds about 14 digits of accuracy for each iteration.

The Borwein method:

$$y_0 = \sqrt{2} - 1, \qquad \alpha_0 = 6 - 4\sqrt{2}$$

$$y_{n+1} = \frac{1 - (1 - y_n^4)^{\frac{1}{4}}}{1 + (1 - y_n^4)^{\frac{1}{4}}}$$

$$\alpha_{n+1} = \left[(1 + y_{n+1})^4 \alpha_n \right] - 2^{2n+3} y_{n+1} (1 + y_{n+1} + y_{n+1}^2)$$

$$\text{then } \frac{1}{\alpha_n} \to \pi \text{ as } n \to \infty$$

The Borwein method approximately increases the number of correct digits by a factor of four for each iteration.

The Machin method:

$$\frac{\pi}{4} = 4 \tan^{-1}(\frac{1}{5}) - \tan^{-1}(\frac{1}{239})$$

Then using a power series for $\tan^{-1}$ to compute π.

Component Implementations

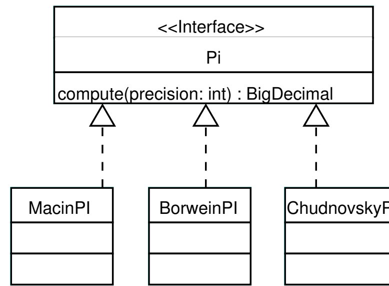
To calculate the value of π with high precision, high precision math is needed. Apfloat[2] is used as a high precision math library. The Chudnovsky algorithm is already implemented in the Apfloat library as an example of library usage. Changing the example code to become a QuA component is trivial, and the implementation of the Machin and Borwein algorithms is straight-forward.

The implementations are shown in figure 5.1. The components are packaged in three separate archives, or to be more precise: They are 3 separate *blueprints* that are bundled together in one persistent *repository*.

[1] The QoSAware interface is changed to handle 1-m relations between QoS statements and resource statements. This is described in section 5.2.1.2 on page 91.

[2] The Apfloat package is available from http://www.apfloat.org/.

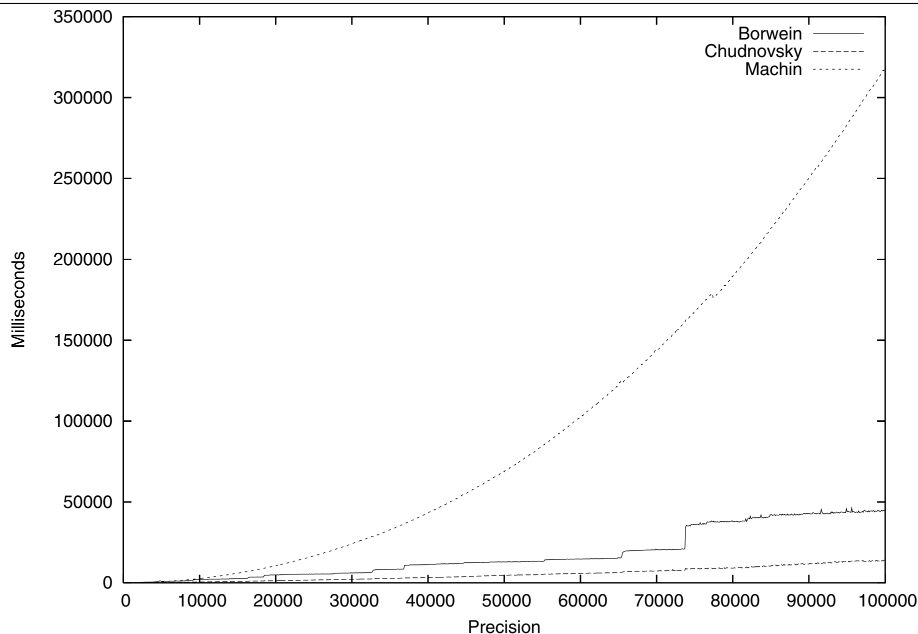
Figure 5.1 Classdiagram showing π components.



Implementation Performance

In order to add QoS awareness to the components, we need to know how they perform and what resources they require. Figure 5.2 shows the performance of the different component implementations up to a precision of 100.000. The tests are run on a 1.3Ghz Duron processor using Java 1.4.2 on Windows 2000. Figure 5.3 shows the same data, but only for precision up to 10.000.

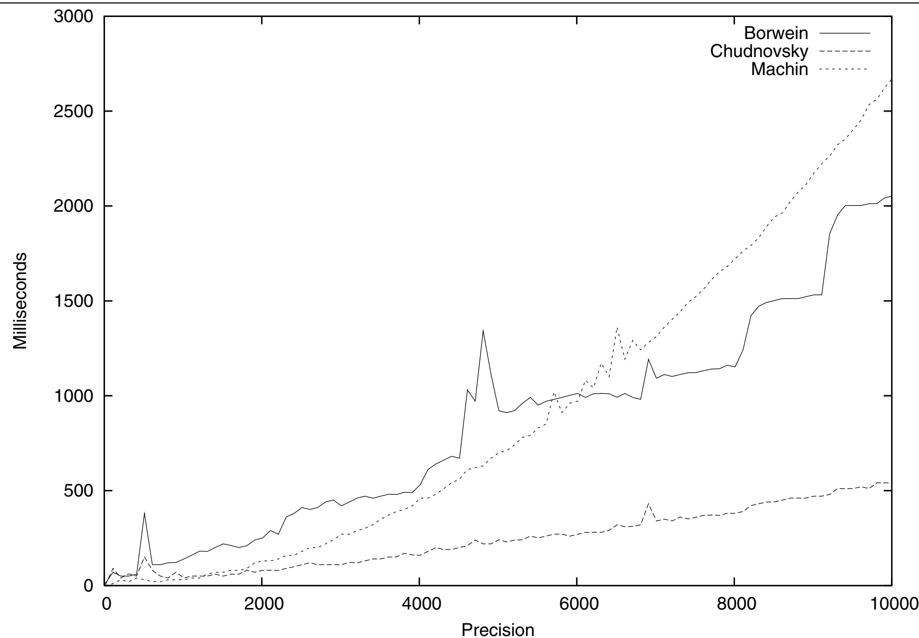
Figure 5.2 Performance for π algorithms.



It is a little disappointing to see the performance numbers. They show that the Chudnovsky component is always fastest and that the Machin component is faster than the Borwein component up to a precision of ca. 5700 decimals.

The algorithms chosen have very different characteristics, and the hope was that one algorithm would not always be best. If the Chudnovsky component had been a little slower, the performance line would cross the steps in the Borwein performance line a number of times.

Figure 5.3 Performance for π algorithms for smaller precision.



Implementation Resource Usage

A QoS aware QuA component must know how much resources it requires to fulfill a quality specification. The main resource used by the π components is CPU. Mathematical operations, such as computing the value of π, use all the CPU power they can get. This can be confirmed by watching the windows task manager while the components are running. The task manager shows 100% CPU usage.

Thus, quantifying the requirement for CPU usage for these components is not trivial. Investigating how to quantify CPU usage properly in this case is beyond the scope for the experiment. Thus, bogus values are chosen for each component.

The Apfloat library used for high-precision math uses the file system when the numbers get very big. Thus, there might be some differences in usage of the disk resource. Disk as a resource is ignored in this experiment.

5.1.1.2 Making the Components QoS Aware

To make the components QoS aware, the *QoSAware* interface must be implemented, i.e., a mapping between the wanted QoS and the resources required to provide that QoS must be created. In this case, a fixed bogus value are used to denote the CPU resource requirement no matter which QoS is required. These requirements are implemented in separate classes that extend the *QoSAware* interface. The implementations are not a part of the component blueprints. To make the component blueprints QoS aware, we must:

- Add the *QoSAware* type to the list of types provided by the component blueprints, i.e., add it to the meta-information for the blueprint.
- Make the component main-class implement the *QoSAware* interface.

Adding the QoSAware Type to the Blueprint

The types provided by a blueprint are described in the blueprint metafile. Adding the type to the metafile would be simple, but doing that without actually providing the type would create an invalid blueprint.

To cleanly separate the QoSAware type, the ideal solution is to let the blueprint only contain the implementation of the *Pi* type. Thus, providing the QoSAware type during blueprint load time is sufficient.

QuA blueprints are loaded into a capsule with a dedicated classloader invoked from the QuA core. The idea is to intercept the blueprint loading process, and then alter the blueprint meta-information.

This is achieved with an *around advice* in AspectWerkz. The advice looks like this in the aspect description file:

```
<advice-def name="MakeQoSAwareAdvice"
            class="qua.aspects.MakeQoSAware"
            deployment-model="perJVM">
  <!-- comma separated list of short names for components
       that should implement QoSAware -->
  <param name="shortNames" value="MachinPi,BorweinPi,ChudnovskyPi"/>
</advice-def>
```

This advice is later bound to a pointcut in an aspect:

```
<pointcut-def name="ConstructComponent"
          type="method"
          pattern="void
            qua.core.componentmanager.ComponentBlueprint.
             initComponent(..)"/>

<bind-advice pointcut="ConstructComponent">
    <advice-ref name="MakeQoSAwareAdvice"/>
</bind-advice>
```

Each QuA blueprint has a shortname defined in the meta-information for the blueprint. The advice definition takes a list of shortnames as parameters. The advice implementation is executed each time the *initComponent* method of *ComponentBlueprint* is called. Then it checks if the shortname matches one of the given shortnames it should make QoS aware. If it matches, the QoSAware type is added to the blueprint meta-information.

Thus, this is a reusable aspect. Adding the *QoSAware* type to a blueprint is a matter of adding the shortname to the advice parameter.

Adding the QoSAware Interface Implementation

The next step is to add an implementation of the *QoSAware* interface to the blueprint's main class. This is done by using an introduction:

```
<!-- The mixin implementation for a QoSAware MachinPI -->
<introduction-def name="QoSAwareMachinPI"
            interface="qua.types.QoSAware"
            implementation="qua.math.aspects.MachinQoSAware"
            deployment-model="perJVM"/>
```

The introduction definition is later bound to the main class in an aspect:

```
<!-- Add the QoSAware interface to the MachinPI component -->
<bind-introduction class="qua.math.MachinPI">
    <introduction-ref name="QoSAwareMachinPI"/>
</bind-introduction>
```

What is meant by binding introductions and advice in an aspect, is to wrap them all in an aspect tag in the aspect definition file:

```
<aspect name="MakeQoSAware">
    <!-- Advice binding -->
    <!-- Introduction binding -->
</aspect>
```

When a test program is started with these aspects enabled, QoS aware versions of the π components become available.

Making the Client-side QoS Aware

Separating QoS awareness in the component blueprints is not enough to separate the QoS concern from an application. To enable QoS, the users of the components need to create a quality specification. This is done by implementing the *QualitySpec* interface in the QuA platform and provide that implementation when composing services. The QualitySpec contains the *utility function*, which describes the QoS boundaries and the utility of a given QoS statement.

We may want to separate this quality specification from the application code, or we may want to enable QoS awareness in an existing application with no notion of QoS. It is already showed how the components can be made QoS aware with the use of AOP, but in order to make use of this QoS awareness, the application must provide a quality specification when the service is composed.

To compose a service, one of the *compose()* methods in the QuA core is used. Intercepting access to the compose methods in order to provide a quality specification is a straight-forward task.

The problem is to map an external quality specification to a given service. A QuA service, or service specification, has no external visible identity. This means that it is not possible to map an external quality specification to a service created in the application.

If we know that the application creates only one service, it is no problem. If this is the case, we can add an external quality specification to any service composed in QuA. However, this is a knowledge we seldom have, and that we can not rely on. Utilizing such knowledge would break the encapsulation in the application and make it fragile.

Alternative Implementation

The implementation described above, used AspectWerkz version 0.8.1. An attempt to reuse the implementation was made in the audio experiment described in section 5.2. At this time, AspectWerkz version 2.0 was released, with significant differences and improvements. The aspects were ported to AspectWerkz 2.0, but significant bugs were found in the AspectWerkz implementation. Those bugs made it impossible to use AspectWerkz as an AOP framework in the experiment.

After the merger of AspectWerkz and AspectJ, AspectWerkz is no longer maintained, which implies that the bugs will not be fixed. Thus, the aspects was ported to use an early access release of AspectJ 5.

