# Politecnico di Milano

## Facoltà di Ingegneria dell'Informazione




Corso di Laurea in Ingegneria Informatica

Dipartimento di Elettronica e Informazione


# An On-Line Aspect-Oriented Monitoring Tool for the SelfLet Framework


Relatore: Elisabetta Di Nitto

Tesi di Laurea di: Francesco NIGRO matr. 666733



Anno Accademico 2009-2010

*alle persone che hanno saputo credere in me, la mia famiglia, i miei amici e Sara, il mio amore...*

# Ringraziamenti

Ringrazio tutte le persone che mi hanno supportato in questa sofferta stesura del lavoro di tesi, a partire da coloro che mi hanno donato la possibilità di esprimere la mia passione per questa disciplina, come la prof.ssa Di Nitto o Nicolò Calcavecchia. Ringrazio inoltre tutti i miei amici e la mia famiglia, che mi hanno spinto e aiutato con ogni mezzo ad affrontare questo ultimo capitolo della mia vita universitaria. Infine, per ultima, ma nel mio cuore, più cara, va la mia gratitudine e stima alla mia ragazza, Sara, che ha stretto i denti insieme a me durante tutto questo duro periodo, spronandomi, con la dolcezza e delicatezza di cui solo lei è capace, a non perdere mai la speranza in un futuro migliore, da costruire con forza e determinazione: qualità che mi miglioreranno sicuramente come ingegnere, ma soprattutto, come uomo. Grazie.

Milano, 19 Marzo 2011

*Francesco*

# Sommario

La proliferazione di sistemi tecnologici nella quotidianità, sta fortemente cambiando il mondo in cui viviamo. Come conseguenza di questa esplosione tecnologica, la domanda relativa a professionisti capaci di gestire e mantenere questi sistemi, sta costantemente crescendo. Sfortunatamente, la richiesta sta superando l'offerta di personale specializzato, sollevando così un problema mai affrontato prima: gestire la complessità dei sistemi informatici.
L'Autonomic Computing è l'ambito di ricerca interessato allo studio di sistemi capaci di autogestirsi, senza l'intervento umano. Questi sistemi si possono caratterizzare per le cosiddette proprietà *Self-* (auto-configurazione, auto-guarigione, auto-ottimizzazione e auto-protezione), che prendono ispirazione dal sistema autonomo nervoso del corpo umano.
L'Autonomic Computing è anche il contesto di riferimento per le SelfLet: una SeflLet[17] è un singolo componente autonomico autosufficiente, che posto in una rete è in grado di comunicare con altre SelfLet. Le SelfLet vengono descritte tramite degli obiettivi da ottenere, dei comportamenti capaci di raggiungere gli obiettivi e delle regole per far fronte a situazioni critiche.
Il lavoro svolto in questa tesi estende il supporto all'utilizzatore delle SelfLet, attraverso la creazione di uno strumento che permetta il monitoraggio distribuito di una rete di SelfLet. Tale strumento deve inoltre risultare:

- di semplice utilizzo
- facilmente estendibile
- altamente configurabile
- a minimo impatto prestazionale

Inizialmente è stata svolta una ricerca delle soluzioni a problemi simili, con l'obiettivo di analizzare differenti approcci al monitoraggio di sistemi dis-

tribuiti; successivamente, una ricerca in letteratura delle architetture e delle tecnologie scelte per favorire una buona manutenibilità della soluzione. Il lavoro continua con lo studio di un modello architetturale per il sistema di monitoraggio che prediliga la semplicità e la robustezza. Particolare attenzione viene data alle tecniche di progettazione tipiche della realizzazione di sistemi programmati difensivamente. Succesivamente è stato realizzato il cuore dell'architettura attraverso le metologie e con gli strumenti trovati nel corso dell'attività di ricerca iniziale. Infine, l'architettura proposta è stata validata attraverso l'implementazione di uno strumento (compatibile con il modello stesso dell'architettura) da applicare a due casi di studio, i cui risultati vengono analizzati in modo tale da quantificare il contributo del lavoro svolto.

## Organizzazione

Questa tesi è organizzata come segue.

- Nel Capitolo 2 viene introdotto lo stato dell'arte dei sistemi di monitoraggio distribuiti. In particolare, vengono presentate le caratteristiche ed i problemi tipici di questo genere di sistemi, nonchè la soluzione proposta da "Technischen Universität München"[1].

- Il Capitolo 3 presenta gli strumenti adottati nello sviluppo della soluzione proposta, una breve descrizione del framework autonomico oggetto del monitoraggio e dell'architettura della soluzione realizzata, con una panoramica di una implementazione conforme ad essa.

- Il Capitolo 4 infine presenta i casi di studio implementati ed i risultati ottenuti dalla loro esecuzione.

La tesi si conclude con il Capitolo 5: sono tratte delle conclusioni e sono presentati alcuni possibili sviluppi futuri.

