



UNIVERSITÀ POLITECNICA
DELLE MARCHE

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA
INFORMATICA E DELL'AUTOMAZIONE

MetaSound: scalable architecture for multimedia metadata-processing

TESI DI LAUREA DI:
Michele Catasta

RELATORE:
Prof. Francesco Piazza

CORRELATORE:
Prof. Paolo Puliti

ANNO ACCADEMICO 2004/2005

Sviluppata da Michele Catasta (mcatasta@acm.org)
presso il Dipartimento di Elettronica, Intelligenza
Artificiale e Telecomunicazioni (<http://www.deit.univpm.it/>),
Facoltà di Ingegneria, Università Politecnica delle Marche.
Terminata nel mese di luglio 2005.

Si vuole qui ringraziare:

- il *Prof. Francesco Piazza* — per la preziosa ed insostituibile guida fornita durante tutto l'arco di svolgimento della tesi
- il *Prof. Paolo Puliti* — per essere stato da subito favorevole all'utilizzo di un Distributed Object System in Java, fulcro dell'architettura realizzata
- l'*Ing. Giovanni Tummarello* e l'*Ing. Christian Morbidoni* — per le frequenti e stimolanti discussioni sui loro argomenti di ricerca
- il *team di sviluppo della libreria ProActive* — per il supporto offerto tramite la mailing-list
- l'*Ing. Giorgio Biagetti* — per la disponibilità dimostrata durante lo sviluppo del template L^AT_EX utilizzato in questa tesi
- *Adelmo De Santis* — per aver contribuito attivamente all'atmosfera produttiva, e allo stesso tempo amichevole, del DEIT



This work is licensed under the Creative Commons Attribution License.

To view a copy of this license, visit <http://creativecommons.org/licenses/by/2.5/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

*“Creati un esercito di servi perfetti,
capaci di eseguire per tuo conto compiti ingrati,
e sarai il padrone del mondo.”* — N. Tartaglia,
(Il general trattato di numeri et misure)

*“The distant king of birds, the Simurgh, drops one of his splendid
feathers somewhere in the middle of China; on learning of this,
the other birds, tired of their present anarchy, decide to seek him.
...Thirty, made pure by their sufferings, reach the great peak of
the Simurgh. At last they behold him; they realize that they are the
Simurgh, and that the Simurgh is each of them and all of them.”* —
J. L. Borges, (The Book of Imaginary Beings)

Introduction

Metadata (literally “data about data”) has been recognized as fundamentally important since much earlier than the inception of the World Wide Web. Traditional metadata technologies as library indexes, keyword searches, etc. do not however scale nor fit the WWW scenario where almost anyone can freely publish huge amount of material. More than sheer scalability, fundamentally new techniques are needed when the content to be indexed and searched is of multimedia nature.

Today, multimedia metadata is often limited to simple, manually-created, “annotations” such as ID3 Tags. While these prove nevertheless useful to software and hardware audio players, very interesting scenarios become possible with machine generated metadata offering various degrees of “automated intelligence”.

However, these formats have already given birth to some new challenging problems. A brief example should sketch the entity of them. Gracenote CDDB® is the largest online database of music information in the world, containing about 48,000,000 metadata records. Given an advanced multimedia metadata format characterized by an average size of 3 MB, storing informations about all songs catalogued in CDDB® would require 144 TB. Much more, guessing an average extraction time of 3 minutes¹, producing metadata for each song would require (rounding down) 273 years of CPU time.

Albeit technological boundaries advance at quick pace, new standards

¹average data about size and extraction time have been extrapolated after some tests on MPEG-7 (using MPEG7AudioEnc library: <http://www.iient.rwth-aachen.de/team/crysandt/mpeg7audioenc/>)

cannot diffuse quickly if no solutions for computational and storage burden are available.

This thesis describes a scalable approach to multimedia metadata handling, addressing its attention on the problem previously mentioned. A distributed architecture, *MetaSound*, has been designed and realized to exploit advantages of parallel computation and redundant storage. Scalability represents its key feature, making it well-suited to loosely-coupled parallel systems (e.g. clusters). Thus, whenever a performance improvement on metadata processing is needed, architecture scalability makes it possible simply employing a faster cluster.

Contents of this thesis are organized in the following way:

- **Chapter 1** sketches current state of multimedia metadata standards
- **Chapter 2** depicts distributed systems capabilities and Distributed Object Systems advantages
- **Chapter 3** shows state of the art about software architectures and design patterns in Object-Oriented programming
- **Chapter 4** describes *MetaSound* architecture, from its prerequisites to its pragmatistical aspects
- **Chapter 5** digs into *MetaSound* internals, using UML class diagrams
- **Chapter 6** draws conclusions and some proposals about *MetaSound* evolution
- **Appendix A** gives a quick overview on UML, a modeling language used to build formal documentation of *MetaSound*
- **Appendix B** presents tools and licenses that influenced *MetaSound* developing

Contents

Introduction	iv
1 Multimedia Metadata	1
1.1 Overview	1
1.2 MPEG-7	2
1.3 Dublin Core	3
2 Distributed Paradigm	5
2.1 Distributed Systems	5
2.1.1 Characteristics	5
2.1.2 Advantages	7
2.1.3 Disadvantages	9
2.2 Distributed Object Systems	10
2.3 A study case: ProActive	13
3 Software Design	15
3.1 Software Architectures	15
3.1.1 Three-tier model	16
3.2 Design Patterns	17
3.2.1 Uses	17
3.2.2 Classification	17
3.2.3 MetaSound & patterns	18
3.3 Active Object Pattern	19
3.3.1 Structure	20
3.3.2 Dynamics	22

4	MetaSound Architecture	24
4.1	Prerequisites	24
4.2	Design	24
4.3	API	25
4.4	Pragmatic aspects	25
4.4.1	Load balancing	25
4.4.2	Logging	27
4.4.3	Storage Index	27
4.4.4	Configuration Files	27
5	MetaSound Internals	29
6	Conclusions	37
A	UML	39
A.1	History	39
A.2	Modeling aspects	40
A.2.1	Use Case Diagram	40
A.2.2	Component Diagram	41
A.2.3	Class Diagram	41
B	Tools and Licenses	43
B.1	Tools	45
B.1.1	Eclipse	45
B.1.2	SourceForge.net	47
B.1.3	L ^A T _E X	47
B.1.4	Doxygen	49
B.2	Licenses	49
B.2.1	MIT License	49
B.2.2	Creative Commons	50

Chapter 1

Multimedia Metadata

There are literally hundreds of metadata initiatives. This chapter, however, will present only the most relevant in the multimedia field: ISO's Multimedia Content Description Interface (MPEG-7) and Dublin Core Metadata Initiative.

These efforts are rather different in scope, and thus cannot be directly compared in full. Generally speaking, only MPEG-7 takes multimedia directly into consideration.

1.1 Overview

Metadata-related issues touch the core of all information sciences and have been influenced by many communities: in particular, the digital library (**DL**) community, and the part of the AI community that interprets, manipulates or generates audio-visual media (**MM-AI**).

The differences between these communities could be so summarized:

- The *DL community* is first of all interested in cataloguing and retrieving information in large document collections. A notable artifact of such community is the Dublin Core Metadata Initiative, which supports retrieving of documents by standardizing 15 commonly agreed metadata elements.

- The *MM-AI community* is mostly interested with “low level features” (e.g. the spectral characteristic of a sound to determine the kind of instrument). This is directly reflected and well supported in the MPEG-7 metadata initiative.

1.2 MPEG-7

Recognizing the opportunities in multimedia metadata standardization, the Motion Picture Expert Group (MPEG) has, since the second half of the 90s, worked on **MPEG-7**, resulting in its publication as an ISO standard in 2001. Formally named *Multimedia Content Description Interface*, MPEG-7 provides a rich set of tools for multimedia annotations aimed at production/consumption of both humans and machines.

Tools offered by MPEG-7 are audiovisual *Description Tools* in form of metadata elements and their structure and relationships. These serve to create “Descriptions”, which will enable applications to efficient access (search, filtering and browsing) to multimedia content. This is a challenging task given the broad spectrum of requirements and targeted multimedia applications, and the broad number of audiovisual features of importance in such context.

MPEG-7 descriptions do not, in general, depend on the ways the described content is coded or stored. It is possible to create an MPEG-7 description of an analogue movie or of a picture that is printed on paper, in the same way as of digitized content. At the same time MPEG-7 doesn’t mandate extraction algorithms, but merely suggests them for the simpler descriptors, such as the Audio Low Level specifications.

Audiovisual data content that has MPEG-7 descriptions associated with it, may include:

- Information describing the creation and production processes of the content (director, title, short feature movie).
- Information related to the usage of the content (copyright pointers, usage history, broadcast schedule).

- Information of the storage features of the content (storage format, encoding).
- Structural information on spatial, temporal or spatio-temporal components of the content (scene cuts, segmentation in regions, region motion tracking).
- Information about low level features in the content (colors, textures, sound timbres, melody description).
- Conceptual information of the reality captured by the content (objects and events, interactions among objects).
- Information about how to browse the content in an efficient way (summaries, variations, spatial and frequency subbands, ...).
- Information about collections of objects.
- Information about the interaction of the user with the content (user preferences, usage history).

MPEG-7 uses XML technologies, therefore facilitating interoperability. XML schema is used as a base for the **DDL** (Description Definition Language) which is then used to express the syntax of the MPEG-7 descriptors. Descriptors might be thought of as “base classes” or even “literals”, description schemata as “classes” that are then instantiated when encoding context.

1.3 Dublin Core

The Dublin Core Metadata Initiative (**DCMI**) is an organization dedicated to promoting the widespread adoption of interoperable metadata standards and developing specialized metadata vocabularies for describing resources that enable more intelligent information discovery systems.