This implementation is different from the AspectWerkz implementation. The aspect that adds *QoSAware* to the blueprint description uses the same principles, but a mixin is no longer used to add the QoSAware implementation to the component main-class. Instead, the ASM (Bruneton et al., 2002b)[3] byte-code manipulation tool was used to introduce the QoSAware interface to the component main-class.

This is possible because QuA uses its own classloader. An aspect intercepts the loading of the component-main class and uses the ASM tool to alter the byte-code before it is loaded. This modification adds "*implements qua.types.QoSAware*" to the signature of the class. It also adds all the methods defined in the interface, and delegates the method implementations to a another class. Which class to delegate to is configurable.

The configuration is read from the QuA core, i.e., the properties used to instantiate a QuA capsule is also used to determine which components should be made QoSAware, and what class implements the QoSAware interface.

Example:

QoSAware/BorweinPi=qua.qos.QoSAwareBorweinPI

Means that the component with short-name *BorweinPI* should be made QoSAware, and that the class *qua.qos.QoSAwareBorweinPI* contains the implementation of the QoSAware interface. An advantage with using byte-code modification in the classloader instead of the AOP mixin mechanism to introduce the QoSAware interface, is that we can add other aspects to QoSAware components. I.e., we can use QoSAware as a pointcut.

5.1.2 Resource Management

The analysis (section 4.2.2.1 on page 53) suggested that resource reservation can be separated with aspects, and that access to the resource manager can be made transactional.

Implementing transactional access to the resource manager was straight-forward, and is similar to the code used as example in the analysis. Separating resource reservation is also similar to the code used as example in the analysis, but some problems occured.

The example code in the analysis assumes that the instantiated components are available after a service is planned. Those components are then examined for their resource requirements, and the resources are reserved. The problem is that the component instances are *not* available after service planning.

Access to the component instances is fundamental. Not only for resource reservation, but also for adaptation. It is impossible to adapt a service if you do not know which objects are part of the service.

Thus, the service planner implementation is changed to save the instantiated components in the service context. Each service is supposed to run in its own service context. Thus, this is a natural place to store the instantiated components. However, this is an ad-hoc change to the QuA prototype, and a more though through solution should be incorporated in the QuA architecture.

[3] See http://asm.objectweb.org/ for more recent information about ASM.

With the component instances available, separation of resource reservation is implemented as suggested in the analysis.

5.1.3 Summary

The experiments shows that QoS awareness can be separated from the component implementations, and that resource reservation can be separated as an aspect.

Separating QoS awareness from the component implementations requires some detailed knowledge about the QuA core, but no information about the component implementations. This implies that the aspects can be reused for any component, provided that the necessary parts of the QuA core does not change.

For the various roles in QuA, this implies that the *component developer* can focus on implementing components – e.g., π components – and that a *QoS expert* can add QoS capabilities to the component (i.e., an implementation of the QoSAware interface) without having to change the component implementation.

Separating resource reservation and transactional access to the resource manager requires no knowledge about the intrinsics of the QuA core – only knowledge of the contractual specification of the resource manager and the service planner. An ad-hoc solution had to be created in order to get access to the component instances used in a service. This indicates a flaw in the QuA architecture more than a need for special implementation knowledge. Thus, the aspects for resource reservation should be reusable with any service planner or resource manager that follows the contractual specification for service planners or resource managers.

For the various roles in QuA, this implies that resource management is of no concern for neither the *platform developer* (i.e., the developer who implements the service planner), the *component developer* nor the *application developer*. The role responsible for enabling the aspects is the *deployer*. The deployer is responsible for configuring the middleware to suit the application's needs. Thus, he must enable the aspects needed for resource management and static QoS. The aspects are enabled by adding some parameters to the java VM that runs the QuA capsule. Adding those parameters are the responsibility of the deployer.

The use of π as an example was useful in the analysis. The use of π led to a discussion of monitoring of discrete services and showed problems with monitoring and adaptation in such services. The use of π was not useful in the prototype implementations. Instead, simple components that returns "Hello World" with different QoS and resource requirements would have been sufficient.

5.2 Complex Compositions: A Distributed Streaming Audio Player

This experiment looks at more complex compositions, i.e., distributed compositions containing many components. The experiment implements an application for streaming audio data in real-time using UDP. The main goal of the experiment is to see whether the composition can be adapted during run-time. Both parameter- and compositional adaptation is investigated. The experiment also looks at some simple QoS monitoring.

See the analysis in section 4.3 on page 59 for the rationale for the experiment.

5.2.1 Implementing an Audio Service

The initial audio service used as a basis for the experiment with aspects, was implemented in cooperation with Øyvind Matheson Wergeland. Wergeland needed a proof-of-concept implementation for his master's thesis on the QuA Java prototype (Wergeland, 2005), and I needed it to experiment with aspects.

The audio service tries to resemble the service discussed in the analysis; an audio source streams audio to an audio encoder, which sends data to an audio decoder by using an UDP binding, which streams audio to a speaker. In the analysis, a microphone is used as audio source. Using a microphone as audio source is cumbersome to experiment with. Instead, this prototype uses a wave file as audio source. A delay is created in the audio file provider to simulate real-time production of audio data.

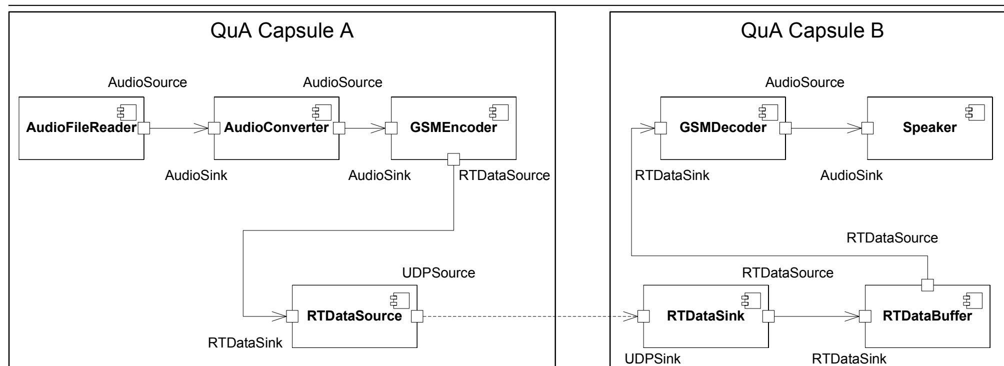
To implement the components, the Java Sound API is used. Additional functionality for converting sample rate, sample size, etc uses Tritonus[4], an alternative Java Sound API implementation. The initial implementation of the audio service was not QoS aware. The initial implementation was created with the sole purpose of implementing an audio service with the QuA Java prototype. Support for monitoring or adaptation was not a concern in the initial implementation – the code had to be refactored later to add support for those concerns.

5.2.1.1 Audio Codecs

To encode audio, audio codecs for speech were chosen. There are not many freely available Java implementations for speech codecs. E.g., codec implementations for G.723 or G.729 are only available commercially.

We found two freely available codecs: GSM and Speex. The GSM codec is available as a part of the Tritonus distribution, and Speex is available from http://www.speex.org/. In addition, Tritonus supports a-law and μ-law encoding. This makes it possible to create a G.711 codec implementation. Only GSM and Speex are used in the prototype.

Figure 5.4 The service composition for streaming GSM.



The GSM codec imposes some restrictions to the audio stream it receives; it only accepts 16 bit PCM signed mono little endian data with a sample rate of 8kHz. In order to make sure that it receives acceptable data, a converter component is created and placed

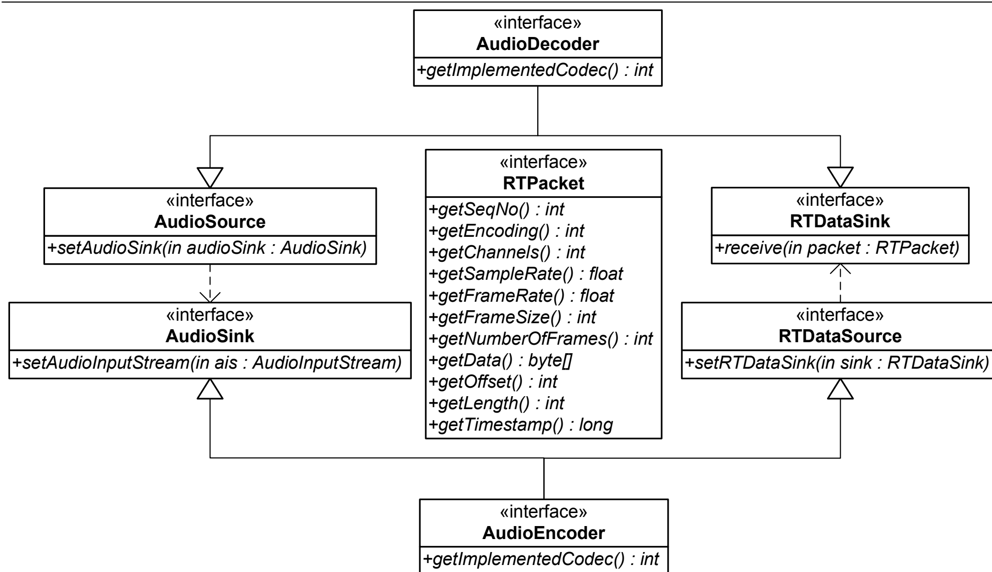
[4]See http://tritonus.org/ for more information about Tritonus.

in front of the GSM encoder in the service composition. The resulting composition can be seen in figure 5.4.

The figure is similar to figure 4.1 in the analysis chapter (section 4.3 on page 59). The figure shows that the service composition spans two QuA capsules using UDP to send real-time data between them. The QuA capsules can run on different hosts.

Figure 5.5 shows the most important interfaces provided and required by the components in the service. Note that the *AudioInputStream* used in the *AudioSink* interface is part of the Java Sound API.

Figure 5.5 UML class diagram showing the most important interfaces.



The GSM codec only supports one configuration: 8 bit sample size, 8kHz sample rate and one channel only. The Speex codec supports multiple configurations; 8, 16 and 32 kHz sample rate and one or two channels, i.e., mono or stereo sound. In addition, it supports features such as variable bit rate (VBR), average bit rate (ABR), and voice activity detection (VAD)[5].

A service using the Speex codec is similar to the service using GSM. Speex also needs a converter component. If the Speex encoder should use 32kHz sample rate and two channels, the audio input has to use the same configuration.

5.2.1.2 Extending the QoSAware Type

The features of the Speex codec adds some possibilities to a Speex component; one component can support different configurations. This requires changes to the QoSAware type. The QoSAware type is able to answer how much resources is required to support a QoS statement. With multiple component configurations available, a QoS statement might map to multiple resource requirements – one requirement for each component configuration that fulfills the QoS statement. Support for multiple configurations of

[5]Those features are described in the analysis chapter, section 4.3.1.1 on page 60

components can also be seen in middleware such as COMQUAD (Göbel et al., 2004b)[6]. In COMQUAD, a component can have multiple profiles, which correspond to multiple configurations.

Figure 5.6 The enhanced QoSAware type.

QoSAware
+requiredResources(in qosStatement : QoSStatement) : Collection<ResourceStatement> +configure(in qosStatement : QoSStatement, in resourceStatement : ResourceStatement) +getConfiguredResources() : ResourceStatement +getConfiguredQoS() : QoSStatement

Figure 5.6 shows the enhanced QoSAware type. The *requiredResources* method is changed from returning a *ResourceStatement* to returning a collection of resource statements.

With multiple configurations possible, the component must be configured to use one of them. Thus, a configure method is added to the type. Other parts of the system might want to know the configuration of the component, thus, the *getConfiguredResources* and *getConfiguredQoS* methods are included. E.g., the aspect for resource reservation uses the *getConfiguredResources* method in order to decide which resources to reserve.

5.2.1.3 Component Implementations

This section describes the component implementations. Most of the details are left out, but some important details and concepts needed to understand how the service works are described.

Audio Components and the Java Sound API

The AudioInputStream object is essential in the Java sound API. An AudioInputStream is used as input to converters, encoders and the speaker (actually a mixer). To create an AudioInputStream, you either need a file containing audio data, another AudioInputStream, or a generic InputStream together with a description of the audio format in the stream.