[1] Università Tecnica di Monaco di Baviera

# Table of Contents

# Chapter 1

# Introduction

The proliferation of technology in everyday life, is greatly changing the world in which we live. As a result of this explosion of technology, the demand for professionals capable of managing and maintaining these systems is steadily growing. Unfortunately, demand is outpacing the supply of skilled personnel, thus relieving a problem never faced before: to manage the complexity of computer systems.
The Autonomic Computing is the field of research concerned the study of systems able to manage themselves without human intervention. These systems can be characterized by the so-called *Self-* properties (self-configuring, self-healing, self-optimizing and self-protection), taking inspiration from the autonomous nervous system of human body.
The Autonomic Computing is the reference framework for SelfLet: a SeflLet[17] is a single self-contained stand-alone component, which if is placed in a network can communicate with other SelfLet. The SelfLet are described by the objectives to be achieved, for conduct capable of achieving the objectives and rules to deal with critical situations.
The work in this thesis extends support to the SelfLet's user through the creation of a tool that allows monitoring distributed network of SelfLets. The tool should also be:

- easy to use
- easily extensible
- highly configurable

- with minimal performance impact

Was initially carried out a search for solutions to similar problems, with the aim to analyze different approaches to monitoring distributed systems, and thereafter a search of the literature of architecture and technology choices to promote good maintainability of the solution. Work continues with the study of an architectural model for the monitoring system that focuses on simplicity and robustness. Particular attention is given to the technical design of the typical implementation of systems programmed defensively. Successive was created the core of the architecture through the metodologies and the tools found in the work of research. Finally, the proposed architecture was validated through the implementation of a tool (compatible with the architectural model itself) to be applied to two case studies, whose results are analyzed in order to quantify the contribution of the solution realized.

## 1.1 Outline of the Thesis

This thesis is organized as follows:

- Chapter 2 introduces the state of the art about monitoring of distributed systems. In particular, it presents the characteristics and problems typical of such systems, as well as the proposed solution from the "Technischen Universität München"[1].
- Chapter 3 presents the tools used in the development of the solution proposal, a brief description of the autonomic framework object of the monitoring and the architecture of the solution made, with an overview of an implementation compliant to it.
- Chapter 4 then presents the case studies implemented and the results obtained from their execution.

The thesis concludes with Chapter 5: the conclusions and the presentation of some possible future developments.

[1] Technical University of Monaco of Bavaria

# Chapter 2

# State of the art

## 2.1 Monitoring

Increasing complexity is an undeniable trend in modern software. The rise in complexity affects not only developers involved in constructing large and complex systems, but also those who intend to debug and maintain the systems, or reuse existing code in new systems: before commencing work, developers have to understand the existing system code.
Understanding code for maintenance reasons is related to the idea of understanding it for educational purposes. A common technique for interpreting how complex software works consists of reading its source code. This approach has its problems: not only it is taxing having to decipher previously unseen code, often it can be difficult for the reader to work out which area of the code s/he meant to focus on, or which program components s/he should be looking at. Traditional methods of collecting program monitoring information suffer from various problems, which range from not providing wide enough spectrum of information to being intrusive.
This kind of problems are magnified in the cases of softwares organized with a distributed approach, as autonomic distributed system, because there are additional issues related to the consistency of monitoring data collected along the execution of the nodes being part of the overall system.

### 2.1.1 The purpose of monitoring

Monitoring is carried out in order to obtain information about a system, and in general, monitoring is part of the process of management. Among the many activities which involve monitoring is possible to find:

- debugging
- testing
- accounting
- performance evaluation
- resource utilisation analysis
- security
- fault detection
- teaching aid.

Monitoring and its management are concerned with providing the necessary information in order to allow the construction of the required model of the observed system and its presentation. It is the purpose of monitoring which dictates what should be observed and also how the information is to be obtained.

### 2.1.2 Modelling and the level of monitoring

The different purposes for which monitoring is carried out can be interpreted at different levels. Thus, for example, debugging a single object as opposed to debugging the interactions among multiple objects will require different events to be observed. A language debugger will require events to be generated at a smaller level of granularity with respect to the one which is aimed at debugging the interactions between objects.
Some of the models constructed for the purposes listed in the following sections will require a different model or models of the distributed system. The exact level of modelling will dictate the granularity of the events the observer wishes to monitor.

### 2.1.3 The problems of monitoring

The following list is an exposition of the problems encountered when monitoring centralized and distributed computer systems[27]:

1. **Direct and indirect observations**: the behavior of some systems can be directly observed, thereby making the process of monitoring relatively straight forward. In computer systems most events of interest cannot be observed directly without special facilities, thereby requiring the incorporation of a monitoring infrastructure in such systems. The monitoring infrastructure will also facilitate the management of monitoring. There may be several levels of indirection between the observed system and the observer. Distribution complicates the process of monitoring, introducing additional levels of indirection and subsequently additional problems.

2. **Complete and incomplete observations**: completeness and incompleteness refer to whether the information necessary in order to construct a particular model of an observed system is available or not. It could be argued that any observation of a system only reveals part of the system. This is not a problem when the observer is constructing a particular model of the system and the observation fits this model. However, incompleteness can cause problems when it is not intended or not catered for.

3. **Presentation problems**: in many cases it is necessary to modify the information from the observed events if that events occur at a *rate* which cannot be easily used by the observer, appear in a *form* which is not suited for immediate use by the observer or the *volume* of events may be such that it overwhelms the observer.

4. **Monitor and interference**: there is a relation between the flexibility of the monitoring facilities, the cost of implementation, and the extent to which they interfere with the behavior of the system. The most general requirement from monitoring, which is independent of the purpose for which it is introduced, is that although the sequence of system events may change as a result of the interference caused by monitoring, it must not result in an illegal sequence of events taking place.