The most notable achievement of the DCMI is the *Dublin Core Metadata Element Set*, also published as RFC and ISO standard.

In the published versions it consists of the following 15 terms:

- **Title:** a name given to the resource
- **Creator:** an entity primarily responsible for making the content of the resource
- **Subject:** a topic of the content of the resource
- **Description:** an account of the content of the resource
- **Publisher:** an entity responsible for making the resource available
- **Contributor:** an entity responsible for making contributions to the content of the resource
- **Date:** a date of an event in the lifecycle of the resource
- **Resource:** the nature or genre of the content of the resource
- **Format:** the physical or digital manifestation of the resource
- **Identifier Definition:** an unambiguous reference to the resource within a given context
- **Source:** a reference to a resource from which the present resource is derived
- **Language:** a language of the intellectual content of the resource
- **Relation:** a reference to a related resource
- **Coverage:** the extent or scope of the content of the resource
- **Rights:** informations about rights held in and over the resource

As evident from the above list, the DCMI element set is rather high level and underspecified, but it can well serve many needs in multimedia metadata scenario.

Chapter 2

Distributed Paradigm

This chapter shows peculiar features of distributed systems and Distributed Object Systems. They represent two mandatory knowledge field to understand *MetaSound* architecture.

The third section is dedicated to a study case: ProActive library, the DOS on which is based *MetaSound* .

2.1 Distributed Systems

A distributed system is a collection of autonomous computers linked by a network and equipped with distributed system software. Obviously, it is the opposite of a centralized system, which consists of a single computer with one or multiple powerful CPUs processing all incoming requests.

2.1.1 Characteristics

A distributed system has six important characteristics: (1) *resource sharing*, (2) *openness*, (3) *concurrency*, (4) *scalability*, (5) *fault tolerance* and (6) *transparency*. These characteristics are not automatic consequences of distribution. Instead, they are acquired as a result of a careful design and implementation.

Resource Sharing

Resources provided by a computer which is a member of a distributed system can be shared by clients and other members of the system via a network. In order to achieve effective sharing, each resource must be managed by a software that provides interfaces which enables the resource to be manipulated by clients.

Openness

Openness determines whether the system is extendible in various ways. This characteristic is measured mainly by the degree to which new resource sharing services can be incorporated without disruption or duplication of existing services.

Concurrency

Concurrency is the ability to process multiple tasks at the same time. A distributed system comprises multiple computers, each having one or multiple processors. The existence of multiple processors in the system can be exploited to perform multiple tasks at the same time (crucial ability to improve the overall performance).

Scalability

A system or application software is scalable if it does not need to change when the scale of the system increases. Scalability is important since the amount of requests processed by a distributed system tends to grow, rather than decrease. In order to handle the increase, additional hardware and/or software usually needed.

Fault Tolerance

Fault tolerance is achieved through an appropriately handling of errors. A system with good fault tolerance mechanisms has a high degree of

availability. A distributed system's availability is a measure of the proportion of time that the system is available for use. A better fault tolerance increases availability. Fault tolerance is achieved by deploying two approaches: *hardware redundancy* and *software recovery*.

Transparency

Transparency is the concealment of the separation of components in a distributed system from the user and the application programmer such that the system is perceived as a whole rather than as collection of independent components. As a distributed system is separated by nature, transparency is needed to hide all unnecessary details regarding this separation from users.

2.1.2 Advantages

The trend of distributed systems is motivated by the potential benefits that they could yield. These benefits are:

Share-Ability

Share-Ability allows the comprising systems to use each other's resources. This sharing takes place on a computer network connected to each system, using a common protocol that governs communications among the systems. Both the network and the protocol are respectively the common communication medium and the common protocol that facilitate sharing.

Expandability

Expandability permits new systems to be added as members of the overall system. It also influences the ability to determine what level of processing power the host machines must have. A distributed system might end up providing unused resources on machines with under-capacity utilization.

Such waste of time and money is resolved by giving the freedom to add shared resources only when they are really needed.

Local Autonomy

Local autonomy permits each system to apply local policies, settings or access controls to its resources and services. This makes distributed systems ideal for organizations whose structure consists of independent entities located in different locations.

Improved Performances

Separate nature of distributed systems is helpful in performance field, thanks to techniques such as replication and load balancing. Moreover, the number of computers in a distributed system benefits the system, in terms of its processing power. This is because the combined processing power of multiple computers provides much more processing power than a centralized system. The limit of which a single computer can be installed with multiple CPUs prohibit an endless increase of its processing power.

Improved Reliability and Availability

Disruptions to a distributed system do not stop the system as a whole from providing its resources. Some resources may not be available, but others are still accessible. This is because these resources are spread across multiple computers where each resource is managed by one computer. If these resources are replicated, the disruption might cause only minimum impact on the system. This is because requests can be diverted to other copies of the target resources.

Potential Cost Reductions

Rather than paying more for a single CPU, the most cost-effective way of achieving a better price to performance ratio is to harness a large number of CPUs to process requests. Another potential cost reduction occurs

when a distributed system is used to handle request processing shared of multiple organizations. All of these organizations could contribute to the setup and maintenance costs. This reduces the per organization costs down compared to setting and maintaining the system independently.

2.1.3 Disadvantages

Beside these advantages a distributed system has the following disadvantages:

Network Reliance

Because all computers in a distributed system rely on a network to communicate to each other, problems on the network would disrupt activities in the system as a whole. This is true especially for physical problems such as broken network cables, routers, bridges, etc. The cost of setting up and maintaining the network could eventually outweigh any potential cost savings offered by distributed systems.

Complexities

A distributed system software is not easy to develop. It must be able to deal with errors that could occur from all computers that make up the distributed system. It must also be capable to manipulate resources of computers with a wide range of heterogeneities.

Security

A distributed system allows its computers to collaborate and share resources more easily. However, this convenience of access could be a problem if no proper security mechanism are put in place. Private resources would be exposed to a wider range of potential hackers, with unauthorized accesses launched from any computers connected to the system. In fact, a centralized system is usually more secure than a distributed system.

2.2 Distributed Object Systems

A distributed system software enables computers to coordinate their activities and to share system resources. A well-developed distributed system software provides the illusion of a single and integrated environment although it is actually implemented by multiple computers in different locations. In other words, the software gives a *distribution transparency* to the systems.

Distributed computing, when applied properly, can improve:

- collaboration through connectivity and internetworking;
- performance through parallel processing;
- reliability and availability through replication;
- scalability and portability through modularity;
- extensibility through dynamic configuration and reconfiguration;
- cost effectiveness through resource sharing and open systems.

However, developing distributed applications whose components collaborate efficiently, reliably, transparently and scalably is a complex task. Much of this complexity arises from limitations with conventional tools and techniques used to develop distributed application software. Many standard network programming mechanisms¹ and reusable components libraries² lack type-safe, portable and extensible interfaces. Moreover, many distributed applications are developed using functional decomposition techniques that are considered deprecated, compared to brand new techniques in software engineering (moreover after OO revolution). Searching for the root, these limitations can be imputed to programming paradigms of some widespread languages used to develop the majority of OS networking facilities and libraries.

¹such as BSD sockets and Windows NT named pipes

²such as Sun RPC

Albeit OS developing cannot change yet its underpinnings for performance and historical motivations, application layer is deeply influenced by object-oriented technologies. In distributed computing domain, OOP provides with many of the same benefits (encapsulation, reuse, portability, extensibility, etc.) as it does for non-distributed computing.

In fact, it is often more natural to utilize object-oriented techniques in the domain of the distributed computing than it is for non-distributed computing. This is due to the inherently decentralized nature of distributed computing. In conventional non-distributed applications, there is often a temptation to sacrifice abstraction and modularity for a perceived increase in performance. For example, many programmers use global variables or access fields in structures directly to avoid the overhead of passing parameters and calling functions, respectively.

In distributed computing, however, performance optimizations based on direct access to global resources are extremely difficult to develop and scale. Even if efforts like NUMA (Non Uniform Access Memory) have been born to provide better memory scalability than in SMP, they represent trade-offs between performance/low latency and agile developing.

Therefore, most distributed applications interoperate by passing messages. There are many variations on this message passing theme (e.g.: RPC, remote event queues, bytestream communication, etc.). However, it doesn't require too much of intuition to observe that message passing in distributed computing is very similar to method invocation on an object in OOP. With this observation in mind, several of the key features of DOC can be discussed:

- *Providing many of the same enhancements to procedural RPC toolkits that object-oriented languages provide to conventional procedural programming languages:* These enhancements include encapsulation, interface inheritance, parameterized types, and object-based exception handling. Encapsulation promotes the separation of interface from implementation. This separation is crucial for developing highly extensible architectures that decouple reusable application-independent mechanisms from application-specific poli-

cies. Interface inheritance and parameterized types promote reuse and emphasize commonality in a design. Object-based exception handling often simplifies program logic by decoupling error-handling code from normal application processing.

- *Enabling internetworking between applications at higher lever of abstraction:* Distributed applications have traditionally been developed using relatively low-level mechanisms. Common mechanisms include the TCP/IP protocol, the socket transport layer programming interface, and the select event demultiplexing system call. These low-level mechanisms provide applications with reliable, untyped, point-to-point bytestream services. In general, these services are optimized for performance, rather than ease of programming, reliability, portability, flexibility, or extensibility. A primary objective of DOC is to enable developers to program distributed applications using familiar techniques such as method calls on objects. Ideally, accessing the services of a remote object should be as simple as calling a method on that object. A surprisingly large number of fairly complicated components must be developed to support remote method invocation on objects transparently. These components include directory name servers, object request brokers (ORBs), interface definition language compilers, object location and startup facilities, multi-threading facilities, and security mechanisms.
- *Providing a foundation for building higher-level mechanisms that facilitate the collaboration among services in distributed applications:* Supporting transparent remote object method invocation is only the first step in the long journey into the realm of distributed object computing. An increasing number of distributed applications require more sophisticated collaboration mechanisms. These mechanisms include common object services such as global naming, event filtering, object migration, reliable group communication, transactional messaging and *quality of service* facilities.