In the audio service, packets with byte arrays are used to send audio samples between many of the components. To make an array of bytes readable from an InputStream, the PipedInputStream and PipedOutputStream objects provided by the java.io package can be used. The only alternative is to implement an InputStream taking byte arrays as input yourself.

A PipedOutputStream is connected to a PipedInputStream. Connected piped streams allows you to write a byte array to the output stream, and read the bytes from the input stream. Reading from the input stream and writing to the output stream has to occur in separate threads. If this happens in the same thread, the thread will be blocked. Thus, all components that takes a byte array of sound samples as input in order to convert or encode the sound samples, will need to create separate threads.

The audio encoders, i.e., the Speex and GSM encoders, read audio samples from an AudioInputStream and provides a new AudioInputStream containing the encoded

[6]See section 2.7.1 on page 25 for a description of COMQUAD.

samples. The encoders need audio input with the same format they use as output. E.g., CD samples use a sample rate of 44.1kHz and two channels, but GSM uses a sample rate of 8kHz and one channel. Thus, the GSM encoder needs an audio stream with 8kHz and one channel as input. The Speex encoder can use different configurations, e.g., 8kHz, 16kHZ or 32kHz sample rate and one or two channels – but it needs audio samples with a corresponding format to encode the samples.

To convert the samples to the format required by the encoders, the converters from the Tritonus distribution is used. Those converters can not convert from 44.1kHz stereo to 8kHz mono in one step, but requires separate steps for sample rate, number of channels, and other characteristics of the audio format. Thus, a separate component that identifies the necessary steps is created for converting audio samples.

There may be some cases where an audio converter is needed after decoding. After decoding, the audio format is the same format as used in the codec, e.g., 8kHz mono for GSM. Not all sound cards are capable of playing that format. In those situations, an audio converter is needed between the decoder and the speaker. All soundcards the prototype has been tested on are capable of playing all the output formats used by the codecs. Thus, no converter is added after the decoder.

The Real-Time Binding

The audio samples are sent over the network using UDP. A protocol inspired by RTP is used to packetize the samples. Figure 5.5 on page 91 shows the interface for the *RTPacket*. The packets sent over the network contains most of the fields described in that interface (some of the fields can be derived from others and are not sent over the wire).

The *RTDataSource* component responsible for sending the packets using UDP, receives single frames of audio from the encoders. E.g., the GSM encoder sends single frames, each containing 33 bytes. If each frame is sent immediatly by the RTDataSource component, we would waste a lot of bandwidth. The IP and UDP protocols add 28 bytes of overhead to each packet sent, 48 bytes if IPv6 is used. In addition, the packets created by the RTDataSource component adds 16 bytes of overhead. Thus, each packet sent adds 44 bytes of overhead, or 64 bytes of overhead if IPv6 is used. This does not count the overhead added by the lower layers, e.g., Ethernet or ATM.

Adding an overhead of 44 bytes to send 33 bytes of GSM data will use nearly 60% of the bandwidth for protocol overhead. Thus, the RTDataSource component is able to delay the received audio frames and send packets containing multiple frames over the network. The component can be configured with the maximum packet size and the delay to use. The maximum packet size should not be greater than the MTU of the network connection to prevent packet fragmentation. The delay is added to prevent starvation. E.g., if the last audio frame is received before the maximum packet size is reached, the packet will never be sent if a maximum delay is not used. When the maximum delay is reached, the packet is sent even if the packet has not reached its maximum size.

The AudioFileReader component used in the prototype tries to simulate a real-time audio source by adding a delay after each packet it sends. Without this delay, the packets would be sent very fast over the network, and thus filling up the buffer on the receiving side.

The buffer on the receiving side is also part of the real-time binding. The buffer smooths out bursts of data, and is able to use the sequence number in the packets to

reorder them if they are received out of order. The buffer can also be configured with a maximum size and delay. A separate thread reads packets from the buffer, and sends them to the decoder in a timely fashion, i.e., the data is pushed from the buffer, and not pulled from the decoder.

5.2.2 Creating a QoS Aware Player Application

To experiment with the audio service, a test application with a graphical user interface is needed. A GUI is needed to display output from QoS monitoring, and to change parameters for the service during run-time.

Figure 5.7 A screenshot of the test application.

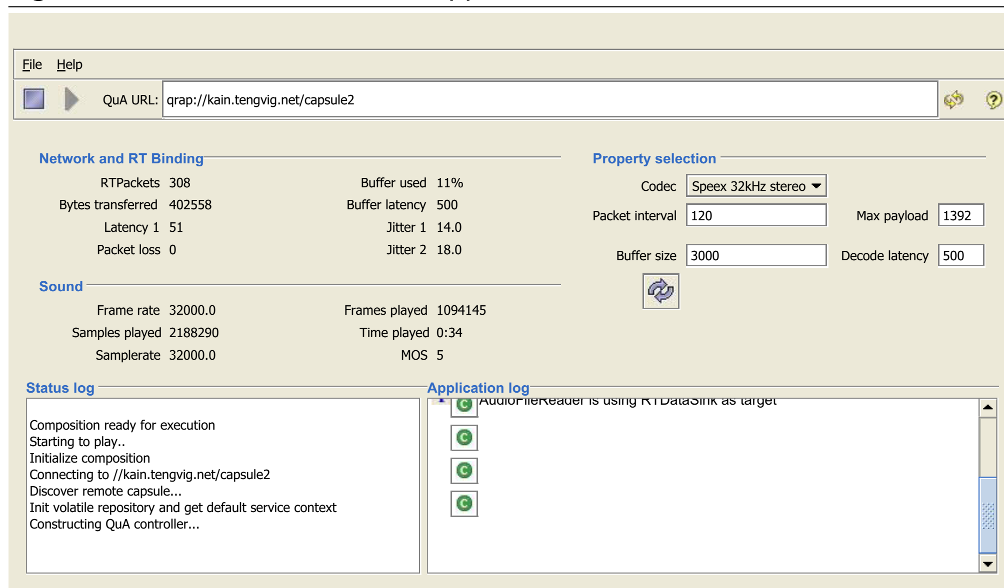


Figure 5.7 shows a screenshot of the audio player test application. The QuA URL at the top points to the remote capsule – the capsule acting as audio source (shown as “QuA Capsule A” in figure 5.4 on page 90). The path to the audio file acting as audio source is placed in a configuration file.

The left side of the GUI contains output from various monitors, and the right side contains properties that can be set for the service. Those properties can be changed during run-time.

There is no generic monitor component created for the QuA prototype. The test application uses its own ad-hoc monitor. This is a singleton object that monitoring aspects can report to. A thread reads the state of the monitor every 200ms in order to update the GUI.

5.2.2.1 Creating QoS Aware Components

To make the components in the composition QoS aware, the method used in the alternative implementation for the π components are reused (see section 5.1.1.2 on page 87). The implementation of the QoSAware interface is implemented separate from the

component implementations and is weaved in by an aspect.

There is no need to have accurate implementations of the QoSAware interface in this experiment, i.e, the components do not need to report accurate measures of their resource demands. The only reason for making the components QoS aware is to experiment with them. The following mapping between QoS requirements and resource demands are implemented[7]:

AudioFileReader This component needs memory. This is hard-coded to use 5kB of memory no matter the QoS requirements.

GSMEncoder This component always needs 2kB memory, 7% CPU and 13.2kbps bandwidth. If the component is not able to fulfill its QoS requirements (e.g., a high value for MOS or sample rate is specified), it requires an impossible amount of resources to prevent being selected.

GSMDecoder This component always needs 2kB memory and 5% CPU. The same predicate as used above for the encoder determines whether the component is able to fulfill the QoS requirements.

SpeexEncoder This component always uses 3.5kB memory and 25% CPU. Bandwidth is calculated based on MOS and number of channels.

SpeexDecoder This component always uses 3kB memory and 10% CPU.

RTPBindingSender This component always uses 1% CPU. Memory and bandwith usage is a function of delay. The sender component adds 48 bytes of overhead to each packet sent. A decrease in delay requires sending of more packets, which requires more bandwith.

RTPBindingReceiver This component always requires 2% CPU no matter the QoS requirements.

RTDataBuffer This component requires memory as a function of delay and jitter, i.e., the size of the buffer depends on the delay it introduces.

This mapping is neither accurate nor complete. E.g., many of the components require separate threads. This is not stated as a resource requirement. However, this is sufficient to select and configure components based on QoS requirements and available resources.

The generic implementation planner – i.e., the implementation planner used for planning QoS aware services – does not support remote repositories[8]. This implies that experiments requiring QoS aware services can not be distributed on different capsules. However, it should be possible to experiment with monitoring and adaptation of distributed services without requiring a QoS aware service.

[7]The components described here do not match the component composition in figure 5.4. Instead, the component composition shown in figure 5.8 on page 97 is used. The rationale for changing the service is described in section 5.2.4 on page 97

[8]See section 3.1.2 on page 37 for a description of service planning with the QuA Java prototype and for the distinction between the generic and the basic implementation planner.

5.2.3 QoS Monitoring

The analysis suggested that separating monitoring of QoS as aspects is a straight-forward task (section 4.3.3 on page 63). Monitoring of most of the QoS dimensions discussed in the analysis are implemented in the following aspects:

DecoderMonitor This aspect intercepts the *receive* method in all AudioDecoder components. It inspects the RTPackets received in order to determine *packet loss*, *MOS* and *sample rate*.

RTMonitor This aspect intercepts the buffer component. It intercepts the *receive* method in order to monitor *latency* and *jitter*. The jitter monitored here is the jitter until the packet reaches the buffer, it is not the experienced jitter for the user of the service. Thus, the smoothing of jitter performed by the buffer is not reflected here. This measure of jitter is shown as "*Jitter 1*" in figure 5.7.

The latency monitored here is the latency from the first audio sample was read by the AudioFileReader component until it reaches the buffer. I.e., latency added by the audio encoder (a GSM packet contains 20ms of sound), the grouping of audio frames in the RTDataSource component, and latency in the network. This is shown as "*Latency 1*" in figure 5.7.

Jitter is calculated as described in the RTP (RFC 3550) specification. This aspect also monitors the number of bytes received.

RTBufferMonitor This aspect intercepts the buffer component in order to monitor the *latency* in the buffer. This is based on how long time a packet spends in the buffer until it is sent to an RTDataSink. This aspect also monitors the jitter based on when the packets leave the buffer – i.e., the smoothing of jitter in the buffer is accounted for. This measure of jitter is shown as "*Jitter 2*" in figure 5.7.

LineListener This aspect intercepts all calls to the Java sound APIs SourceDataLine object, i.e., writing of data to the speaker. The aspects monitors sample size, sample rate, and other characteristics of the audio format. This is a highly inefficient aspect, as a separate method call is used for every audio frame. For GSM, this method is called 50 times pr second.

All the relevant information for monitoring QoS is monitored in the aspects described above. This aspect is mainly used for debugging. It is described here because it is responsible for the remaining fields shown in the test application.

All the aspects operate on the component specifications, i.e., the Java interfaces the components must implement. This implies that the aspects are not special for a given component implementation, but is valid for all components that implement the required types. All monitoring aspects report to the QoS monitor in the test application, no generic monitor component is created.

Only the latency from the first sample is sent to the encoder until it reaches the decoder is monitored. The time spent in the decoding process is not monitored. This latency is measured in the prototype implementation, and is found to be neglible.

5.2.4 Adaptation

The analysis (section 4.3.5 on page 66) has a simple idea for adaptation: Use aspects to add a layer of indirection and redirect calls to adapted components in this layer. There are some requirements on the service to make it adaptable:

1. The original service and the adapted service must be isomorph – i.e., the same service specification can be used for both services.
2. All dependencies between components must be stated in the service specification.

The second requirement is not satisfied in the audio service shown in figure 5.4 (page 90). The encoder components require a converter component to receive audio data with correct sample rate. The analysis suggests solving such dependencies by creating composite components, i.e., a composite encoder component containing both the converter component and the encoder component.