### 2.1.4 Program Monitoring

Program Monitoring is the regular observation and recording of activities taking place in a program. It is a process of routinely gathering information on all aspects of the project. To monitor is to check on how program activities are progressing. It is the performing of systematic and purposeful observations. Monitoring also involves giving feedback about the progress of the program to the implementors and beneficiaries of the system. Reporting enables the gathered information to be used in making decisions for improving program performance.
The above definition of the monitoring problem is usually faced with two kind of approaches[27]:

- event-driven
- data-driven

The major pitfall of the event-driven approach is that it requires modification of source code, which is intrusive. The alternative data-driven approach monitors the target program for functions calls and data modifications. Programs can be monitored in various ways, which include hardware monitoring , postprocessing the executable form of the program, modifying the language processor and preprocessing the program source.
The four techniques presented below are often used in combination, as they all have their own strengths and weakness, as well as collecting different type of information. While they do not involve modification of source code, they still require modification of existing tools, some at quite low level, or creation of completely new tools. Furthermore, they are non necessary capable of collecting the required monitoring information.

#### Hardware Monitoring

Since computer programs must be executed by a hardware processor, it is possible to modify the processor to start monitoring the execution of the program when a certain condition occurs. This monitoring can take various forms, such as invocation of a debugger, which will single-step the program, or use of memory access traps to monitor changes in data structures. A typical problem

encountered with hardware monitoring is that it is difficult to control granularity refers to the level of detail of animation events. Since any change to a variable being mapped causes the program state to change, by default the granularity is high: usually some mechanism is used to control the granularity i.e. determine when the state has changed sufficiently to require visualization updating.
For many programming languages, the primary problem with this approach is that it generally means that informations will be reported in terms of assembly language. This not have to be the case for Java however, since the Java Platform Debugger Architecture provides the Java Debug Interface (JDI). The JDI allows for collection of debugging and tracing information from a running Java program without modification of source code.

### Postprocessing the Executable Form of the Program

This approach involves modifying the program's executable form directly, as opposed to modifying the processor or the language software. An advantage of this approach is that the program can be changed after the language processor, and even the target program itself has been loaded and run. In other words, it is possible to modify the program modifying the program binary and without having to do additional program retranslations. A disadvantage is that the only information yielded will be that which can be extracted from the program binary.

### Modifying the language processor

To avoid having to work with assembly language information or worse still, binary output, it is possible to monitor programs by modifying the language processor directly. This approach can be successfully applied to both compiler processors and interpreter processors, but it does mean that the language processor needs to be extended to include the monitoring.

### Preprocessing the Program Source

This involves using a preprocessor to modify the target program source prior to sending it to the language processor. The task of the preprocessor is to insert statements in relevant places that will output useful monitoring information.

A clear advantage is that program source code is dealt with, and existing software and hardware does not need to be modified. A disadvantage is that a preprocessor needs to be created, and furthermore, one needs to be careful that the source code after preprocessing will present the same program behavior to the user as it did prior to preprocessing.

## 2.2 Distribution and monitoring

This section discusses aspects of distribution which affect monitoring: physical separation, concurrency, heterogeneity, federation, scaling and evolution. A summary of the assumptions which are no longer valid when monitoring distributed systems, as opposed to centralised systems, is presented. The section also identifies the three major problem areas of providing monitoring in distributed systems: management of monitoring, reconstruction of the causal flow of events, and presentation of monitoring information.

### 2.2.1 Aspects of distribution

The following sections discuss the aspects of distribution which have an effect on monitoring: physical separation, concurrency, heterogeneity, federation, scaling and evolution[27].

#### Physical separation

In a distributed system the physical separation of objects is unavoidable. In addition, communication delays among objects are usually variable and unpredictable. As a result there is no single point of reference from which events in the entire system can be directly observed. In order to obtain a global view of the system it is necessary to collect information on local events from several locations, from which a reconstruction of the flow of global events can be made. For example, to determine whether a certain event at one location is causally related to another event at some other location.
There are situations in which it is not possible to monitor events in certain parts of the system. This may be the result of the absence of monitoring facilities, or policy decisions imposed on an object. There are two additional complications in distributed systems:

- failures can occur during communication
- services may partially fail.

Such failures may affect not only the activities being monitored, but also the monitoring of these activities, resulting in incomplete information.
This complicates the reconstruction of the flow of events in the system, and results in an incomplete picture of the system.
Distributed systems are characterized by the possibility of partial failures. Partial failures may lead to situations where some but not all of the managed objects in a system can be accessed. Moreover, some of the management infrastructure itself may fail. Fault tolerant techniques may therefore have to be applied to the management facilities themselves in order to make them more robust to failure.
The physical separation of systems together with the variable communication delays also means that there is no single point of control in a distributed system. This together with the absence of a single point of observation means that checkpoints, tracing, breakpoints and single stepping of a distributed application are difficult, if not impossible without changing the nature of the system.
If a system is distributed, the absence of a single point of control and the absence of a single point of observation implies that the monitor and controller must be distributed as well. Furthermore, in some systems the decision making process may either be distributed and/or have to be carried out in the face of incomplete information.

### Cuncurrency

Distributed systems will support multiple objects and activities. Bindings between objects will be set up and discarded, and objects will be able to invoke other objects asynchronously through these bindings. Furthermore, objects will be created and destroyed as the need arises. The dynamic initiation and termination of activities will lead to situations where the activities stemming from an application may spread throughout the system. The extent of the initiated activities may not be known in advance.
From the point of view of monitoring this creates several difficulties. In order to gain sufficient understanding of the flow of events in a system it may not be

enough to monitor a single object or simply its interactions with other objects. In fact we may wish to gain information on how activities spread in a system. Thus we may wish to:

- fully activate monitoring of objects with which a monitored object interacts
- follow the chain of activity as it moves from one object to another

As different combinations of these strategies may occasionally be required, the management of the monitoring activities in such circumstances will be difficult unless extremely flexible management structures can be provided.
Together with different monitoring activation strategies, additional event information to allow the observer to follow activities throughout the system is necessary.