2.3 A study case: ProActive

ProActive is a GRID Java library for *parallel, distributed, and concurrent computing*, also featuring mobility and security in a uniform framework. With a reduced set of simple primitives, ProActive provides a comprehensive API allowing to simplify the programming of applications that are distributed on Local Area Network (LAN), on cluster of workstations, or on Internet Grids.

The library is based on an Active Object pattern that is a uniform way to encapsulate:

- a **remotely** accessible object
- a **thread** as an asynchronous activity
- an **actor** with its own script
- a **server** of incoming requests
- a **mobile** and potentially secure entity

On top of those basic features, ProActive also includes advanced aspects, such as:

- Typed **Group** Communications
- **OO SPMD**
- Distributed and Hierarchical **Components**
- **Security**
- a **P2P** infrastructure
- a Graphical User Interface **IC2D**
- a powerful **deployment** model based on XML descriptors

ProActive is only made of standard Java classes, and requires no changes to the Java Virtual Machine, no preprocessing or compiler modification; programmers write standard Java code. Based on a simple Meta-Object Protocol, the library is itself extensible, making the system open for adaptations and optimizations. ProActive currently uses the RMI Java standard library as a portable transport layer.

ProActive features several optimizations improving performance. For instance, whenever two active objects are located within the same virtual machine, a direct communication is always achieved, without going through the network stack. This optimization is ensured even when the co-location occurs after a migration of one or both of the active objects.

ProActive has an architecture that allows the library to interoperate with various official or de facto *standards*:

- Web Service Exportation
- HTTP Transport
- ssh, rsh, RMI/ssh Tunneling
- Globus GT2 and GT3, sshGSI
- LSF, PBS, Sun Grid Engine

Chapter 3

Software Design

This chapter introduces some best practices, fetched from modern software engineering.

An opening section about software architectures shows the pool of choice from which *MetaSound* design has been influenced. Then, being all the architecture based on Object Oriented Programming, follows a section about design patterns, with detailed explanations about patterns used in ProActive and *MetaSound* code. The last section, about “Active Object Pattern”, is fundamental to understand ProActive basics.

3.1 Software Architectures

Software architecture is a coherent set of abstract patterns guiding the design of each aspect of a larger software system.

Software architecture underlies the practice of building computer software. In the same way as a building architect sets the principles and goals of a building project as the basis for the draftsman’s plans, so too, a software architect sets out the software architecture as a basis for actual system design specifications, per the requirements of the client.

A software architect employs extensive knowledge of software theory and appropriate experience to conduct and manage the high-level design of a software product. The software architect develops concepts and

plans for software modularity, module interaction methods, user interface dialog style, interface methods with external systems, innovative design features, and high-level business object operations, logic, and flow.

Software architecture is a sketchy map of the system: it describes the coarse grain components (usually describes the computation) of the system. The connectors between these components describe the communication, which are explicit and pictured in a relatively detailed way. In the implementation phase, the coarse components are refined into “actual components”, e.g., classes and objects. In the object-oriented field, the connectors are usually implemented as interfaces.

There are many common ways of designing computer software modules and their communications, among them: *Client-Server*, *P2P system*, *Monolithic system*, *Software componentry*, *Three-tier model*, etc.

3.1.1 Three-tier model

Three-tier is a client-server architecture in which the user interface, functional process logic (“business rules”), data storage and data access are developed and maintained as independent *modules*, most often on separate platforms. The term “three-tier” or “three-layer”, as well as the concept of multi-tier architectures, seems to have originated within Rational Software.

Apart from the usual advantages of modular software with well defined interfaces, the three-tier architecture is intended to allow any of the three tiers to be upgraded or replaced independently as requirements or technology change. For example, a change of desktop operating system from Linux to Mac OS X would only affect the user interface code.

Typically, the user interface runs on a desktop PC or workstation and uses a standard graphical user interface, functional process logic may consist of one or more separate modules running on a workstation server or application server, and an RDBMS on a database server or mainframe contains the data storage logic. The middle tier may be multi-tiered itself (in which case the overall architecture is called an “n-tier architecture”).

3.2 Design Patterns

Design patterns are standard solutions to common problems in software design. Instead of focusing on how individual components work, design patterns take a systematic approach, which focuses on the patterns of interaction. Design patterns describe abstract systems of interaction between classes, objects, and communication flow.

3.2.1 Uses

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems.

Often, programmers only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

Patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them likely to perform better than ad-hoc designs.

3.2.2 Classification

Design patterns can be classified based on multiple criteria, the most common of which is the basic underlying problem they solve. According to this criterion, design patterns can be classified into various classes, some of which are:

- Fundamental patterns
- Creational patterns

- Structural patterns
- Behavioral patterns
- Concurrency patterns

3.2.3 MetaSound & patterns

Here follows a detailed description of design patterns that have influenced *MetaSound* developing.

Singleton

The singleton design pattern is designed to restrict instantiation of a class to one (or a few) objects. This is useful when exactly one object is needed to coordinate actions across the system. Sometimes it is generalized to systems that operate more efficiently when only one or a few objects exist.

The singleton pattern is implemented by creating a class with a method that creates a new instance of the object if one does not exist. If one does exist it returns a reference to the object that already exists. To make sure that the object cannot be instantiated any other way the constructor is made either private or protected.

The singleton pattern must be carefully constructed in multi-threaded applications. If two threads are to execute the creation method at the same time when a singleton does not yet exist, they both must check for an instance of the singleton and then only one should create the new one.

Factory Method

The Factory Method pattern is a creational pattern, therefore it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. Factory Method handles this problem by defining a separate method for creating the objects, which subclasses can then override to specify the type of product that will be created.

The main classes in the Factory Method pattern are the creator and the product. The creator needs to create instances of products, but the concrete type of product should not be hardcoded in the creator subclasses of creator should be able to specify subclasses of product to use.

To achieve this an abstract method (the factory method) is defined on the creator. This method is defined to return a product. Subclasses of creator can override this method to return instances of appropriate subclasses of product.

More generally, the term Factory Method is often used to refer to any method whose main purpose is to create objects.

Composite

Composite pattern decomposes objects into their parts, and reassemble them in whichever combination one needs with “has-a” relationships. That is to say, to break things into the largest parts that still allow reuse of the parts, and build objects of those.

The other definition is to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

3.3 Active Object Pattern

Active Object pattern decouples method execution from method invocation in order to simplify synchronized access to an object that resides in its own thread of control. The Active Object pattern allows one or more independent threads of execution to interleave their access to data modeled as a single object. A broad class of producer/consumer and reader/writer applications are well-suited to this model of concurrency.

This pattern is commonly used in distributed systems requiring multi-threaded servers. In addition, client applications, such as windowing systems and network browsers, employ active objects to simplify concurrent,

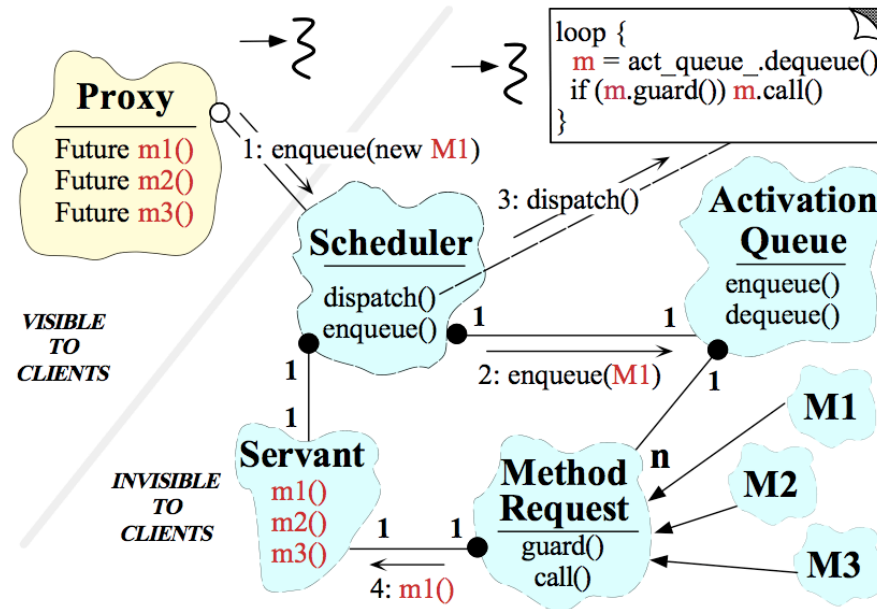


Figure 3.1: Active Object structure

asynchronous network operations.

3.3.1 Structure

There are six key participants in the Active Object pattern:

Proxy

A Proxy provides an interface that allows clients to invoke publically accessible methods on an Active Object using standard, strongly-typed programming language features, rather than passing loosely-typed messages between threads. When a client invokes a method defined by the Proxy, this triggers the construction and queuing of a Method Request object onto the Scheduler's Activation Queue, all of which occurs in the client's thread of control.

Method Request

A Method Request is used to pass context information about a specific method invocation on a Proxy, such as method parameters and code, from the Proxy to a Scheduler running in a separate thread. An abstract Method Request class defines an interface for executing methods of an Active Object. The interface also contains guard methods that can be used to determine when a Method Request's synchronization constraints are met. For every Active Object method offered by the Proxy that requires synchronized access in its Servant, the abstract Method Request class is subclassed to create a concrete Method Request class. Instances of these classes are created by the proxy when its methods are invoked and contain the specific context information necessary to execute these method invocations and return any results back to clients.