Another requirement, not mentioned in the analysis, is that all the components and bindings in the service must be QoS aware. This is not the case for the service in figure 5.4. The *AudioInputStream* class from the Java sound API is used as a binding between many components (all *AudioSource/AudioSink* bindings use *AudioInputStream* as a binding).

The method for adapting components suggested in the analysis, adapts a component when a method is called on the component. If AudioInputStream is used as a binding, no methods are called on the receiving end (e.g., an encoder) – data is written to the stream by the sender, and read from the stream by the receiver. This makes it impossible to adapt the receiver with the method suggested in the analysis. With a QoS aware AudioInputStream, we could adapt the binding in order to adapt the receiver, but the AudioInputStream is not QoS aware.

Figure 5.8 The improved audio service.

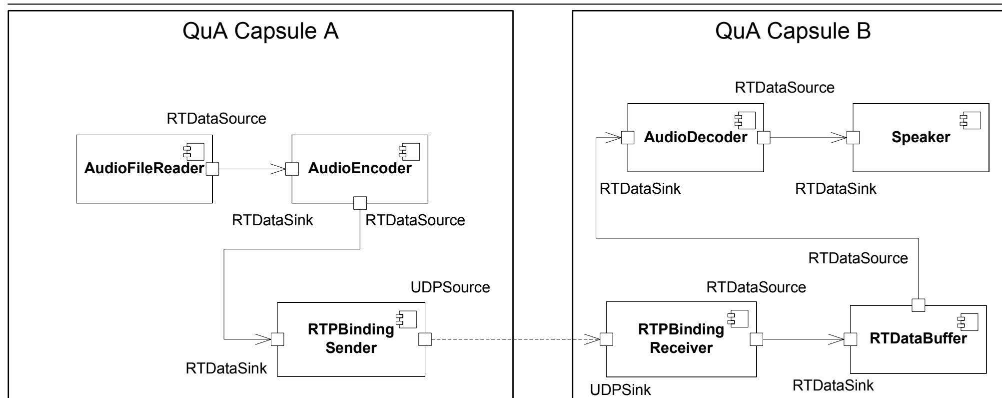


Figure 5.8 shows an improved audio service. The dependency between the converter component and the encoder components is removed. The QuA Java prototype does not support composite components, instead, the converter is made a part of the encoder components.

The use of AudioInputStream as a binding between components is removed. Instead, all components use the *RTDataSink/RTDataSource* binding.

5.2.4.1 Architectural Reflection

To adapt a component, or a service, as suggested in the analysis, the adaptation mechanism needs knowledge about the service composition. We need to know the following:

- The service specification. This includes the binding specification and the quality specification.
- Which components are part of the service, i.e., which component blueprint instances are used.
- Which service context are used. The service context provides access to the resource manager and to the planners needed to instantiate QoS aware components.

All this information is available when a service is planned, but it is not kept anywhere after service planning is finished. Aspects can collect this information and make it available to the adaptation mechanism. The analysis (section 4.3.5.2 on page 68) suggests using the AOP mixin mechanism to add a reflective interface to QuA components that provides access to this information.

There is one problem with using the mixin approach; QuA components does not have a place where one could add a pointcut. From an AOP perspective, a QuA component looks like any other Java object.

The CORBA binding for Java uses *marker interfaces* to solve a similar problem. IDL generated interfaces implements an empty interface, *IDLEntity*, whose only purpose is to inform the CORBA marshaller that a corresponding helper class exists.

If QuA components had to implement an interface, we could create pointcuts on that interface. Forcing components to implement an interface is not an unusual approach. E.g., Fractal (Bruneton et al., 2002a) components must implement the *ComponentIdentity* interface.

To avoid forcing components to implement an interface, we could use the Java dynamic proxy mechanism. QuA components are instantiated by the *QuAComponent* factory object. This object could add dynamic proxies to all the instantiated components. There are a few limitations to dynamic proxies. Dynamic proxies do not alter the behavior of the object under consideration, they only add a proxy to the object. As dynamic proxies are created during run-time, they can not be used as a basis for a pointcut.

To avoid forcing the components to implement an interface, and to avoid the problems with dynamic proxies, the same mechanism as used in the alternative QoS aware implementation for the π components is used[9]. This implementation used byte-code manipulation to add the QoSAware interface to selected components. The same mechanism can also be used to add an arbitrary interface to any QuA component. When the interface is added in the byte-code before the class is loaded by a classloader, it can be used as a pointcut by an AOP framework that supports load-time weaving.

The byte-code modification mechanism is used to add an interface, *ReifiableQuAComponent*, to the main-class of all QuA components. This interface supports one method, *reify()*, which returns a meta-component for the QuA component.

[9]The alternative QoS aware implementation for the π components are described in section 5.1.1.2 on page 87.

Figure 5.9 UML Class diagram showing the meta-model for components.

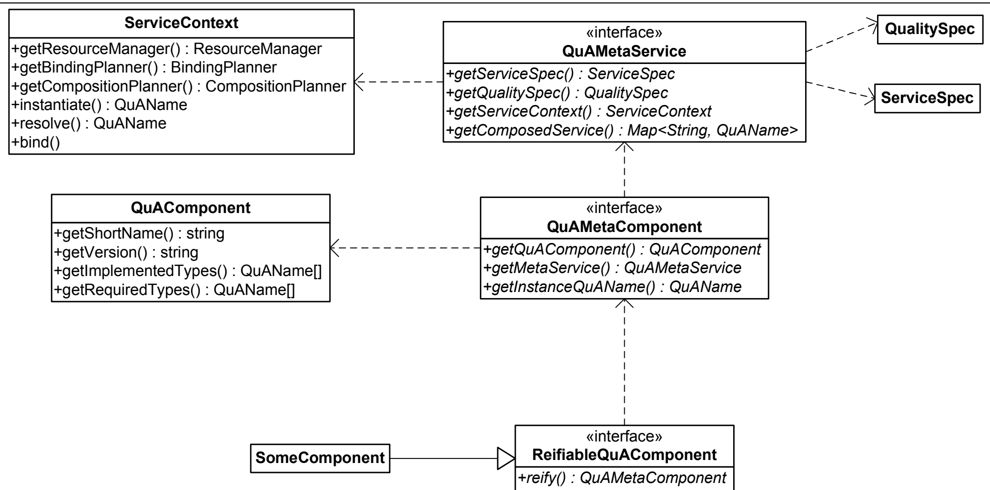


Figure 5.9 shows a UML class diagram describing the meta-model provided by the *reify* method. The figure shows an arbitrary component, *SomeComponent*, whose byte-code has been modified to include an implementation of the *ReifiableQuAComponent* interface. This interface provides the reify method, which gives access to the *QuAMeta-Component*.

The *QuAMetaComponent* provides access to the *QuAComponent*[10] object. This object provides meta-information about a component. E.g., its short-name, version, required-types, etc. The diagram in figure 5.9 shows only the most important methods of the *QuAComponent*.

If the component is part of a service, the QuAMetaComponent also provides access to meta-information about the service. This meta-information includes information about the component instances used in the service.

The *ReifiableQuAComponent* interface is added by using byte-code manipulation and the meta-model is populated with aspects. The QuAComponent is added to the meta-model when the component is instantiated, and the QuAMetaService is added to the meta-model when the service planned. The *instanceQuAName* in QuAMetaComponent is added when the component is added to a repository.

Some extra precaution must be taken when the service includes remote components. In the audio service, there are remote components (e.g., the encoder). The aspect that populates the meta-service checks all the components that are part of the composition. If a component is remote, it will instantiate a helper component, *QuAQRAPReflection*, in the remote capsule. The aspect then sends all the information necessary to populate the QuAMetaService object for the remote component to the QuAQRAPReflection component. The only exception is the *QualitySpec*, which is an interface implemented by the user of the service. The QuAQRAPReflection component will then populate the

[10]The *QuAComponent* object was called *ComponentBlueprint* in early versions of the QuA Java prototype. This name is used in the section describing the π components. In the prototype version used here, it is called *QuAComponent*, and in the newest version of the QuA Java prototype, the name has changed back to *ComponentBlueprint*.

QuAMetaService object for the remote component. Thus, all the components in the composition will have meta-information attached.

This meta-model adds limited support for *architectural reflection* to QuA. It is limited because it only supports *introspection*. I.e., it only allows reading of the meta-model, not modification of it. The meta-model is added with aspects. I.e., the byte-code manipulation mechanism is added with aspects, and the population of the meta-model is added with aspects. This implies that support for architectural reflection is configurable. If architectural reflection is needed, the aspects can be added at startup. If architectural reflection is not needed, there is no need to add the aspects.

5.2.4.2 The Adaptation Aspect

An adaptation aspect intercepts all QoS aware components. The analysis has suggestions on how such an aspect might look like (figure 4.3.1 and figure 4.3.2 on page 67). The suggestions in the analysis uses AspectWerkz features with AspectJ syntax. The ability for an *around advice* to proceed to a different target is an AspectWerkz feature. To implement this with AspectJ, we have to use the Java reflection API to invoke a method on a different object.

The implemented adaptation aspect is based on the suggestion in the analysis.

Figure 5.10 Pseudo-code for the adaptation aspect.

```
aspect AdaptationAspect {
  pointcut accessQoSAware: execution(public * qua.types.QoSAware +.*(..))  &&
        !cflowbelow(within(AdaptationAspect))

  Object around(): accessQoSAware {
    ReifiableQuAComponent targetComponent = (ReifiableQuAComponent)jp.getTarget();
    QuAName instanceName = targetComponent.reify().getInstanceQuAName();
    ServiceContext ctx = targetComponent.reify (). getServiceContext ();
    AdaptationManager am = ctx.getAdaptationManager();

    Object newTarget = am.getNewTarget(instanceName);
    if (newTarget != null) {
      // Proceed to the adapted component
      return proceedToNewTargetUsingReflection(newTarget);
    } else {
      // No adapted component exists , proceed to original target
      return proceed();
    }
  }
}
```

Figure 5.10 shows pseudo-code for the adaptation aspect. All public methods on objects that implement the QoSAware interface are intercepted. The adaptation manager is asked for a replacement object for the target component, i.e., a new adapted component. If an adapted component exists, execution proceeds to that component. If not, execution proceeds normally.

5.2.4.3 The Adaptation Manager

The adaptation aspect uses an adaptation manager to determine which component should execute a request. The QuA Java prototype is changed to include an adaptation manager component. This component is loaded when a capsule is initialized and is available from the service context.

Figure 5.11 The adaptation manager interface.

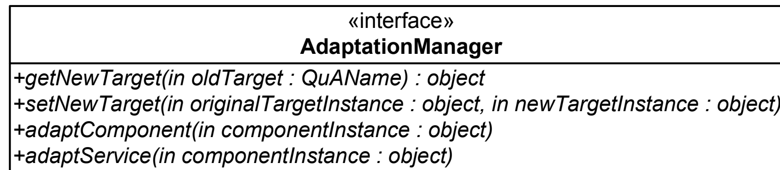


Figure 5.11 shows the interface for the adaptation manager:

getNewTarget This method is used by the adaptation aspect. The *oldTargetInstance* parameter refers the component under consideration. The return value is a new component that replaces the original, or null, if no replacement exists.

setNewTarget This method adapts a component. The *originalTargetInstance* adapts to *newTargetInstance*. This method supports both compositional and parameter adaptation.

adaptComponent This method uses the service planner to find a new component. Architectural reflection is used to get the quality specification of the original component. This specification is used when requesting a new component.

adaptService This method adapts a complete service. The parameter is a component in the service. Architectural reflection is used get access to the service specification for the service.

Compositional and Parameter Adaptation

Most of the work is performed in the *setNewTarget* method. This method will:

1. Determine whether compositional or parameter adaptation should be used.
2. Transfer bindings from the original to the new component if compositional adaptation is used.
3. Update the reflection of the service.
4. Stop the original component and start the new component.

If the old and new component refers to the same blueprint, e.g., both refers to a SpeexEncoder component, parameter adaptation can be used. A requirement for using parameter adaptation is that the component implements the *Reconfigurable* interface. This interface contains one method:

```
void reconfigure (QoSStatement qs, ResourceStatement rs );
```

The parameters are the same as in the *configure* method in the QoSAware interface. When parameter adaptation is used, the old component is reconfigured to use the QoSStatement and ResourceStatement in the new component.