### Heterogeneity

Large scale distributed systems inevitably include some diversity in their hardware, operating systems and their distributed system infrastructure. It is reasonable to assume, therefore, that this diversity will be reflected in the implementations of monitoring facilities. Distribution does not only refer to the run-time physical separation of components, but also to the possibility of a distributed development environment. In such a case it is possible to have different implementations of monitoring which do not conform to one another. In order to make possible monitoring across heterogeneous systems, it is necessary to reach agreement on monitoring conformance issues.
Standard management facilities cannot be assumed across domain boundaries. Different monitoring and control facilities may exist in different management domains.
The problem of the integration of management infrastructures where different monitoring and control facilities may exist can be overcome through agreement on facilities or by the incorporation of facilities which allow dynamic integration of the different local management facilities.

### Federation

The existence of centralised ownership and universal and technical control in large scale distributed systems cannot be assumed, and separate sources of authority will inevitably reside side by side. In such systems a "federated" style of interworking will be necessary in which no participant is in control of the others. Each system controls its own services locally according to its policies. Different monitoring policies must be anticipated within federated systems and problems will arise when attempting to monitor across federation boundaries between systems whose monitoring policies clash. Cooperation between systems requires the parties responsible for them to negotiate the use of services either prior to the request for use of service or as a result of such a request.

### Scaling

As discussed in the section on concurrency, activities in a distributed system can spread and encompass large parts of the system. In cases where monitoring is expected to report on such activities, it is important to note that the monitoring activity itself will have to spread, thus consuming increasing storage, processing and communication resources. It is therefore essential that (the distribution of) monitoring itself scales well.
The requirement for scaling needs monitoring structures which can accommodate distribution, system evolution, and growth of the activity in the face of resource constraints and performance requirements. Both management and collation structures must be designed with scaling in mind.

### Evolution

Distributed systems will evolve over time, possibly in an inconsistent manner. If monitoring procedures change over time, there may be clashes between monitoring standards embedded in new components and monitoring standards in existing components. The problems arising in evolving systems are often similar in nature to those arising in heterogeneous and federated systems.

### 2.2.2 Conclusions about monitoring in a distributed system

The problems cited above which distribution introduces to monitoring can be grouped together into three major problem areas:

1. the definition, design and incorporation of a monitoring and management infrastructure to facilitate the dynamic monitoring of distributed systems

2. ordering and reconstruction of the flow of events in a distributed system from the monitoring information: the transformation of a collection of monitoring information of local events into a global picture. The ability to reconstruct can be seen as a pre-requisite for providing useful presentations of monitoring information

3. the visualization of monitoring information in order to provide the observer with useful models of the system and the activities in it

## 2.3 The OMIS approach

OMIS (On-line Monitoring Interface Specification)[24] is the definition of a standard interface between various types of run-time tools for parallel and distributed systems and the systems themselves. Speaking of run-time tools we mean debuggers, performance analyzers, program flow and result visualizers, load and resource management systems etc. These tools require means for observation and manipulation of the execution of parallel programs. Different tools need similar sets of information and manipulation facilities. These facilities are called *monitoring systems* and must be implemented for a large variety of target systems.
A monitoring system with a standardized interface allows to quickly supply various target systems with the same powerful set of tools. OMIS is the definition of such an interface. It allows tool developers to attach new tools to already existing implementations of OMIS compliant monitoring systems on different target architectures. An OMIS compliant monitoring system can concurrently serve several compliant tools, thus offering a means for tool interoperability. Universality with respect to new tool environments is guaranteed by OMIS' intrinsic mechanisms of extendibility.

OMIS is not the first approach that tries to define a standard interface for middle-ware software layers, however, it's currently the only one that concentrates on complete tool environments for parallel and distributed system. An overall view of the system model of an OMIS compliant monitoring system embedded into an environment with tools and a parallel programming library is presented in Figure 2.1.

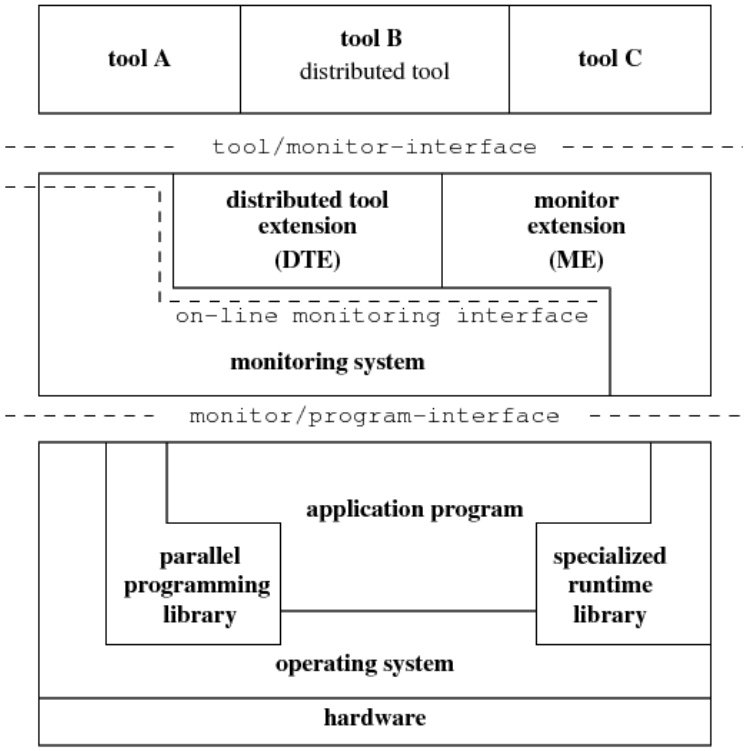


Figure 2.1: System model OMIS 2.0 compliant.