Activation Queue

An Activation Queue maintains a bounded buffer of pending Method Requests created by the Proxy. This queue keeps track of which Method Requests to execute. It also decouples the client thread from the servant thread so the two threads can run concurrently.

Scheduler

A Scheduler runs in a different thread than its clients, managing an Activation Queue of Method Requests that are pending execution. A Scheduler decides which Method Request to dequeue next and execute on the Servant that implements this method. This scheduling decision is based on various criteria, such as *ordering*, e.g., the order in which methods are inserted into the Activation Queue, and *synchronization constraints*, e.g., the fulfillment of certain properties or the occurrence of specific events, such as space becoming available for new elements in a bounded data structure. A Scheduler typically evaluates synchronization constraints by using method request guards.

Servant

A Servant defines the behavior and state that is being modeled as an Active Object. Servants implement the methods defined in the Proxy and the corresponding Method Requests. A Servant method is invoked when its corresponding Method Request is executed by a Scheduler; thus, Servants execute in the Schedulers thread of control. Servants may provide other methods used by Method Requests to implement their guards.

Future

A Future allows a client to obtain the results of method invocations after the Servant finishes executing the method. When a client invokes methods through a Proxy, a Future is returned immediately to the client. The Future reserves space for the invoked method to store its results. When a client wants to obtain these results, it can “rendezvous” with the Future, either blocking or polling until the results are computed and stored into the Future.

3.3.2 Dynamics

The following figure illustrates the three phases of collaborations in the Active Object pattern:

1. **Method Request construction and scheduling:** In this phase, the client invokes a method on the Proxy. This triggers the creation of a Method Request, which maintains the argument bindings to the method, as well as any other bindings required to execute the method and return its results. The Proxy then passes the Method Request to the Scheduler, which enqueues it on the Activation Queue. If the method is defined as a *two-way*, a binding to a Future is returned to the client that invoked the method. No Future is returned if a method is defined as a *oneway*, i.e., it has no return values.

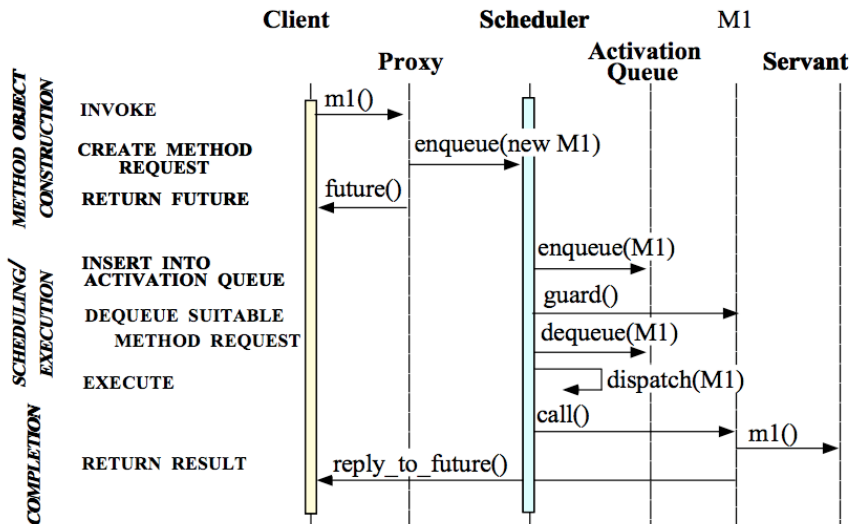


Figure 3.2: Three phases of collaboration (Active Object)

2. **Method Execution:** In this phase, the Scheduler runs continuously in a different thread than its clients. Within this thread, the Scheduler monitors the Activation Queue and determines which Method Request(s) have become runnable, e.g., when their synchronization constraints are met. When a Method Request becomes runnable, the Scheduler dequeues it, binds it to the Servant, and dispatches the appropriate method on the Servant. When this method is called, it can access/update the state of its Servant and create its result(s).
3. **Completion:** In the final phase, the results, if any, are stored in the Future and the Scheduler continues to monitor the Activation Queue for runnable Method Requests. After a two-way method completes, clients can retrieve its results by rendezvousing with the Future. In general, any clients that rendezvous with the Future can obtain its results. The Method Request and Future are deleted or garbage collected when they are no longer referenced.

Chapter 4

MetaSound Architecture

This chapter explains the design and implementation of *MetaSound* architecture.

4.1 Prerequisites

MetaSound prerequisites are quite obvious: provide a scalable architecture for multimedia metadata handling and storing. Previous chapters have shown the state of the art about distributed systems and programming paradigms to obtain performances and, at the same time, good software design. *MetaSound* main effort is the merging of all these best practices, to meet standards required by a similar distributed application.

4.2 Design

MetaSound is a three-tier scalable architecture based on a Distributed Object Systems: ProActive library. A Java-only approach has been chosen, instead of using CORBA, because every project that relies on *MetaSound* (i.e. MPEG7AudioDB) is based on Java. Thus, ProActive is the DOS of choice because of the plethora of additional features it has compared to Java RMI.

For a transparent distributed architecture like *MetaSound*, Java pro-

vides some advantages: native cross-platform support, full-stack set of technologies, etc. Obviously, these are the same reasons why ProActive team has chosen Java to build its GRID library.

Thus, *MetaSound* combines an effective software architecture with a bleeding-edge solution for what concerns distributed programming. Thanks to OOP capabilities of Java and use of design patterns, *MetaSound* expandability and modularity is assured.

Three-tiers of *MetaSound* are: client, server and storage. Client is the tier which accepts tasks from remote computers, storage is the tier that manages metadata memorization, and server is the “glue-tier”, thanks to its client and server manager components.

4.3 API

MetaSound API is straightforward: it is composed by 5 classes. Two of them are simple init and shutdown classes (safe shutdowns can be obtained only using API). The third class is a static method container, useful to try connecting on the given server. The remaining two classes are task wrappers, with support for automatic execution of a set of methods, automatic dispatching on client tier of both task and relative file, remote storing capabilities, ...

4.4 Pragmatic aspects

This section outlines practical aspects encountered during the development of *MetaSound*.

4.4.1 Load balancing

Load balancing is a technique to enhance resources, utilizing parallelism exploiting, and to cut response time through an appropriate distribution of the application.

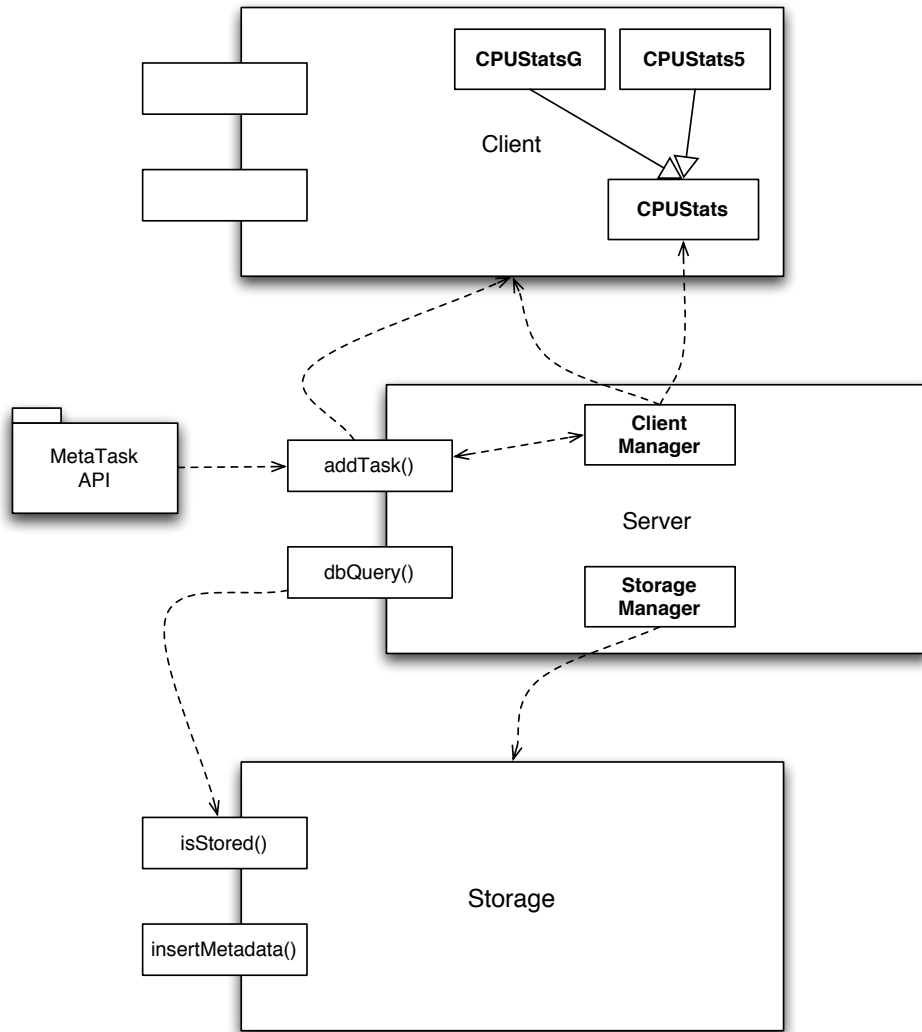


Figure 4.1: MetaSound components interaction

There exist two typical load balancing approaches: Static and Dynamic. Static load balancing is characterized by pre-execution task placement based on a prior knowledge of the application and target system characteristics. Dynamic load balancing can adapt to changes in target systems, according to the protocol provided for manage that changes.