If parameter adaptation can not be used, compositional adaptation is used. This will replace the original component with a new component. When a component is compositionally adapted, each call to *getNewTarget*, with the old component as parameter, will return the new component.

The new component must inherit the bindings from the original component. E.g., if the adapted component is an encoder, it must use the same *RTDataSink* as the original component. All the bindings are available in the reified service, i.e., the *QuAMetaComponent* and *QuAMetaService*. The meta service is inspected for bindings, and those bindings are transferred to the new component.

Many of the components in the service start their own threads. This means that they must be started and stopped when adaptation occurs. This situation is not handled in the analysis, and shows a weakness in the analysis.

To make the adaptation manager able to start and stop components, a new interface is introduced: *ActiveComponent*. This interface contains two methods: *startComponent()* and *stopComponent()*. If a component implements this interface, the adaptation manager will stop the old component and start the new component. The notion of active components can also be found in COMQUAD (Göbel et al., 2004b)[11].

When a component is adapted, the reflection of the service it is part of must also be updated. The *QuAMetaService* objects contains a map from *role name* to QuA name. Role name is the role name in the service specification, e.g., "encoder". The adaptation manager updates the meta-service to use the new component.

Optimizing Compositional Adaptation

Adapting a component adds an extra layer of indirection. If a component is adapted more than once, we get multiple layers of indirection. This can be optimized in the adaptation manager.

E.g., assume that the original audio service uses a GSMEncoder component. This component adapts to a SpeexEncoder, and then back to a GSMEncoder. When the *receive* method on the original GSMEncoder is called, the adaptation aspect will ask the adaptation manager for a new component. The adaptation manager returns the SpeexEncoder, and the adaptation manager proceeds execution to the SpeexEncoder.

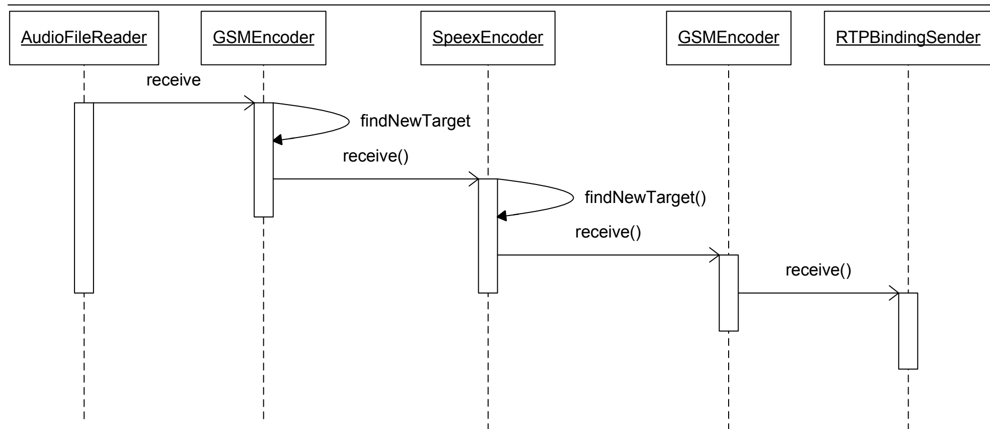
The adaptation aspect will intercept the invocation of the SpeexEncoder, and ask the adaptation manager for a new component. The adaptation manager returns a GSMEncoder, and the aspects proceeds execution to the GSMEncoder.

Figure 5.12(a) shows a UML sequence diagram describing this situation. The sequence diagram is simplified for brevity, the adaptation manager and the adaptation aspect is not shown.

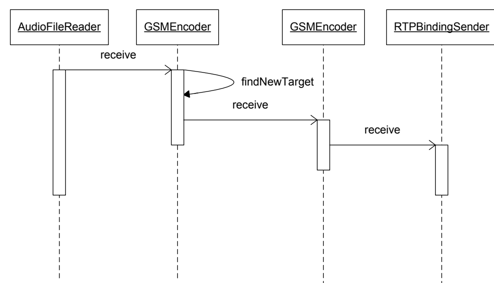
The adaptation manager optimizes the situation in figure 5.12(a). When the *setNewTarget* method is called with the Speex encoder as the original component and the GSM encoder as the new component, the adaptation manager recongnizes the Speex encoder as a new target for the original GSM encoder. Thus, the new GSM encoder is not only

[11] See section 2.7.1 on page 25 for a description of COMQUAD.

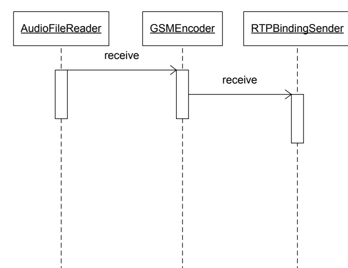
Figure 5.12 Simplified UML sequence diagrams for adapted components.



(a) Unoptimized adaptation.



(b) Optimized adaptation.



(c) Adaptation with altered bindings.

a replacement component for the Speex encoder, but also a replacement component for the original GSM encoder.

The situation in figure 5.12(b) can be further optimized. The binding specification in the meta-service is used to find all components having bindings *to* the adapted component. In the audio service, the *AudioFileReader* is the only component having a binding to the encoder. The adaptation manager uses the meta service to change the bindings in the AudioFileReader component to use the new encoder component.

Not all components support change of bindings during run-time. The adaptation manager requires that a component must implement the *ReconfigurableBindings* marker interface in order to change bindings during run-time. The AudioFileReader component implements this interface. Figure 5.12(c) shows the result of this optimization.

Transferring State

When compositional adaptation is used on components that need to transfer state, the memento pattern (Gamma et al., 1995) is used. This is the case for the RTDataBuffer component.

The adaptation manager checks whether the new and the original component implements the *MementoOriginator* interface. If they do, a memento is transferred from the original component to the new component.

Adapting a Complete Service

The *adaptService* method in the adaptation manager adapts a complete service. The only parameter to the method is an object that is part of the service. The adaptation manager uses architectural reflection on this object in order to retrieve the service specification and the quality specification for the service.

This is used to compose a new service using the original quality specification. The adaptation manager iterates through all components in the new service and adapts the corresponding component in the original service. The *setNewTarget* method is used for this. This implies that parameter adaptation is used when possible and that compositional adaptation is used when parameter adaptation can not be used.

The *adaptService* method can only be used in the same QuA capsule that created the original service. This is because the quality specification is not accessible in remote capsules. A more sophisticated implementation of the adaptation manager could idententify the capsule that created the original service, and delegate adaptation to the adaptation manager in that capsule.

Adapting the Decoder Components

The encoder component and the decoder component must support the same codec. If the encoder changes from using GSM to using Speex, the decoder must also change from using GSM to using Speex. There is no way to state this dependency in the QuA Java prototype. If the encoder change, we can not state that the decoder also has to change.

This situation is described in the analysis, section 4.3.5.2 on page 70. The analysis suggests making a dedicated aspect for handling the situation.

An aspect is added to intercept calls to the *AudioDecoder.receive()* method. All decoders receive audio packets with that method. The aspect then inspects the received *RTPacket*. This packet contains information about the codec used. The codec in the packet is compared with the codec implemented by the decoder. If they use different codecs, the decoder must adapt. The method *getImplementedCodec* was added to the AudioDecoder interface in order to make it more convenient for the aspect to detect which codec the decoder component implements.

The aspect uses the QoS aware service planner to find a new decoder component supporting the codec in the packet. To make it possible for the service planner to find a component that implements a given codec, a quality dimension for codec is added to the quality model for audio. I.e., a QoS constraint that determines which codec to use can be included in the service planning process.

The aspects that adapts the decoder is dependent on the adaptation aspect described in section 5.2.4.2. The decoder aspect must have precedence over the adaptation aspect. If a packet containing Speex encoded data is sent to a GSM decoder, the decoder aspect must run first in order to set a new target for the GSM decoder. If the adaptation aspect runs first, the new decoder component will not be used until the next packet with audio data arrives. Thus, we will loose one audio packet as the GSM decoder will discard packets containing other codecs.

5.2.5 Resource Management

The aspects for resource reservation created in the π component experiment (see section 5.1.2 on page 88) are reused in the audio service. The resource reservation aspects require a QoS aware service. This is because the aspects read the components' configured resource requirements from the *getConfiguredResources()* method in the QoSAware interface, and this configuration is set by the QoS aware generic implementation planner.

The aspect is extended to also separate resource management from the adaptation manager. There are two methods in the adaptation manager that instantiates new components: *adaptComponent* and *adaptService*. Both methods need to release resources for the original component before instantiating a new component. If resources are not released first, there might not be enough resources left to instantiate the new component. The resource reservation aspects are extended to handle the resource management required by those methods.

As the QoS aware generic implementation planner does not support remote capsules, the resource reservation aspects are only tested on a version of the audio service that do not use remote capsules, i.e., both the audio source and the audio sink run in the same capsule. This local service still uses UDP for transferring audio packets.

If the generic implementation planner supported remote capsules, we could not reuse the resource reservation aspects without modification. In a distributed service, the resource reservation aspects would only reserve resources in the local capsule. To make them work for distributed services, the same technique as used for architectural reflection could be used; make the resource reservation aspects instantiate a helper component in the remote capsule and use this component to reserve the remote resources.

This approach is not implemented as it is difficult to test it without support for remote capsules in the generic implementation planner. However, the same concept is tested with the architectural reflection aspects and should also work in this case.

5.2.6 Testing the Adaptation Mechanism

The audio player test application described in section 5.2.2 on page 94 is used for testing the adaptation mechanism.

5.2.6.1 Testing with a Local Capsule

As the QoS aware service planner in the QuA prototype does not support remote repositories, the main part of the testing is done with a local QuA capsule. Using a local QuA capsule implies that the encoder and the decoder reside in the same QuA capsule. However, UDP is still used for sending audio packets between the RTPBindingSender component and the RTPBindingReceiver component.

Adapting the Decoders

The first test checks whether adaptation of the decoders work. To test this, a service using a Speex encoder and a GSM decoder is created. The components are configured to output debug statements. By examining the debug log we can see that the decoder is adapted. This is also confirmed by listening to the audio stream – if it did not work, we would not hear a sound.

By looking at the debug log, it is also confirmed that the aspects for architectural reflection and resource reservation works.

Adapting the Encoders

The audio player test application contains a drop-down list where a codec and codec configuration can be selected (e.g, Speex 32kHz stereo). The application is changed to let this also be selectable during run-time.

If a new codec is selected during run-time, the encoder will change. The QoS aware service planner is used to find a new encoder component that implements the selected codec. The QoS statement used to find the component also contains the selected sample rate and number of channels.

The component is then adapted by using the *setNewTarget* method in the adaptation manager.

The following can be observed:

- Adaptation is smooth – i.e., the sound in continuous and no artifacts can be heared.
- Parameter adaptation is used if only the configuration is changed and not the codec. I.e., if the original configuration was Speex 32kHz stereo and the new configuration is Speex 16kHz mono, parameter adaptation is used to reconfigure the Speex encoder component.
- Adaptation of the decoder component still work.

This test also tests the optimizations for compositional adaptation described on page 102. If we change from GSM to Speex and then back to GSM again, the debug log shows that the adaptation aspect uses the last GSM encoder as a replacement for the original GSM encoder. It does not replace the original GSM encoder with the Speex encoder, and then replace the Speex encoder with the GSM encoder.

By adding the *ReconfigurableBindings* interface to the *AudioFileReader* component, the adaptation is further optimized. In this case, the debug log shows that the adaptation aspect do not proceed to a new component as the bindings in the AudioFileReader component are updated to use the newly adapted component directly.

The original encoder implementations tried to close the data sink they were using when they were stopped. The RTPBindingSender component is the data sink for the encoder components. If this component is closed, it will stop working. This had to change in order to get adaptation to work.

Adapting the Buffer

To test transferring of state, the RTDataBuffer component is used. The component is changed to implement the *MementoOriginator* interface, thus making it capable of transferring state. To make sure that the component is adapted, two identical component blueprints are created. The only difference is the implementation of the QoSAware type; one buffer only accepts the Speex codec, and the other only accepts the GSM codec.

The audio player test application is changed to also adapt the buffer component when it adapts encoder component. The debug log shows that the buffer is compositionally adapted and that state is transferred.