The cooperation between the tool and the monitoring system is based on the *service request/reply* mechanism. A tool sends a service request to the monitoring system, e.g. as a coded string which describes a condition (event) (if any) and activities (action list) which have to be invoked (when the condition gets true). In this way the tool programs the monitoring system to listen for event occurrences, perform needed actions, and transfer results to the tool. OMIS relies on a hierarchy of the abstract objects: nodes, processes, threads, messages queues and messages (see Figure 2.2). Every object is represented by an abstract identifier (*token*) which can be converted into other token types by the conversion functions *localization* and *expansion*

which are automatically applied to every service definition that has tokens as a parameter. Each tool at each moment has a well defined scope, i.e. it can observe and manipulate a specific set of objects attached on a request from this tool.

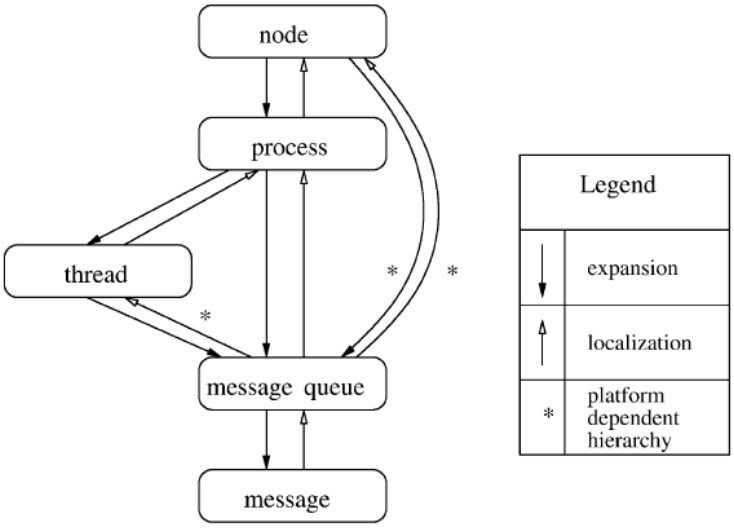


Figure 2.2: Hierarchy of objects in OMIS.

For each system object OMIS defines a set of services that belong to one of the three categories:

- **information services**: provide information about the object
- **manipulation services**: allow to manipulate the object
- **event services**: trigger arbitrary actions whenever a matching event takes place

Due to the distributed nature of parallel application, the monitoring system must itself be distributed and needs one monitoring component per node.
OMIS allows the monitoring system to be expanded with a *tool extension* or *monitor extension*, which adds new services and new types of objects to the basic monitoring system, for specific programming environments.
In the next section an extended version of this standard specifically designed to work with Java and an istance of this specification that represents the current state of art of a monitoring distribuited system for Java is presented.

### 2.3.1 J-OMIS

The OMIS specification focuses on the definition of an interface between tools and a monitoring system.
The original OMIS specifies a hierarchy of system objects for the message passing paradigm, stemming from its initial focus on PVM applications (see Figure 2.2). The specific features of Java necessitated to review this hierarchy of system objects and to introduce new objects and relationships between them, which underly the *J-OMIS* (Java-bound On-line Monitoring Interface Specification)[26]. The extended specification (see Figure 2.3) distinguishes between two kinds of system objects: *execution objects*, i.e. nodes, JVMs, threads and *application objects*, i.e. interfaces, classes, objects, methods.

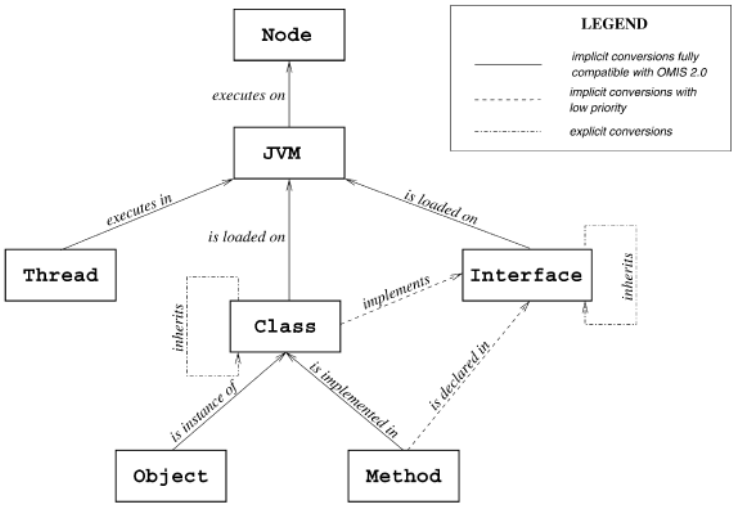


Figure 2.3: Hierarchy of objects in Java oriented extension to OMIS.

Figure 2.4 shows the software component structure of the monitoring system. On the top of Figure 2.4 there are various tools running which communicate with the central component of the monitoring system. This component is called the *node distribution unit* (NDU). The NDU is intended to analyze each request issued by a tool and split it into pieces that can be processed locally by the local monitors on the nodes involved. The NDU must also assemble the partial answers which are received from the local monitor processes into a global reply sent back to the requesting tool. Addition and removal of nodes is also detected and handled by the NDU.