MetaSound implements a Java CPU calibration tool for load balancing in distributed applications [1]. This facility, as sketched on the components interaction picture, is loaded on the client tier (CPUStats). Then, it is periodically probed by Client Manager (a server component).

Thanks to the load tool, *MetaSound* server tier, given CPU load data and an overload penalty policy, can dynamically schedule task dispatching in a transparent way to end user.

4.4.2 Logging

MetaSound use Apache Log4J facility to produce both console and file logging. Server and Storage tiers log on a rolling file, timestamping each entry. Anyway, console logging is common to all tiers. Log4J configurations, obviously, are not hardcoded, but can be changed for each tier on its specific log4j file.

4.4.3 Storage Index

Storage tier use SHA-1 digests as primary keys of its internal database. Index persistence is obtained using a file that mirrors constantly internal index. Its consistency is guaranteed also after program/computer unexpected hang.

4.4.4 Configuration Files

XML is used for configuration files, thanks to the human-readability of such format. Files are parsed using an Apache Xerces handler class opportunely extended and instanced as a singleton. Thus, during life time

of the tier, a cache immutable object containing configuration parameters is available.

Chapter 5

MetaSound Internals

This chapter digs into *MetaSound* internals using UML class diagrams.

First of all, an overview about package structure is given. To improve tier *decoupling*, interfaces are placed all together in a separate package. This solution makes possible to distribute “one-tier releases”, simply leaving inside core the given tier and the interfaces package.

Other diagrams are about: interfaces, tier collaborations, tier factories and *MetaSound* API. Collaboration diagrams show evidently the composite pattern used in client and server tier.