If the buffer is adapted many times, we can observe occasional packet loss. This is detected by the monitoring aspects and can also be heard. One packet is lost in about one out of eight adaptations of the buffer.

This happens because the original buffer is stopped before transferring state to the new component and starting the new component. If a packet is received after the buffer is stopped and before the new component is started, it will be lost.

Adapting a Complete Service

Adaptation of a complete service is also tested with the audio player application. The first test for this was a complete failure. The debug log showed that all components adapted successfully, but no sound could be heard.

The adaptation approach taken here is quite agressive. It assumes that a component by default is compositionally adaptable if it is QoS aware. To enable parameter adaptation for a component, extra effort is needed – i.e., it does not support it by default.

Some components are clearly not suited for compositional adaptation. E.g., the AudioFileReader component is not suited for compositional adaptation. The UDP binding and the speaker components are neither suited for compositional adaptation.

After changing those components to enable parameter adaptation, adapting the complete service worked.

The speaker component might never be suitable for compositional adaptation as used in this thesis. The speaker component uses the Java sound API for playing sound samples. This API buffers the samples sent to it. If you send samples faster than they can be played, they are buffered. This buffer is unreachable outside of the API. Thus, it is not possible to transfer the state from the speaker component.

To successfully adapt the speaker component compositionally, we would have to wait until the internal buffer in the Java Sound API is emptied before stopping the component and replacing it with a new component. This is not possible with the adaptation scheme used in this thesis.

To handle this, the adaptation scheme must be extended to let the component perform adaptational behaviour. Such adaptation schemes can be seen in QuO (Duzan et al., 2004)[12] and in (Almeida et al., 2001).

5.2.7 Testing with Remote Capsules

Testing with remote capsules is limited as the QoS aware service planner does not support remote components. To test this, the audio player is configured to use the basic service planner – i.e., the default service planner with no QoS support. Only remote reflection and adaptation of the encoder component is tested with remote capsules.

Whether to use remote or local capsules are a configurable option to the audio player test application. When a local capsule is used, the QuA URL is ignored. When a remote capsule is used, the remote capsule is looked up based on the QuA URL, and the service is composed without a QoS aware service planner.

Testing component adaptation with a remote capsule requires changes to the adaptation test code used in the previous tests. Instead of just using the service planner to

[12]See section 2.7.2 on page 26 for a description of QuO.

find a new component for a given quality specification, a specific component must be instantiated in the remote capsule.

Only the encoder components are adapted in this test. To adapt the encoder, the test application creates a new encoder component in the remote capsule. Then, it resolves the adaptation manager in the remote capsule and calls the *setNewTarget* method using the QuA remote access protocol.

The debug log confirms that the component adapts. It also confirms that the reflection aspect is able to transfer the reflection of the service to the new capsule. Adaptation of the encoder can also be confirmed by listening to the service, and by watching the decoder component being adapted by the decoder adaptation aspect.

5.2.8 Summary

This experiment shows that AOP can be used for monitoring QoS, architectural reflection, resource management and for adaptation of a service.

An audio player application was created in order to test the aspects in the experiment. This application contains a simplified QoS monitor. Aspects were created for monitoring some QoS dimensions, and they report to the QoS monitor in the audio player application.

Aspects were used to create a meta-model of components and services. This meta-model provides architectural reflection capabilities to QuA. The architectural reflection capabilities were used to adapt components and services. Both compositional adaptation and parameter adaptation were enabled with AOP. The aspects are capable of adapting a single component or a complete service.

Enforcing a consistent service is not guaranteed by the adaptation mechanism alone. To ensure that the decoder component matches the encoder component, a separate aspect was created. This aspect adapts the decoder component to fit the codec used by the encoder component.

The adaptation mechanism uses an optimistic approach. It assumes that all QoS aware components are adaptable. The experiment has showed that this is not always the case. It has also showed an example of a component that can not be adapted with the adaptation mechanism used.

The aspects used for monitoring, architectural reflection and adaptation, only use information from the contractual specifications of components, i.e., they use black box encapsulation. The audio service is able to stream audio without enabling the aspects. The aspects are only needed if monitoring or adaptation should be enabled. Enabling this, and enabling the aspects, is a matter of configuring the application.

The resource reservation aspects used in the previous experiment on π components were reused in this experiment and extended to also separate resource management from the adaptation manager.

All the code produced in the experiments are available from the QuA Subversion repository:

https://svn.simula.no:40081/svn/nd/code/QuA/branches/private/toreen/trunk.

Chapter 6

Evaluation

The chapter contains an evaluation of the experiments. The experiments are evaluated based on the criteria set forth in the analysis chapter. A conclusion is presented together with a summary of the results of the thesis. In the experiments, aspects are enabled by adding some parameters to the Java VM when starting a QuA capsule. The topic of aspect deployment and aspect components are proposed as subject for further work.

6.1 Evaluating the Experiments

The analysis (section 4.4.1.4 on page 80) suggests that the experiments should be evaluated based on whether they manage to separate concerns and on the degree of encapsulation, modularity and reusability.

The experiments validate the conclusion of the analysis to some degree:

- The experiment with π components showed that we could separate static QoS from the component implementations. The method for achieving this was reused in the experiment on audio streaming.
- The resource management concern can be separated from the component implementations and from the application using the components. By using AOP for resource reservation, neither the component implementations nor the audio test application needs to contain code for resource management.
- QoS monitoring can benefit from AOP. Some of the QoS dimensions relevant for audio streaming are monitored by aspects.
- The analysis suggests that architectural reflection should be used for transferring bindings during adaptation. The experiments show how architectural reflection can be added to QuA with AOP.
- The adaptation manager and adaptation aspects in the experiments show how to adapt components and whole services, both by using compositional and parameter adaptation.

The experiments show that some concerns can be separated using AOP. Resource management and adaptation are completely separated concerns. Resource monitoring is not tested in the experiment, but monitoring of some QoS dimensions is separated with AOP.

However, some components had to change their implementations in order to make adaptation work; one component also had to extend its interface (i.e., its contractual specification) in order to make adaptation work.

The experiments showed some weaknesses in the analysis as the analysis did not handle the concept of active components. Active components run in their own thread and need some simple life-cycle methods in order to be adapted, i.e., they must be started and stopped.

Also, the mechanism for adaptation described in the analysis is too optimistic. The method described in the analysis assumes that every QoS aware component can be adapted. This is not the case. A better approach would be to explicitly mark the components that are adaptable, e.g., by adding a marker interface.

The experiments showed an example of a component that can not be compositionally adapted with the mechanism described in the analysis. The speaker component needs additional behavior in order to synchronize its internal buffer when compositional adaptation is used. This indicates that the adaptation mechanism should be extended to let the components under consideration take part in the adaptation process.

The experiments on static QoS was implemented using the AspectWerkz AOP framework, and the experiments on audio streams was implemented using an early access release of AspectJ 5. Common for the experiments is that *load-time weaving* is used. To enable the aspects, the Java VM running the QuA capsule must be started with some extra parameters, and with the compiled aspects in its classpath. This implies that it is the *deployer's* responsibility to enable aspects. The deployer is responsible for configuring the middleware to suit the application's needs.

6.1.1 Modularization and Reuse

To have a proper separation of concerns, we must have a proper modularization of the separated concerns. The analysis[1] suggests using the degree of encapsulation and orthogonality to evaluate this. A high degree of encapsulation and orthogonality is also assumed to promote reuse.

Four concerns were separated: Static QoS, QoS monitoring, resource management and adaptation.

6.1.1.1 Static QoS

Static QoS is part of the QuA QoS aware service planning process. Separation of static QoS means that the implementation of the QoSAware type is separated from the component implementation. The QoSAware type is required for QoS aware service planning, and provides a mapping between QoS requirements and resource demands.

Separation of the static QoS concern takes a black box approach to the components

[1] See the analysis section 4.4.1.1 on page 72 and section 4.4.1.2 on page 75 for a discusssion of reuse and orthogonality.

it acts upon. I.e., it assumes nothing about the implementation of the components it acts upon. However, it do utilize knowledge about the intrinsics of the QuA core. Knowledge about the QuA core's loading process for component blueprints is used to change the meta-information about component blueprints and to instrument the Java class implementing the component.

Even though the aspect for separating the QoSAware type considers components as black boxes, the actual *implementation* of the QoSAware type can not consider the component as a black box. E.g., to provide a mapping between QoS requirements and resource demands for an audio encoder component one would need to measure the resource demands for the component. It is also likely that inspection of the component implementation – i.e., a whitebox approach – is needed to create such a mapping.

Following the understanding of orthogonal aspects by Colyer et al. (2004), the aspect for separating static QoS is orthogonal to the components. It is safe to both add and remove the aspect to the component.

The orthogonality of the aspect, and the black box approach to encapsulation, indicates that this is a reusable aspect. However, the dependencies on the intrinsics of the QuA core makes it volatile to changes in the QuA core.

6.1.1.2 QoS Monitoring

Monitoring of some QoS dimensions are separated as aspects. Packet loss, sample rate, theoretical MOS, latency and jitter are monitored with aspects. Three aspects are created in order to monitor this, each aspect monitors different QoS dimensions.

All the aspects operate only on the contractual specification of the components, i.e., they consider the components as black boxes. They are also orthogonal to the components, as they can both be safely added and removed.

However, some of the aspects do make assumptions about the composition of the components. If the buffer is removed from the client side of the composition, jitter and latency will not be monitored.

Making assumptions about the component composition is reasonable when the aspects monitors QoS for a service. Thus, the orthogonality and the black box approach to components indicates that the monitoring aspects are reusable for similar services.

6.1.1.3 Resource Management

All reservation and releasing of resources are separated as an aspect. The aspect intercepts components defined as a part of the QuA core – i.e., the implementation planner and the adaptation manager – in order to reserve and release resources.

The resource management aspect is orthogonal to the QuA core components it intercepts. Neither the adaptation manager nor the implementation planner alter their behavior if the resource management aspect is activated. The aspect only operates on the contractual specification of the components, i.e., the components are considered black boxes. This implies that the behaviour of the aspect should not change for different component implementations.

The aspect for resource management only handles resources in local capsules. For distributed services, a helper component, like the helper component used for distributed architectural reflection, would be needed.

6.1.1.4 Adaptation

In order to separate adaptation, an aspect that provides architectural reflection for QuA components and services was created. Both compositional and parameter adaptation are supported.

The aspect for creating architectural reflection utilize the same knowledge about the intrinsics of the QuA core as the aspect for separating static QoS. I.e., knowledge about the QuA core's loading process for component blueprints is used to instrument the Java class implementing the component in order to attach an implementation of a reflection class.

Also, knowledge about the QuA control flow is used to populate the meta-model. Meta information about the service a component is part of are gathered when the service is composed. The QuA name for the instantiated component is gathered when the component is placed in the volatile repository.

Such knowledge about the QuA core makes it volatile to changes in the core, but as stated in the analysis (section 4.1.3 on page 50), a well defined program flow is required to handle such concerns.

The aspect for creating architectural reflection makes use of a helper component. This helper component is used if the service is distributed on multiple QuA capsules. The helper component is used to populate the meta-model for components and services in the remote capsules.

The aspect regards components and services as black boxes. It is orthogonal to the components, but not to the QuA core.

Adaptation Manager

For adaptation, an adaptation manager component is created. This component is considered a pluggable part of the QuA core. The adaptation manager relies on architectural reflection to perform adaptation. Following the understanding of orthogonal aspects by Colyer et al. (2004), the adaptation manager is *weakly orthogonal* to the aspect for architectural reflection if architectural reflection is considered part of the adaptation concern. The adaptation manager updates the meta-information in the reflection of a service when components in the service are adapted.

An adaptation aspect is also needed to make adaptation work. This aspect intercepts calls to components, and checks with the adaptation manager whether the call should be delegated to another component. In some situations, this aspect is not needed. If a component is declared to have reconfigurable bindings, the adaptation manager will reconfigure its bindings to use the new component directly. In those cases, the adaptation aspect is not needed.

The optimistic approach for adaptation in this thesis – i.e., assuming that all QoS aware components by default are compositionally adaptable – is not orthogonal to the components that should be adapted. Some components are not compositionally adaptable and will malfunction if they are adapted.