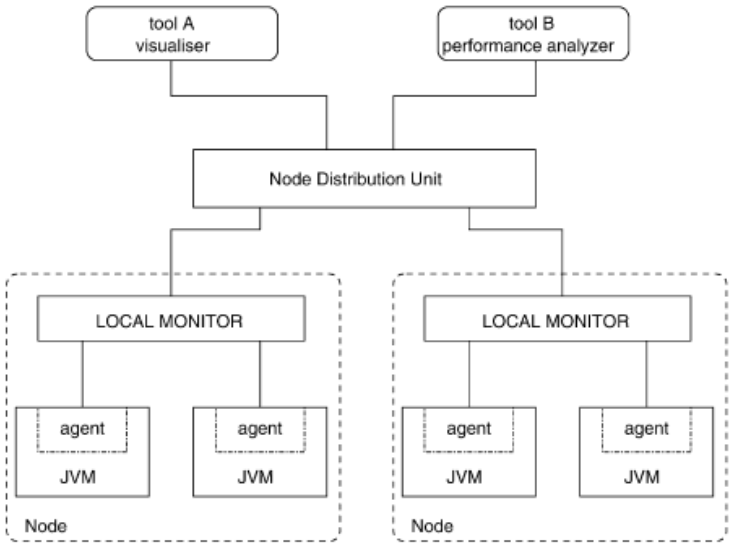


Figure 2.4: Architecture of the monitoring environment.

The distributed part of the monitoring system consists of multiple *node local monitors* (NLM), one per node of the target system and *Java virtual machine local monitors* (JVMLM), one for each JVM process (there can be multiple JVMs on one node). The NLM processes control the JVM processes via agents (JVMLMs) and the operating system interfaces. The NLM offers a server interface that is similar to the monitoring interface, with the exception that it only accepts requests that can be handled locally. The NLM is responsible for the cooperation with JVMLMs. In addition, it gathers information from the outside of the JVM process. The JVMLM is an agent that is embedded in the virtual machine process. The agent is responsible for the execution of the requests received from the NLM. Its implementation depends on the virtual machine native interfaces that provide low-level mechanisms for interactive monitoring of the JVM. In order to make the monitoring system widely independent from a concrete JVM implementation, it is required to build JVMLMs that will use JVM interfaces like JVMPI (Java Virtual Machine Profiler Interface), JNI (Java Native Interface), JVMDI (Java Virtual Machine Debug Interface) or Java bytecode instrumentation.

### 2.3.2 J-OCM

*J-OCM*[18] is a J-OMIS compliant monitoring system, a Java-oriented extension to OCM[25] (OMIS Compliant Monitoring system) that extends

the functionalities of the OCM, via adding new software components and adapting existing ones (see Figure 2.5).

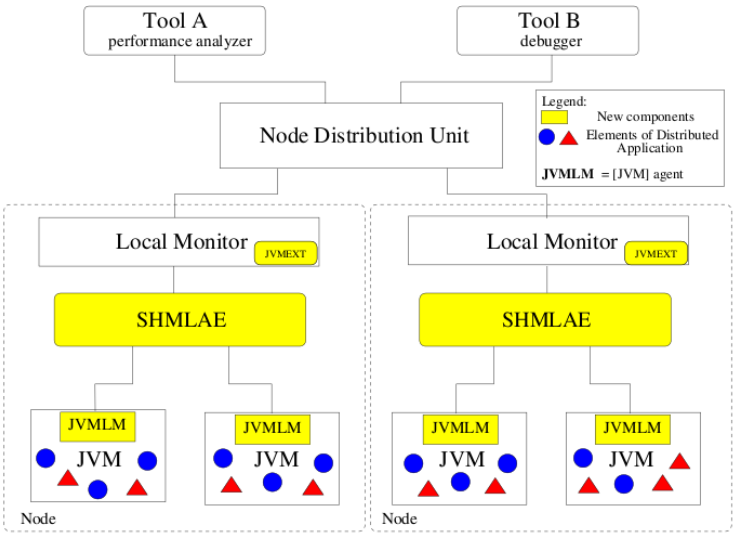


Figure 2.5: Architecture of J-OCM.

To support the monitoring of Java applications, the LM's extension, *JVMEXT*, provides new services defined by J-OMIS, which control JVM via agents. JVMEXT is linked to LMs as a dynamically linked library at run-time using the *dlopen* interface, whenever the tool issues a service request to JVMEXT.
The *Java Virtual Machine Local Monitor* (JVMLM) is an agent embedded into a JVM process, as seen in the section 2.3.1.
The *Shared Memory based Local Agents Environment* (SHMLAE) is a communication layer to support cooperation between agents and the LM. This allows the components involved in communication to find each other during start-up and notify about their existence.

### Request Processing

The monitoring infrastructure allows the tool to see the whole monitored application as a set of distributed objects and the monitoring system as a higher-level software layer (middleware) that provides a standardized interface to access those objects, regardless of implementation details, like hardware platform or software language.

To deal with the distributed target system the J-OCM will be considered as a distributed system, which has to usually comprise additional architectural elements[13]:

- **Interface definition** of a remote object (e.g. methods, data types), written in an *Interface Definition Language* (IDF) file called *registry*
- **Stub and Skeleton** of the object, based on the *Proxy Design Pattern* where the object is represented by another object (the proxy), in order to control access to the object. The monitored objects are identified by tokens which refer to the proxy object. The proxy is a representation of the real object in the monitoring system. The object proxy contains all information that is needed to deliver the tool's requests to the JVM agent (JVMLM) which directly accesses the JVM[18]. The JVM agent acts as a skeleton, while the remote proxy which is embedded into the JVM is a platform dependent native library. The agent transforms a call and parameters received from the LM into the format required by one of interfaces used to interact with JVM.
- **Object manager and registration/naming service**. The Object Manager routes the calls issued by the client to the propoer object on the server and the results back to the client. The registration/naming service acts as an intermediary layer between the object client and the object manager. Once an interface to the object has been defined, an implementation of the interface needs to be registered with the naming service so that the object can be accessed by clients using the object's name. NDU and LMs can be classified as an object manager and provide operations similar to the naming service that is present in distributed systems.