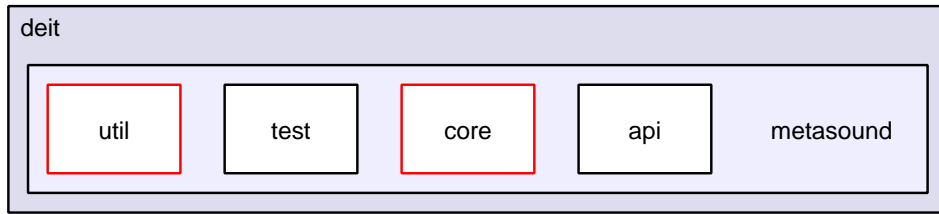


Figure 5.1: MetaSound package structure

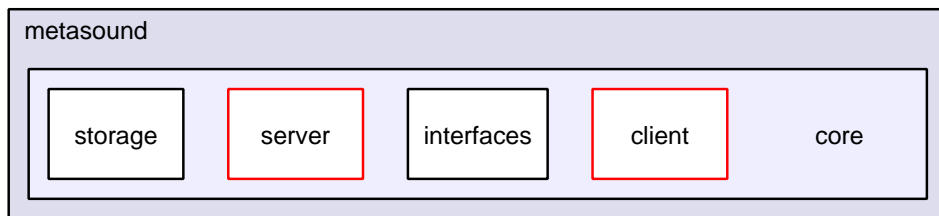


Figure 5.2: Core package structure

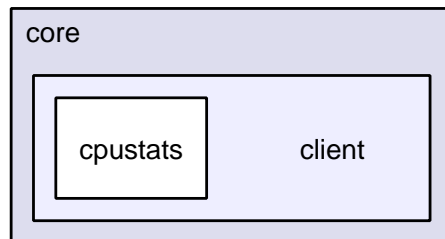


Figure 5.3: Client package structure

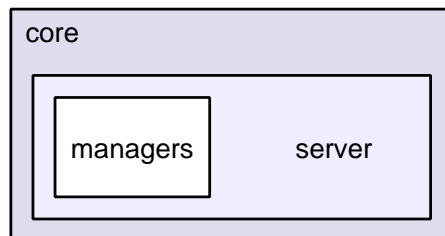


Figure 5.4: Server package structure

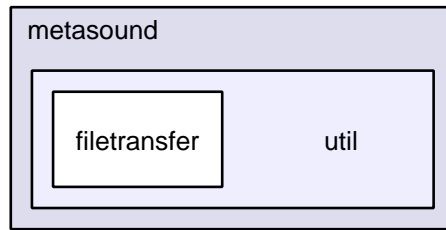


Figure 5.5: Util package structure

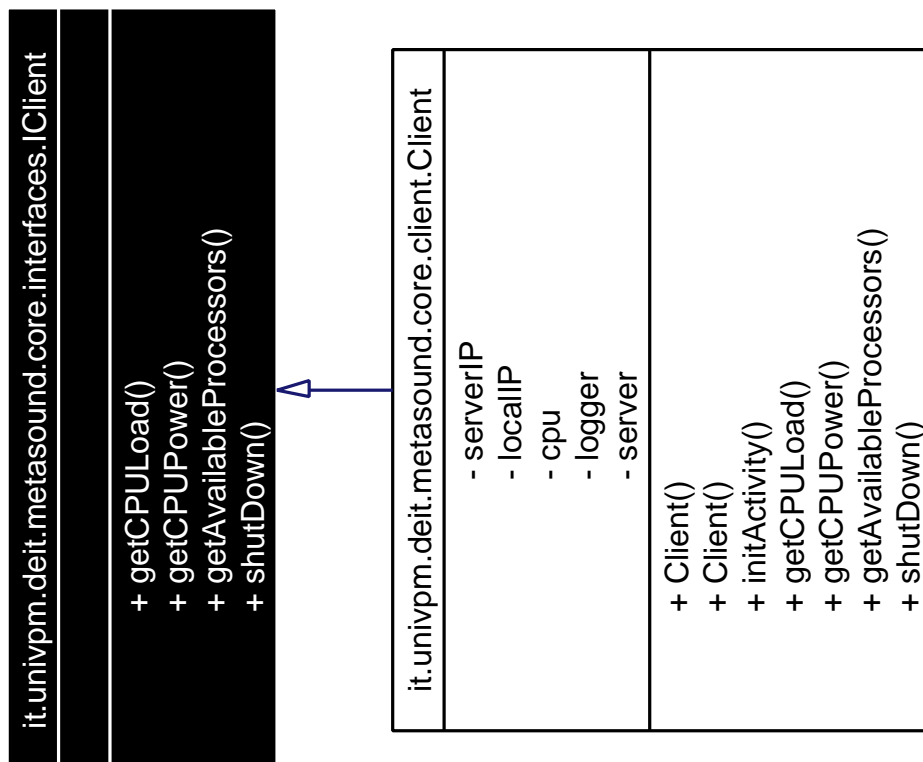


Figure 5.6: Client interface

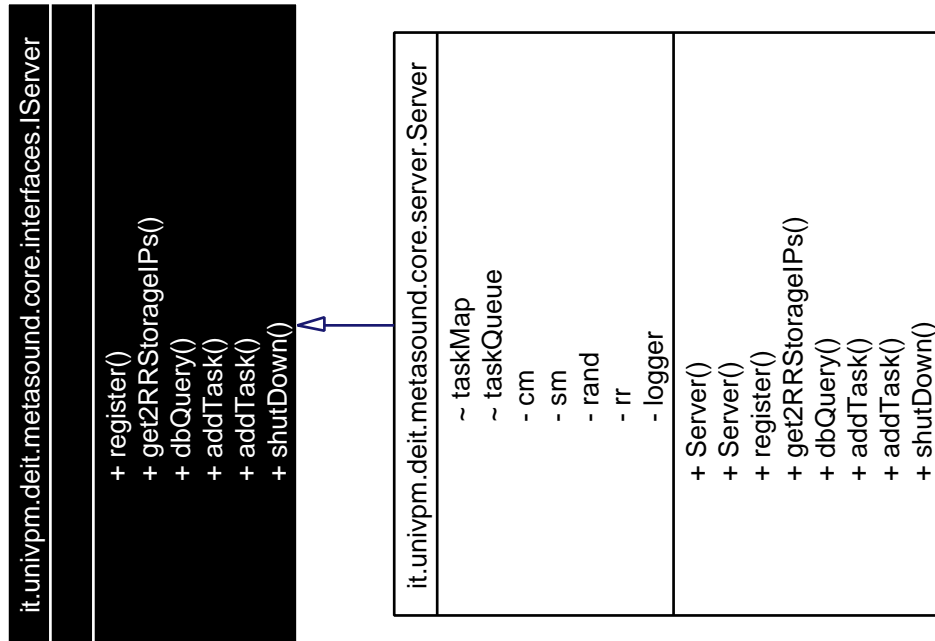


Figure 5.7: Server interface

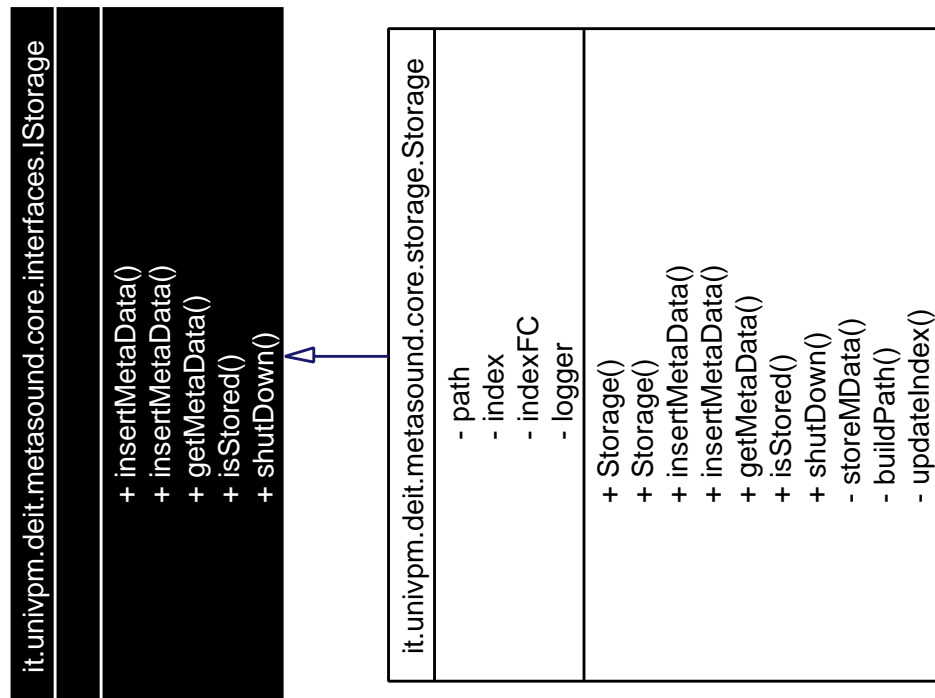


Figure 5.8: Storage interface

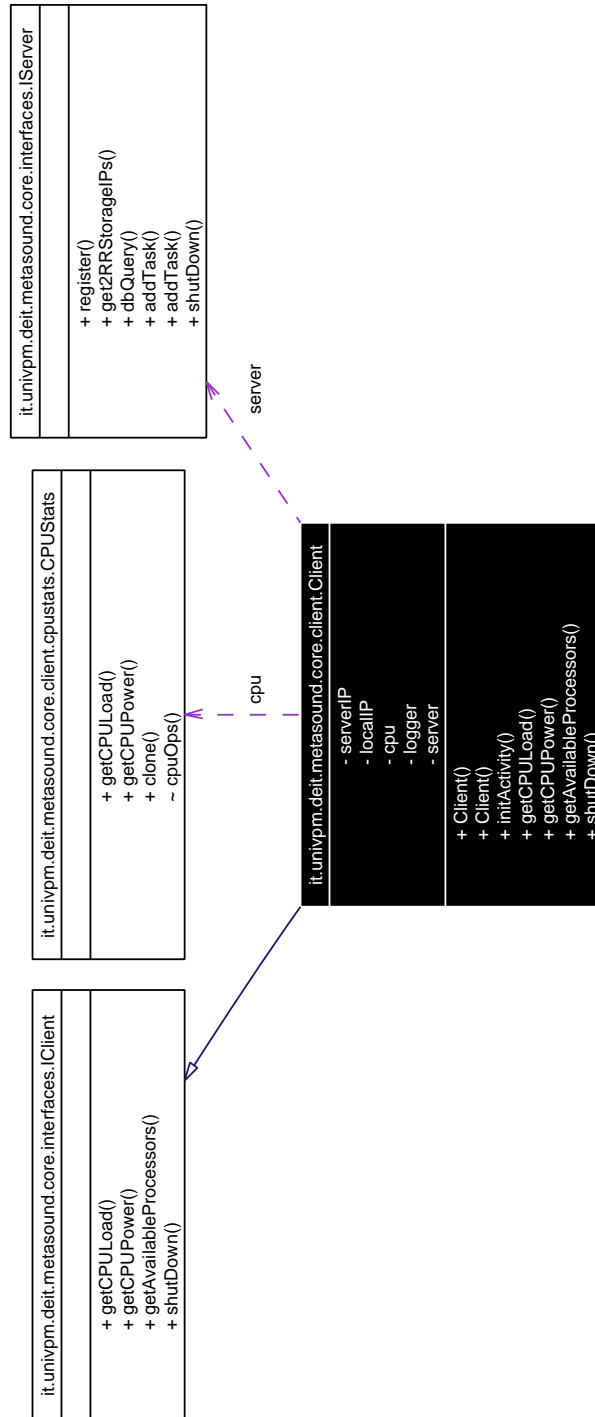


Figure 5.9: Client collaboration

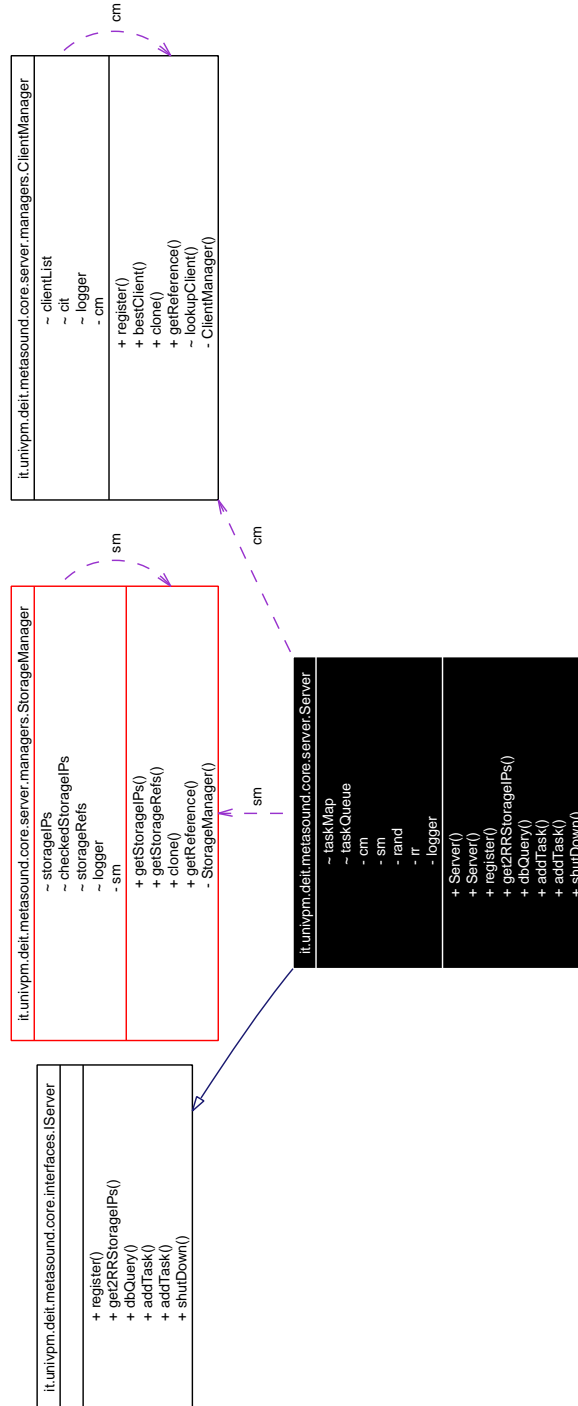


Figure 5.10: Server collaboration

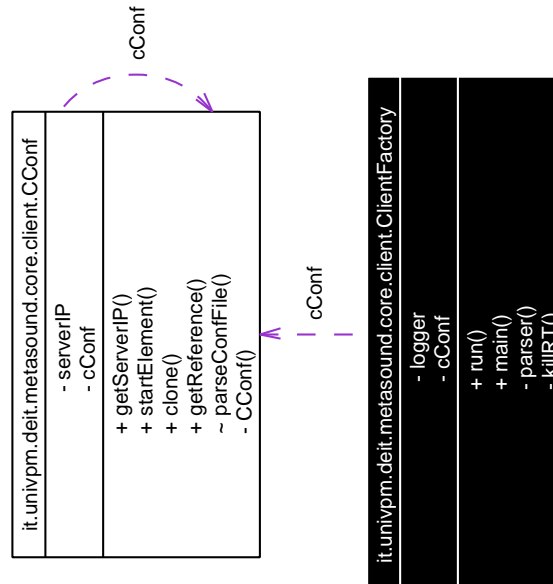


Figure 5.11: Client factory

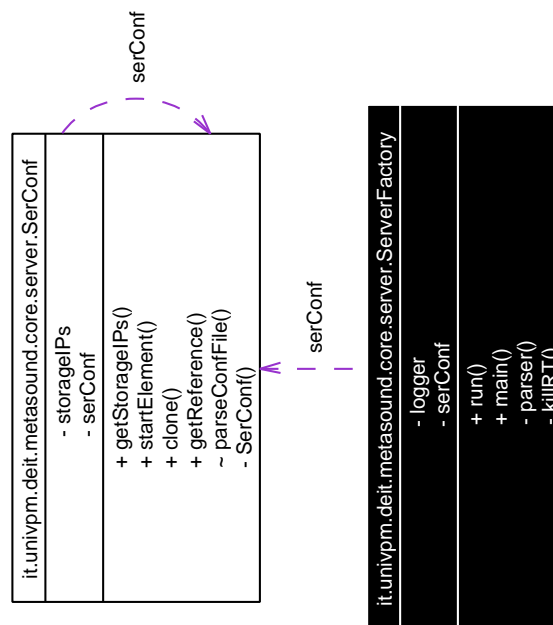


Figure 5.12: Server factory

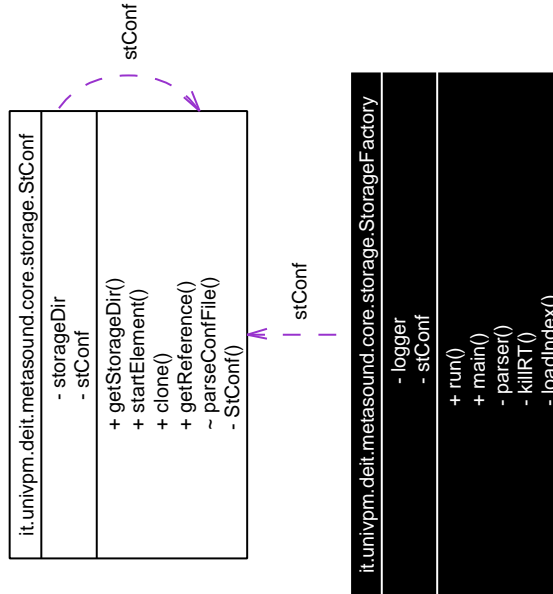


Figure 5.13: Storage factory

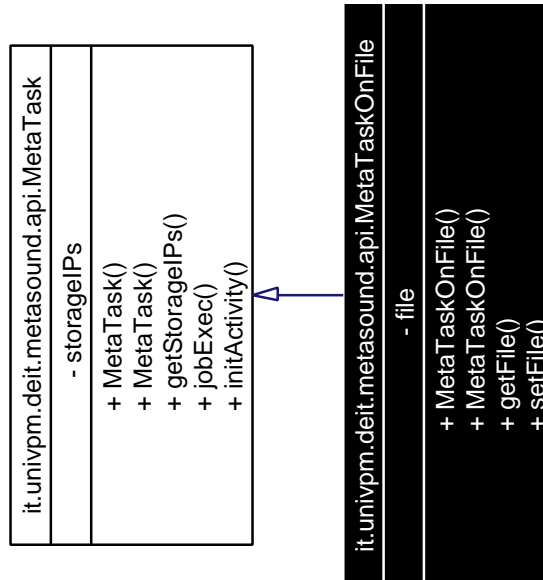


Figure 5.14: MetaTask API

Chapter 6

Conclusions

MetaSound fully meets its initial prerequisite set, providing a scalable architecture for metadata handling and storing. Albeit a comprehensive test in a realistic environment has not been executed, [8] shows how architectures realized with ProActive library could scale efficiently if a parallel-exploitable task is used. Thus, expecting similar performances from *MetaSound* is down-to-earth.

Thanks to the refactor-able structure that characterizes OO software, *MetaSound* fosters the birth of novel development branches. During its designing, some brand new desirable features have been sketched . They represent a little amount of the possible evolution paths:

- Avoid load balancing policy to be file transfer agnostic. This can be obtained letting storage tier know if it coexist on a computer where a client tier is available. Thus, task scheduling should become not only function of CPU power/load, but also of network transfer latency.
- Provide storage tier with a RDF DB facility. Mapping storage index on an RDF back-end would bring sensible advantages, compared to traditional limitations of RDBMSs.
- Modify storage policy with metadata-related heuristics. A feasible approach could be based on Vector Quantization techniques,

extracting a coherent metric from metadata files. Associating a server with one or more centroids, storage choice becomes only a matter of “distance”. This method should exploit proximity of files, for example, in “similarity” queries.

Appendix A

UML

In software engineering, *Unified Modeling Language* (UML) is a non-proprietary modeling and specification language. However, the use of UML is not restricted to software modeling. It can be used for modeling hardware (engineering systems) and is commonly used for business process modeling and organizational structure modeling.

The UML is a method used to specify, visualize, construct and document the artifacts of an object-oriented software-intensive system under development. It represents a compilation of best engineering practices which have proven to be successful in modeling large, complex systems, especially at the architectural level.

A.1 History

Graphical modeling languages have been around in the software industry for a long time. The fundamental driver behind them all is that programming languages are not at a high enough level of abstraction to facilitate discussions about design.

The UML is a relatively open standard, controlled by the Object Management Group (OMG), an open consortium of companies. The OMG was formed to build standards that supported interoperability, specifically the interoperability of object-oriented systems. The OMG is

perhaps best known for the CORBA (Common Object Request Broker Architecture) standards. The UML was born out of the unification of the many object-oriented graphical modeling languages that thrived in the late 1980s and early 1990s. Since its appearance in 1997, it has relegated that particular tower of Babel to history.

A.2 Modeling aspects

There are three prominent aspects of the modeled system, that are handled by UML:

- **Functional Model:** showcases the functionality of the system from the user's Point of View. Includes Use Cases Diagrams.
- **Object Model:** showcases the structure and substructure of the system using objects, attributes, operations, and associations. Includes Class Diagrams.
- **Dynamic Model:** showcases the internal behavior of the system. Includes Sequence Diagrams, Activity Diagrams and Statechart Diagrams.

A.2.1 Use Case Diagram

Use cases are a technique for capturing the functional requirements of a system. Use cases work by describing the typical interactions between the users of a system and the system itself, providing a narrative of how a system is used.