Changing the adaptation mechanism to require adaptation support from components before adapting them is a trivial change. Doing so would make the adaptation mechanism orthogonal to the components.

Thus, one might say that the adaptation mechanism is orthogonal to components

prepared for adaptation, but that the *adaptation concern* is not orthogonal to arbitrary components. I.e., the adaptation concern can not be separated for arbitrary components or services, it requires adaptation support in the adapted components. This confirms the observation by Almeida et al. (2001).

Ensuring Composition Integrity

A newly adapted component, or service, is valid according to the service specification. The service specification is the only means for the adaptation manager to create a valid service. Not all constraints to ensure a valid service are kept in the service specification. The relationship between the encoder component and the decoder component is impossible to express in the service specification. This relationship should ensure that the decoder is able to decode the packets generated by the encoder, i.e., that they use the same codec.

To express this dependency in a general way, additional architectural constraints (Blair et al., 2001) is needed when specifying the service. *Architectural style* (Garlan et al., 2004) is one way describing such constraints.

Instead, a separate aspect is created to ensure that the encoder and decoder use the same codec. This aspect is not part of the general adaptation mechanism, but is regarded as a special purpose aspect for the decoder components. This aspects monitors the packets sent to a decoder component and adapts the component if the codec supported by the decoder do not match the codec used in the packet.

The aspect only operates on the contractual specification for decoder components. It is also orthogonal to the decoder component, i.e., the decoder do not need the aspect to run, but an adaptable service needs the aspect to ensure a valid component composition. Thus, the aspect should be reusable for any decoder component.

The aspect depends on the adaptation manager in order to adapt the decoder component. As the adaptation manager is regarded as a part of the QuA core, this is not regarded as an unwanted coupling or dependency. However, the aspect must have *precedence* over the adaptation aspect to prevent loosing the first packet received when the decoder is adapted. This is an unwanted coupling to the adaptation aspect.

6.1.1.5 Summary

The aspects used to separate concerns are mostly orthogonal to the services and the components in the services. Most of the aspects also regard the components as black boxes, i.e., they operate on the components' contractual specifications and do not make assumptions about their internal behavior.

The adaptation mechanism is not orthogonal to components and services when using the approach described in the analysis. That approach assumes that all QoS aware components are compositionally adaptable, which is not the case. However, only small changes to the adaptation mechanism are required to change this behavior.

The implementation of some of the components in the audio service had to change in order to get the adaptation mechanism to work. This implies that the components could not be reused as black boxes when adaptation was enabled. If the adaptation mechanism is changed to require the components to explicitly state whether they support adaptation, the components could probably be reused as black boxes and the adaptation mechanism

would be orthogonal to the service. The orthogonality of the aspects indicates that they are reusable.

For the various roles in QuA, the separation of concerns into aspects implies that the *component developer* can focus on implementing components – e.g., π components or audio codecs – and that a *QoS expert* can add QoS capabilities to the components (i.e., an implementation of the QoSAware interface) without having to change the component implementations. The QoS expert also has to enable QoS in the application (i.e., provide a *quality speicification* for services). The *application developer* can focus on developing applications using QuA components, and the *deployer* is responsible for enabling aspects by configuring the middleware to suit the application's needs.

The aspects used for separating resource management relieves not only application developers and component developers from the task of handling resources, *platform developers* are also relieved from this task. The aspect for resource management handles releasing of, and reservation of, resources for both the adaptation manager and the service planner.

There are no empirical data showing whether we have achieved a more modular implementation with reduced complexity. There is no other implementation to compare with. There are also no quantitative measures for the modularity and complexity in the implementation. However, quantitative measures are not needed to show that the code for handling concerns are modular. The orthogonal properties of the implementations also indicate that the different concerns are well separated in a modular fashion. It is reasonable to assume that an alternative implementation not using AOP would have problems reaching the same degree of modularity. Thus, it is also reasonable to assume that an alternative implementation not using AOP would be contain more complexity.

6.2 Aspect Components

There are some loose ends in this thesis. Loose ends which it is beyond the scope of the thesis to investigate, but which nevertheless is interesting topics for research. Most interesting are the topics of aspect deployment and aspect components.

The aspects in the thesis are packaged in standard Java jar files. The jar file contains the compiled aspects and a meta-file declaring some of the classes in the jar file as aspects. When a QuA capsule is started, some extra parameters are added to enable load-time weaving with AspectJ. When the jar file containing the aspects are added to the classpath, the aspects will be detected by the AspectJ run-time, which will weave in the aspects.

It should be possible to package aspects as components. E.g., the QuA component blueprint loader could interact with the AspectJ weaver in order to "deploy" the aspects. If an aspect framework supporting run-time weaving (e.g., AspectWerkz) is used, the newly deployed aspect could also affect already loaded components.

Packaging aspects as components would make them more fit for a component-based middleware platform. It would make it possible for aspects to explicitly state their dependencies. It would also make it possible to discover and load aspects from remote component repositories.

One could also imagine packaging a set of aspect components together with an ordinary component. Thus, packaging a complete concern. E.g., the adaptation manager

component, adaptation aspect and the architectural reflection aspect could be packaged as an "adaptation concern".

How to package and deploy aspects is an interesting topic for further research.

6.3 Results

The QuA project is investigating how a component architecture can preserve the safe deployment property for QoS sensitive applications. For components deployed on such a platform to be reusable, environment dependent implementation decisions must be separated from the implementation. To achieve this, we need platform-managed QoS. This implies that the middleware platform must be able to adapt components and services to suit their QoS requirements as resource availability changes.

This thesis shows how concerns such as adaptation and QoS monitoring can be separated from the component implementations using aspect oriented programming. AOP is used to enable architectural reflection for components and services deployed on a Java prototype of the QuA platform. This is used by an adaptation mechanism, consisting of aspects and components, to dynamically adapt a running service. Both parameter and compositional adaptation are supported.

Experiments conducted in this thesis also shows how monitoring of some QoS dimensions can be separated from the application code using AOP. It is also showed how resource management can be separated from both the component implementations and from the QuA core components. Finally, it is showed how static QoS can be separated from the component implementations.

Using AOP to separate these concerns has resulted in a modular separation of concerns. This modularity allows for enabling and disabling of different concerns as a deployment option to the middleware. E.g., adaptation is only enabled if it is needed. This suits the idea of component-based middleware where the middleware itself is composed of different components in order to suit the needs of an application.

AOP seems to modularize cross-cutting concerns well in this thesis. Thus, it seems like a promising technique to enable platform managed QoS within QuA. Although no quantitative measures for the complexity in the experiments exists, it is reasonable to assume that the techniques used in the experiments reduce complexity compared to other approaches.

The experiments showed that the adaptation mechanism suggested in the analysis is too optimistic for some cases, and that there are some cases where the suggested mechanism can not be used to compositionally adapt a component. A suggestion on how to improve the mechanism is described, and it would be interesting to see this mechanism implemented and tested in other cases. Also, it would be interesting to see the topic of aspect components elaborated in future experiments. Creating components out of aspects might provide some of the same benefits as creating components out of objects.

References

Aagedal, Jan Øyvind. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.

Abrahamsen, Espen. Mapping of QoS-enriched models to a generic resource model. Master's thesis, University of Oslo, 2005.

Aksit, Mehmet, L. Bergmans, and S. Vural. An Object-Oriented language-database integration model: The composition-filters approach. In *Proceedings of ECOOP'02*, number 615 in LNCS, pages 372–395, Utrecht, The Netherlands, June 1992. Springer-Verlag.

Aksit, Mehmet and Z. Choukair. Dynamic, adaptive and reconfigurable systems overview and prospective vision. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops(ICDCSW'03*, pages 84–89. IEEE, May 2003.

Almeida, João Paulo A., M. Wegdam, L. F. Pires, and M. van Sinderen. An approach to dynamic reconfiguration of distributed systems based on object-middleware. In *Proceedings of the 19th Brazilian Symposium on Computer Networks (SBRC 2001)*, 2001.

Andersen, Anders, G. Blair, V. Goebel, R. Karlsen, T. Stabell-Kulø, and W. Yu. Arctic beans: Configurable and re-configurable enterprise component architectures. *IEEE DS Online*, 2(7), Nov. 2001.

Bailey, D. H., J. M. Borwein, P. B. Borwein, and S. Plouffe. The quest for pi. *Math. Intelligencer*, 19(1):50–57, 1997. ISSN 0343-6993.

Bergmans, Lodewijk and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001. ISSN 0001-0782.

Berset, Geir. Strategic management to support quality of service. Master's thesis, University of Oslo, 2004.

Blair, Gordon S., G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The design and implementation of Open ORB 2. *IEEE DS Online, Special Issue on Reflective Middleware*, 2(6), 2001.

Blair, Gordon S., G. Coulson, and P. Grace. Research directions in reflective middleware: the Lancaster experience. In *Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 262–267. ACM Press, 2004. ISBN 1-58113-949-7.

Bonér, Jonas. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD'04)*, pages 5–6. ACM Press, 2004. ISBN 1-58113-842-3.

Bracha, Gilad and W. Cook. Mixin-based inheritance. In Meyrowitz, Norman, editor, *OOPSLA - ECOOP'90 Proceedings*, number 1241 in ACM SIGPLAN, pages 303–311. Addison-Wesley, June 1990. ISBN 0-0201-52430-4.

Bruneton, Eric, T. Coupaye, and J.-B. Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the 7th International workshop on Component-Oriented Programming (WCOP'02) at ECOOP'02*, 2002a.

Bruneton, Eric, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Systèmes à composants adaptables et extensibles*, Oct. 2002b.

Cazzola, Walter, A. Savigni, A. Sosio, and F. Tisato. Architectural reflection. Realising software architectures via reflective activities. In Emmerich, Wolfgang and S. Tai, editors, *Proceedings of EDO 2000*, number 1999 in LNCS, pages 102–115. Springer-Verlag, Nov. 2001.

Chiba, Shigeru and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proceedings of the 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03)*, number 2830 in LNCS, pages 364–376. Springer-Verlag, 2003.

Chiba, Shigeru. Load-time structural reflection in Java. In *Proceedings of ECOOP 2000*, number 1850 in LNCS, pages 313–336. Springer-Verlag, 2000.

Cilia, Mariano, M. Haupt, M. Mezini, and A. Buchmann. The convergence of AOP and active databases: Towards reactive middleware. In *Proceedings of the 2nd international conference on generative programming and component engineering*, pages 169–188. Springer-Verlag New York, Inc., 2003. ISBN 3-540-20102-5.

Clarke, Michael, G. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Proceedings of Middleware'01, IFIP/ACM International Conference on Distributed Systems Platforms*, number 2218 in LNCS. Springer-Verlag, Nov. 2001.

Clark, Lawrence. A linguistic contribution to goto-less programming. *Commun. ACM*, 27(4):349–350, 1984. ISSN 0001-0782.

Coady, Yvonne and G. Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 50–59. ACM Press, 2003. ISBN 1-58113-660-9.

Colyer, Adrian and A. Clement. Large-scale AOSD for middleware. In *Proceedings of the 3rd international conference on Aspect-Oriented Software Development (AOSD'04)*, pages 56–65. ACM Press, 2004. ISBN 1-58113-842-3.

Colyer, Adrian, A. Rashid, and G. Blair. On the separation of concerns in program families. Technical Report COMP-001-2004, Lancaster University, 2004.

Costa, Fabio M., H. A. Duran, N. Parlavantzas, K. B. Saikoski, G. S. Blair, and G. Coulson. The role of reflective middleware in supporting the engineering of dynamic applications. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 79–98. Springer-Verlag, 2000. ISBN 3-540-67761-5.

Coulson, Geoff, G. S. Blair, M. Clarke, and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, April 2002. ISSN 1432-0452.

Czarnecki, Krzysztof and U. W. Eisenecke. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000. ISBN 0-201-30977-7.

Dai, Renshou. A technical white paper on Sage's PSQM test, Aug. 7 2000. URL http://www.sageinst.com/downloads/925/psqmwp8_00.pdf Last accessed: Jul. 31 2005.

Demers, F. N. and J. Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *Workshop on Reflection and Meta-Level Architectures and their Applications in AI, IJCAI'95*, pages 29–38, 1995.