### Event Handling

In order to follow the idea of event-based monitoring both the LM and the JVMLM must support the event notification.
JVM notifies several internal events to the JVMLM, using JVMPI and JVMDI. These events are fired by changes in the state of Java threads, like (started, ended, blocked on a locked monitor ), the beginning/ending of an

invoked method, class loading operations, object allocation/deallocation, and the beginning/ending of JVM garbage collection, exception throwing, etc.
To support the interactive observation of the target system, all events must be processed by the JVM agent, while the agent sends the events to the LM selectively, to avoid too much overhead on the LM. This is based on a filtering mechanism (implemented by a filter table) introduced into the JVM agent, which selects which events should be sent to the LM.
The J-OCM, as an adaptation of the OCM for Java applications, extends the *event tree* of the OCM by its own subtree. The new event hierarchy, shown in Figure 2.6, consists of three types of event classes[12].

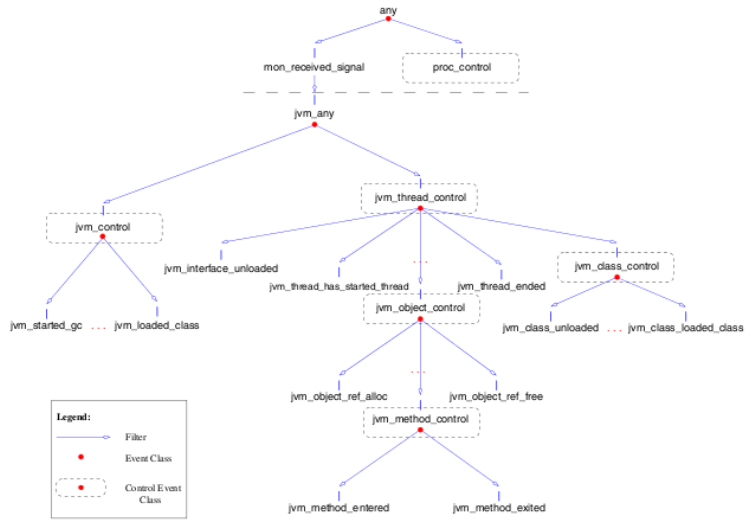


Figure 2.6: Hierarchy of J-OCM events.

The `jvm_any` is the "root" of Java related event classes and is triggered whenever any of JVMs registered in the monitoring system generates an event. The `jvm_any` is a *Singleton* , the event class that has only one instance to provide the global point of access to it in the Local Monitor. All other event classes, which relate to other object types specified in J-OMIS, e.g. jvms, threads, classes, etc. are derived from this one using filters.
The `jvm_control`, `jvm_thread`, `jvm_class`, `jvm_object`, `jvm_method` control classes represent *abstract object control event* classes.
Leaves, those elements of the tree which do not have children, represent the events which are defined by the interface specification, i.e. J-OMIS. J-OMIS classifies event services based on the categories of elements of the Java program

architecture the services operate on. The J-OCM event tree follows this classification, similarly the control services group the event services operating on a type of token, e.g. thread, JVM. But some event services have been moved over in the event hierarchy in order to recognize the situation where the event took place. The information which is needed to determine whether an event matches a given event class is the location (context) of the event, e.g. the event service `jvm_method_entered` indicates when a given method is called, i.e. is needed t o specify the most dynamic elements of a Java application execution, e.g. JVM, thread, object because they determine the context of the event occurrence, i.e. an event may occur on a particular JVM, in a thread, and refer to an object.
The presented shape of the J-OCM events tree (Figure 2.6) allows to narrow the event detection and to simplify extending the tree by new events.

# Chapter 3

# The SelfLetMonitor framework

## 3.1 The SelfLet framework

In this section the (either general and internal) structure of the SelfLet framework, an implementation of an Autonomic System[21], is presented; this is necessary in order to understand all the requirements that have led to the realization of the Monitoring System in the current form, both in term of the tools chosen, the design principles used and the in-deep implementation of some of its features.

### 3.1.1 SelfLet Conceptual Model

All the SelfLet framework is founded on the concept of SelfLet: a SelfLet is a self-sufficient piece of software which is situated in some kind of logical or physical network, where it can interact and communicate with other SelfLets[17].
According to the SelfLet's model, every SelfLet is defined in terms of the *offered Services* and *Autonomic Policies*[14].
The services represent high level tasks to accomplish and their implementation, called *behaviour*, is programmed using UML state diagrams[1]. A service can have more than one implementation and one of these is the *default behaviour*. There are two kinds of behaviours: *complex* (i.e. with a generic number

[1]Currently ArgoUML[2] is used to design the implementation of the services

of internal states) and *elementary*[2] (i.e. with a single internal state). The services can be offered in different ways: *Can Do*, *Can Teach*, *Know Who Can Do* and *Know Who Can Teach*. Any combination of these "offer modes" is allowed. The relationships characterizing services, behaviours and policies of SelfLets are presented in Figure 3.1.