Use cases are well known as an important part of the UML. However, the surprise is that in many ways, the definition of use cases in the UML is rather sparse. Nothing in the UML describes how the content of a use case should be captured. What the UML describes is a use case diagram, which shows how use cases relate to each other. But almost all the value of use cases lies in the content, and the diagram is of rather limited value.

Use case diagrams depict:

- **Use cases:** a use case describes a sequence of actions that provide something of measurable value to an actor and is drawn as a horizontal ellipse.
- **Actors:** an actor is a person, organization, or external system that plays a role in one or more interactions with the system.
- **Associations:** associations between actors and use cases are indicated in use case diagrams by solid lines. An association exists whenever an actor is involved with an interaction described by a use case. Associations are modeled as lines connecting use cases and actors to one another, with an optional arrowhead on one end of the line. The arrowhead is often used to indicating the direction of the initial invocation of the relationship or to indicate the primary actor within the use case.

A.2.2 Component Diagram

The different high-level reusable parts of a system are represented in a Component diagram. A component is one such constituent part of a system. In addition to representing the high-level parts, the Component diagram also captures the inter-relationships between these parts.

Component diagrams represent the implementation perspective of a system. Hence, components in a Component diagram reflect grouping of the different design elements of a system, for example, classes of the system.

A.2.3 Class Diagram

A class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them. Class diagrams also show the properties and operations of a class and the constraints that apply to the way objects are connected. The UML uses the term *feature* as a general term that covers properties and operations of a class.

A class diagram is composed primarily of the following elements that represent the system's business entities:

- **Class:** a class represents an entity of a given system that provides an encapsulated implementation of certain functionality of a given entity. These are exposed by the class to other classes as *methods*. Apart from business functionality, a class also has properties that reflect unique features of a class. The properties of a class are called *attributes*.
- **Interface:** an interface is a variation of a class. It provides only a definition of business functionality of a system. A separate class implements the actual business functionality.
- **Package:** a package provides the ability to group together classes and/or interfaces that are either similar in nature or related. Grouping these design elements in a package element provides for better readability of class diagrams, especially complex class diagrams.

To show different class relationships, notations are available for: Association, Multiplicity, Directed Association, Reflexive Association, Aggregation, Composition, Inheritance/Generalization, Realization.

Appendix B

Tools and Licenses

MetaSound developing is thoroughly influenced by the open-source philosophy: all tools and facilities used to foster *MetaSound* belong to FOSS world.

- **Eclipse**: being *MetaSound* a full Java project, it has been chosen because it is one of the actual Java IDEs of choice (includes CVS support)
- **SourceForge.net**: the essence of the Open Source development model is the rapid creation of solutions within an open, collaborative environment. Thus, being part of MPEG7AudioDB project, *MetaSound* is hosted at <http://sourceforge.net/projects/mpeg7audiodb/> and its source code can be checked-out from the main CVS (“metasound” module).
- **L^AT_EX**: used to write this thesis
- **Doxygen**: used to create *MetaSound* documentation, including UML diagrams contained in this thesis

Consequently, both source code and documentation of *MetaSound* are released under “open-licenses”.

- *MetaSound* source code is released under MIT license

```
/*
 * Copyright (c) 2005 Michele Catasta
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the
 * "Software"), to deal in the Software without
 * restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute,
 * sublicense, and/or sell copies of the Software, and
 * to permit persons to whom the Software is furnished
 * to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or
 * substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
 * COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES
 * OR OTHER LIABILITY, WHETHER IN AN ACTION OF
 * CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF
 * OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALING IN THE SOFTWARE.
 */
```

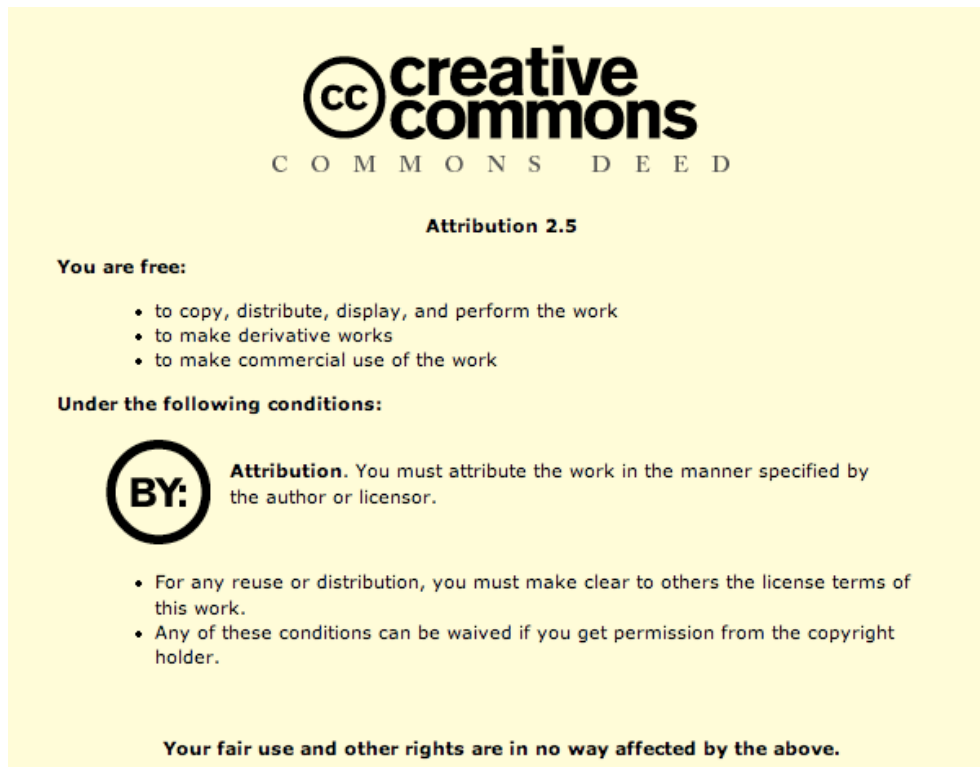


Figure B.1: human read-able summary of the Legal Code (Attribution License)

- *MetaSound* thesis is licensed under the Creative Commons Attribution License

B.1 Tools

This section provides an overview of tools and facilities used for *MetaSound* developing.