Dijkstra, Edsger W. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968. ISSN 0001-0782.

Dittrich, Klaus R., S. Gatziu, and A. Geppert. The active database management system manifesto: A rulebase of ADBMS features. In Sellis, T., editor, *Proceedings of the 2nd International Workshop on Rules in Database Systems*, volume 985 of *LNCS*, pages 3–20, Athens, Greece, 1995. Springer.

Duzan, Gary, J. Loyall, R. Schantz, R. Shapiro, and J. Zinky. Building adaptive distributed applications with middleware and aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 66–73, Lancaster, UK, Mar. 2004. ACM Press. ISBN 1-58113-842-3.

Ecklund, Denise J., V. Goebel, T. Plagemann, and E. F. Ecklund. Dynamic end-to-end QoS management middleware for distributed multimedia systems. *Multimedia Systems*, 8(5):431–442, Dec 2002.

EJB 2.1. Enterprise JavaBeans™Specification 2.1, November 2003. Sun Microsystems.

Elrad, Tzilla, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38, 2001. ISSN 0001-0782.

Fenton, Norman E. and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thomson Computer Press, 2nd edition, 1997. ISBN 1-85032-275-9.

Fitzpatrick, Tom and G. Blair. A software architecture for adaptive distributed multimedia applications. *IEEE Proceedings – Software*, 145(5):163–171, Oct. 1998.

Frankel, David S. *Model Driven Architecture™: Applying MDA to Enterprise Computing*. Wiley Publishing, 2003. ISBN 0-401-31920-1.

Friedman, Daniel P. and M. Wand. Reification: Reflection without metaphysics. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 348–355. ACM Press, 1984. ISBN 0-89791-142-3.

Gamma, Erich, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995. ISBN 0-201-63361-2.

Garcia, Alessandro, U. Kulesza, C. Sant'Anna, C. Lucena, E. Figueiredo, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 3–14. ACM Press, 2005. ISBN 1-59593-043-4.

Garlan, David, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37 (10):46–54, Oct. 2004.

Göbel, Steffen, C. Pohl, R. Aigner, M. P. S. Röttger, and S. Zschaler. The COMQUAD component container architecture. In *Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, pages 315–318, Oslo, Norway, Jun. 2004a. IEEE.

Göbel, Steffen, C. Pohl, S. Röttger, and S. Zschaler. The COMQUAD component model – enabling dynamic selection of implementations by weaving non-functional aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 74–82, Lancaster, UK, Mar. 2004b. ACM Press. ISBN 1-58113-842-3.

Golm, Michael and J. Kleinöder. MetaXa and the future of reflection. In *Proceedings of OOPSLA Workshop on Reflective Programming in C++ and Java*, Oct. 1998.

Grace, Paul, G. S. Blair, and S. Samuel. ReMMoC: A reflective middleware to support mobile client interoperability. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'03)*, number 2888 in LNCS. Springer-Verlag, 2003.

Haible, Bruno and T. Papanikolaou. Fast multiprecision evaluation of series of rational numbers. *Lecture Notes in Computer Science*, 1423:338–350, 1998. ISSN 0302-9743.

Hallsteinsen, Svein, J. Floch, and E. Stav. A middleware centric approach to building self-adapting systems. In *4th International Workshop on Software Engineering and Middleware (SEM'04)*, volume 3437 of *LNCS*, pages 107–122. Springer, Mar. 2005.

Hannemann, Jan and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of OOPSLA 2002*, pages 161–173. ACM Press, 2002. ISBN 1-58113-471-1.

Harrison, William, H. Ossher, S. M. S. Jr., and P. Tarr. Concern modeling in the concern manipulation environment. Research Report RC2334, IBM, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, Sep. 2004.

Harrison, William and H. Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the 8th annual conference on Object-oriented programming systems, languages, and applications (OOPSLA'93)*, pages 411–428. ACM Press, 1993. ISBN 0-89791-587-9.

IEEE 1471. IEEE recommended practice for architectural description of software-intensive systems, 2000.

ISO 13236. Information technology – Quality of Service: Framework, Oct. 2003. ISO document ISO/IEC JTC1/SC6 13236:1998.

ISO CD15935. CD 15935 information technology: Open Distributed Processing – reference model – Quality of Service, Oct. 1998. ISO document ISO/IEC JTC1/SC7 N1996.

ITU-T P.800. Methods for subjective determination of transmission quality, 1996. ITU-T Recommendation Series P.800: Telephone transmission quality.

Kiczales, Gregor, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991. ISBN 0-262-11158-6.

Kiczales, Gregor, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with ASPECTJ. *Communications of the ACM*, 44(10):59–65, 2001. ISSN 0001-0782.

Kiczales, Gregor, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In Aksit, Mehmet and S. Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming*, number 1241 in LNCS, pages 220–242. Springer-Verlag, June 1997. ISBN 3-540-63089-9.

Kiczales, Gregor. Beyond the black box: Open implementation. *IEEE Software*, 13(1): 8,10–11, January 1996.

Kon, Fabio, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Commun. ACM*, 45(6):33–38, 2002. ISSN 0001-0782.

Kon, Fabio, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. 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.

Leclercq, Matthieu, V. Quéma, and J.-B. Stefani. DREAM: A component framework for the construction of resource-aware, reconfigurable MOMs. In *Proceedings of the 3rd workshop on Adaptive and Reflective Middleware*, pages 250–255. ACM Press, 2004. ISBN 1-58113-949-7.

Ledoux, Thomas. OpenCorba: A reflective open broker. In *Proceedings of the 2nd intl. conference on Meta-Level Architectures and Reflection (Reflection'99)*, number 1616 in LNCS, pages 197–214. Springer-Verlag, Jul. 1999.

Lieberherr, Karl J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

Lieberherr, Karl J. Controlling the complexity of software designs. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 2–11. IEEE Computer Society, 2004. ISBN 0-7695-2163-0.

Liskov, Barbara. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM Press. ISBN 0-89791-266-7.

Lopes, Cristina Videira and S. K. Bajracharya. An analysis of modularity in aspect oriented design. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 15–26. ACM Press, 2005. ISBN 1-59593-043-4.

Loyall, Joseph, R. E. Schantz, J. A. Zinky, and D. E. Bakken. Specifying and Measuring Quality of Service in Distributed Object Systems. In *Proceedings of the first International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 1998)*, pages 43–52, 1998.

Maeda, Chris, A. Lee, G. Murphy, and G. Kiczales. Open implementation analysis and design. In *Proceedings of the 1997 symposium on Software reusability*, pages 44–52. ACM Press, 1997. ISBN 0-89791-945-9.

Maes, Pattie. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA'87)*, pages 147–155. ACM Press, 1987. ISBN 0-89791-247-0.

Malenfant, J., M. Jacques, and F. N. Demers. A tutorial on behavioral reflection and its implementation. In *Proceedings of Reflection'96*, Apr. 1996.

McKinley, Philip K., M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, Jul. 2004.

Narasimhan, Priya, L. E. Moser, and P. M. Melliar-Smith. Using interceptors to enhance CORBA. *IEEE Computer*, 32(7):62–68, Jul. 1999.

Nordberg, Martin E. Aspect-oriented dependency inversion. In *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*. ACM Press, 2001.

orbos-02-06-57. CORBA/IIOP Specification 3.0.2, 2002. OMG document orbos-02-06-57, Ch. 21 Portable Interceptors.

Ossher, Harold and P. Tarr. Hyper/J: Multi-dimensional separation of concerns for Java. In *Proceedings of the 22nd international conference on Software engineering*, pages 734–737. ACM Press, 2000. ISBN 1-58113-206-9.

Ossher, Harold and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Commun. ACM*, 44(10):43–50, 2001. ISSN 0001-0782.

Papapetrou, Odysseas and G. A. Papadopoulos. Aspect oriented programming for a component-based real life application: a case study. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1554–1558. ACM Press, 2004. ISBN 1-58113-812-1.

Parnas, D. L., P. C. Clements, and D. M. Weiss. Information distribution aspects of design methodology. In *Proceedings of IFIP Congress 71*, volume 1, pages 339–344. North-Holland Publishing Co, Amsterdam, Netherlands, 1971. ISBN 0-7204-2063-6.

Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972. ISSN 0001-0782.

Parnas, D. L. Why software jewels are rare. *Computer*, 29(2):57–60, 1996. ISSN 0018-9162.

Poladian, Vahe, J. P. Sousa, D. Garlan, and M. Shaw. Dynamic configuration of resource-aware services. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 604–613. IEEE Computer Society, May 2004.

ptc/2004-09-01. UML™profile for modelling Quality of Service and fault tolerance characteristics and mechanisms, 2004. OMG document ptc/2004-09-01 final adopted specification.

Rajkumar, Ragunathan, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for QoS management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 298–307, Dec 1997.

Reenskaug, Trygve, P. Wold, and O. A. Lehne. *The OOram Software Engineering Method*. Prentice Hall, 1996. ISBN 0-13-452930-8.

RFC 2205. Resource reservation protocol (RSVP) – version 1 functional specification, Sep. 1997. IETF.

RFC 3550. RTP: A transport protocol for real-time applications, Jul. 2003. IETF.

Röttger, Simone and S. Zschaler. CQML+: Enhancements to CQML. In *Proceedings of the QoS in CBSE03 workshop*, 2003.

Saltzer, J. H., D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984. ISSN 0734-2071.

Schantz, Richard, J. Loyall, M. Atighetchi, and P. Pal. Packaging quality of service control behaviors for reuse. In *Proceedings of the fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, pages 375–385, 2002.

Schmidt, Douglas C. Middleware for real-time and embedded systems. *Commun. ACM*, 45(6):43–48, 2002. ISSN 0001-0782.

Smith, Brian Cantwell. *Procedural Reflection in Programming Languages*. PhD thesis, MIT, 1982.

Smith, Brian Cantwell. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35. ACM Press, Jan. 1984. ISBN 0-89791-125-3.

Sommerville, Ian. *Software Engineering*. Addison Wesley, 5th edition, 1995. ISBN 0-201-42765-6.

Staehli, Richard, F. Eliassen, and S. Amundsen. Designing adaptive middleware for reuse. In *Middleware 2004 Companion, 3rd Workshop on Reflective and Adaptive Middleware*, 2004.

Staehli, Richard and F. Eliassen. QuA: A QoS-aware component architecture. Technical Report Simula 2002-12, Simula Research Laboratory, 2002.

Staehli, Richard and F. Eliassen. Compositional quality of service semantics. In *SAVCBS'04, Workshop at ACM SIGSOFT 2004/FSE-12*, Oct.31–Nov.1 2004.

Standish, Thomas A. An essay on software reuse. *IEEE Transactions on Software Engineering*, SE-10(5):494–497, Sept. 1984.

Szyperski, Clemens, S. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 2nd edition, 2002. ISBN 0-201-74572-0.

Tanenbaum, Andrew S. *Computer Networks*. Prentice-Hall, 4th edition, 2003. ISBN 0-13-066102-3.

Tarr, Peri, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. IEEE Computer Society Press, 1999. ISBN 1-58113-074-0.

Vanderperren, Wim, D. Suvée, B. Verheecke, M. A. Cibrán, and V. Jonckers. Adaptive programming in JAsCo. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 75–86. ACM Press, 2005. ISBN 1-59593-043-4.

Weiser, Mark. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981. ISBN 0-89791-146-6.

Welch, Ian and R. J. Stroud. Kava - using byte-code rewriting to add behavioral reflection to Java. In *Proceedings of COOTS 2001, USENIX Conference on Object-Oriented Technologies and Systems*, pages 119–130, Feb. 2001.

Wergeland, Øyvind Matheson. Service planning in a QoS-aware component architecture. Master's thesis, University of Oslo, 2005. Work in progress.

Yoder, Joseph W. and R. E. Johnson. The Adaptive Object-Model architectural style. In *Proceedings of WICSA3*, pages 3–27. Kluwer B.V., 2002. ISBN 1-4020-7176-0.

Zhang, Charles and H.-A. Jacobsen. Refactoring middleware with aspects. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1058–1073, Nov. 2003.

Zhang, Charles and H.-A. Jacobsen. Resolving feature convolution in middleware systems. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'04)*, pages 188–205. ACM Press, 2004. ISBN 1-58113-831-9.