B.1.1 Eclipse

Eclipse is an open-source platform-independent software framework for delivering what the project calls “rich-client application” (as opposed to “thin clients”, this means the clients perform heavy-duty work on the host running the application). So far this framework has typically been

used to develop IDEs (Integrated Development Environments), such as the highly-regarded Java IDE called *Java Development Toolkit* (JDT) and compiler that come as part of Eclipse (and which are also used to develop Eclipse itself).

History

Eclipse was originally developed by IBM, but is now developed by the Eclipse Foundation, an independent not-for-profit consortium of software industry vendors. Many notable software tool vendors have embraced Eclipse as a future framework for their IDEs, among them Borland and IBM Rational.

Architecture

The basis for Eclipse is the rich client platform (RCP). The following components constitute the rich client platform:

- Core platform (boot Eclipse, run plug-ins)
- OSGi (a standard bundling framework)
- SWT (a portable widget toolkit)
- JFace (file buffers, text handling, text editors)
- The Eclipse Workbench (views, editors, perspectives, wizards)

Eclipse's widgets are based on IBM's third generation widget toolkit for Java called SWT, unlike most Java applications, which use Sun's first and second generation toolkits (AWT and Swing, respectively). Eclipse's user interface also depends on an intermediate GUI layer called JFace which simplifies the construction of applications based on SWT.

Eclipse employs plug-ins in order to provide all of its additional functionality on top of the rich client platform, in contrast to some other IDEs where functionality is typically hard-coded. This plug-in mechanism is a lightweight software componentry framework and for example allows

Eclipse to support other languages in addition to Java. Separate plug-ins have been created that add support for, among others, C/C++ (CDT), Perl, Ruby, Python, telnet and database development. The plug-in architecture supports writing any desired extension to the environment, such as for configuration management. It does not have to be used solely to support other programming languages.

B.1.2 SourceForge.net

SourceForge.net (<http://www.sourceforge.net/>) is a centralized location for software developers to control and manage open source software development, and acts as a source code repository. SourceForge.net is hosted by VA Software and runs a version of the SourceForge software. A large number of open source projects are hosted on the site (it had reached 100,000 by May 2005), making it the largest repository of Open Source code and applications available on the Internet.

B.1.3 L^AT_EX

L^AT_EX is a document preparation system for the T_EX typesetting program. It offers programmable desktop publishing features and extensive facilities for automating most aspects of typesetting and desktop publishing, including numbering and cross-referencing, tables and figures, page layout, bibliographies, and much more.

L^AT_EX was originally written in 1984 by Leslie Lamport and has become the dominant method for using T_EX; few people write in plain T_EX anymore. The current version is L^AT_EX 2_ε.

Features

L^AT_EX is based on the idea that authors should be able to concentrate on writing within the logical structure of their document, rather than spending their time on the details of formatting. It encourages the separation of formatting from content, whilst still allowing manual typesetting ad-

justments where needed. By keeping the formatting details in a separate file from the text, it is often regarded as superior to word processors and most other desktop publishing systems, which allow trivially easy visual layout changes but tend to intertwine content and form so tightly that consistency and automation are often difficult. \LaTeX can be arbitrarily extended by using the underlying macro language for developing custom formats.

Community

\LaTeX was originally most commonly used by mathematicians and scientists, amongst whom it remains the favored tool for writing papers, preprints, and books. Because of the underlying \TeX system, originally developed for documents with mathematics, laying out mathematical expressions is considered to be easier, and the resulting typesetting of higher quality, than any competing document-processing systems.

The popularity of \LaTeX in the technical and academic communities is perhaps partly due to its early availability on Unix systems, and the comparative unavailability of competing word processors on those platforms until recently. But from an early stage \LaTeX was available on a wider range of hardware and software than any other program, and versions are now available for almost any system from PDAs to desktop PCs to supercomputers.

Licensing issues

\LaTeX is free software. It has a peculiar license called LPPL, not compatible with the GNU General Public License, that allows redistribution and modification, but requires that modified files carry a modified filename. This ensures that files that depend on other files will produce the expected behavior and avoids problems similar to DLL hell. A new version of the LPPL that will be compatible with the GPL is in the works.

B.1.4 Doxygen

Doxygen is a documentation system for C++, C, Java, Objective-C, IDL (Corba and Microsoft flavors) and to some extent PHP, C#, and D.

It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in \LaTeX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.

Doxygen is developed under Linux and Mac OS X, but is set-up to be highly portable. As a result, it runs on most other Unix flavors as well.

It is released under the terms of the GNU General Public License.

B.2 Licenses

This section provides an overview of licenses used to release *MetaSound* project (source code and documentation).

B.2.1 MIT License

The *MIT License*, originated at the Massachusetts Institute of Technology, is a license for the use of certain types of computer software. It allows commercial use, and for the software released under the license to be incorporated into commercial products. Works based on the material may even be released under a proprietary license

The MIT License is most similar to the 3-clause BSD license, which is essentially different only in the fact that it contains a notice prohibiting the use of the name of the copyright holder in promotion. The 4-clause BSD license also includes a clause requiring all advertising of the software to display a notice; the MIT License has never had this clause. The MIT license, however, more explicitly states the rights given to the end-user, including the right to use, copy, modify, merge, publish, distribute, sublicense, and/or sell the software.

B.2.2 Creative Commons

The *Creative Commons* (CC) is a non-profit organization devoted to expanding the range of creative work available for others to legally build upon and share.

The Creative Commons website enables copyright holders to grant some of their rights to the public while retaining others, through a variety of licensing and contract schemes, which may include dedication to the public domain or open content licensing terms. The intention is to avoid the problems which current copyright laws create for the sharing of information.

The project provides several free licenses that copyright holders can use when they release their works on the web. They also provide RDF/XML metadata that describes the license and the work to make it easier to automatically process and locate licensed works.

Bibliography

- [1] G. Paroux, B. Toursel, R. Olejnik, V. Felea: *A Java CPU calibration tool for load balancing in distributed applications*, Proceedings of the ISPDC/HeteroPar04, IEEE (2004)
- [2] J. B. Jimenez: *Robin Hood: An Active Objects Load Balancing Mechanism for Intranet*, <http://www.dcc.uchile.cl/~jbustos/Pub/rh.pdf>
- [3] N. G. Shivaratri, P. Krueger, M. Singhal: *Load Distributing for Locally Distributed Systems*, IEEE Computer Magazine (December 1992)
- [4] D. C. Schmidt, R. G. Lavender: *Active Object*, <http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf>
- [5] D. C. Schmidt, S. Vinoski: *Introduction to Distributed Object Computing*, SIGS Vol. 7 No. 1 (January 1995)
- [6] G. Tummarello: *Scalable Merging of Multimedia annotations using P2P, MPEG-7 and Semantic Web technologies*, Ph.D. thesis (2004)
- [7] Oasis Groups at INRIA Sophia-Antipolis: *ProActive Manual*, Revision 2.2 (April 2005)
- [8] F. Huet, D. Caromel, H. Bal: *A High Performance Java Middleware with a Real Application*, SuperComputing Conference (November 2004)

- [9] D. Caromel, W. Klauser, J. Vayssiere: *Towards Seamless Computing and Metacomputing in Java*, Concurrency Practice and Experience (September-November 1998)
- [10] P. Van Roy, S. Haridi: *Concepts, Techniques and Models of Computer Programming*, The MIT Press (2004)
- [11] B. Eckel: *Thinking in Java*, 3rd Edition, Prentice Hall PTR (2002)
- [12] D. Hunter, A. Watt, J. Rafter, K. Cagle, J. Duckett, B. Patterson: *Beginning XML*, 3rd Edition, Wrox (2004)
- [13] J. Bloch: *Effective Java - Programming Language Guide*, Addison-Wesley Professional (2001)
- [14] E. R. Harold: *Java Network Programming*, 3rd Edition, O'Reilly (2004)
- [15] J. Farley: *Java Distributed Computing*, O'Reilly (1998)
- [16] D. Reilly, M. Reilly: *JavaTM Network Programming and Distributed Computing*, Addison-Wesley Professional (2002)
- [17] W. Grosso: *Java RMI*, O'Reilly (2001)
- [18] Z. Tari, O. Bukhres, *Fundamentals of Distributed Object Systems: The CORBA Perspective*, Wiley-Interscience (2001)
- [19] M. Fowler: *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd Edition, Addison-Wesley Professional (2003)
- [20] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*, Addison-Wesley Professional (1995)
- [21] A. Shalloway, J. Trott: *Design Patterns Explained: A New Perspective on Object-Oriented Design*, 2nd Edition, Addison-Wesley Professional (2004)

- [22] B. Eckel: *Thinking in Patterns*, <http://www.mindview.net/Books/TIPatterns/>, Revision 0.9 (2003)
- [23] T. Oetiker, H. Partl, I. Hyna, E. Schlegl: *The Not So Short Introduction to L^AT_EX 2_ε*, Version 4.16 (2005)
- [24] Various Authors: *Wikipedia - various Wiki-entries*, http://en.wikipedia.org/wiki/Main_Page