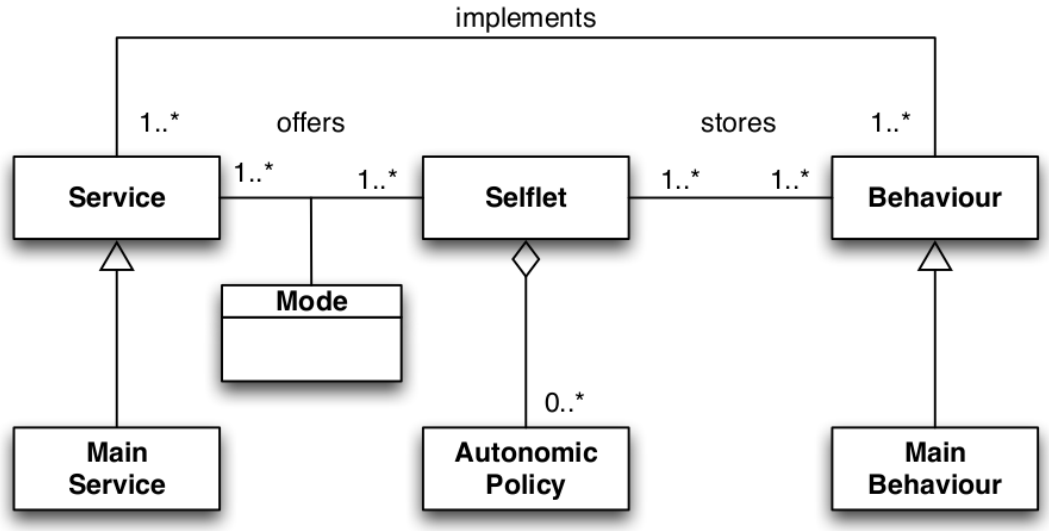


Figure 3.1: The SelfLet framework

Each SelfLet has a list of known providers for each service required, so, when there is a need for a given service, it selects one of these providers according to a given *policy* in order to ask it for the service. The SelfLets in a network cooperate to keep such lists always updated. A *main service* represents the objective of the SelfLet and is proactively executed after the SelfLet has been initialized.
The *Autonomic policies* define reactions to abnormal situations that can occur during the life-time of the SelfLet. The SelfLet offers to policy writers a list of actions that enable the transformation of several aspects of the SelfLet itself:

- changing the way a service is offered or asked;
- install action of a new service;
- install action new abilities;

[2]More specifically, elementary behaviours execute the actual computation which is contained in OSGi[30] bundles called *abilities*.

- switch from a service implementation to another;
- modification of a given behaviour.

### 3.1.2 The SelfLet Architecture

The components which constitute a SelfLet are shown in the architecture presented in Figure 3.2.

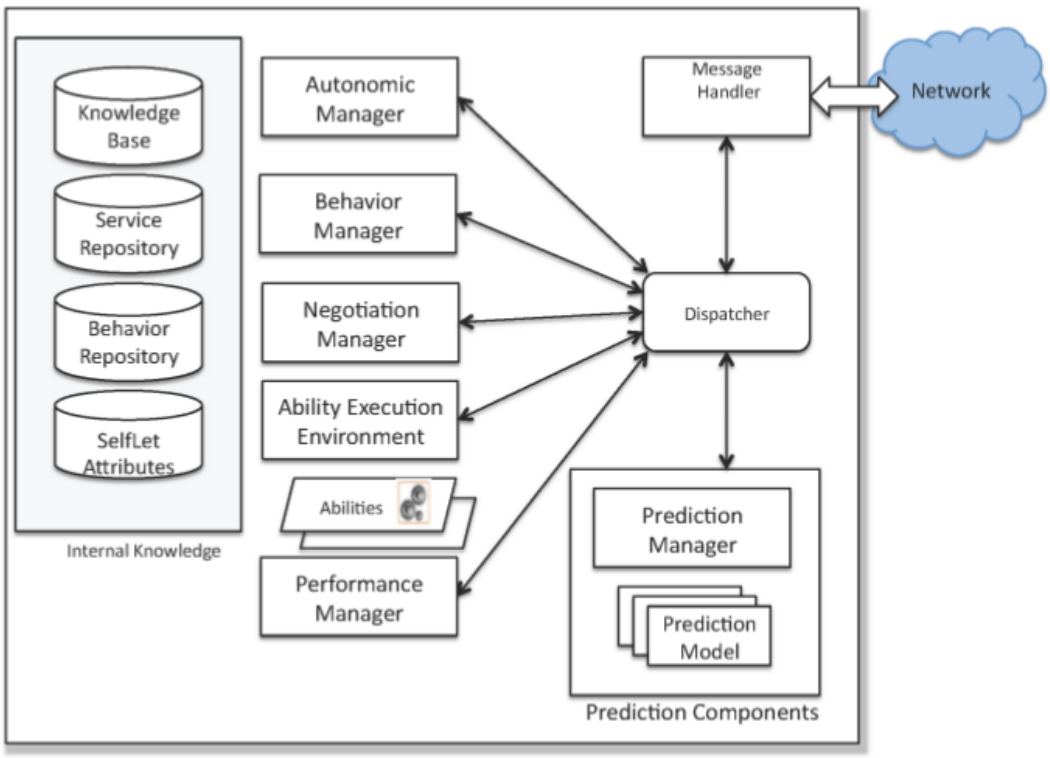


Figure 3.2: Internal Architecture of a SelfLet.

The central part of the SelfLet architecture is the *Autonomic Manager*, which is responsible for the evolution of a SelfLet depending on the set of *Autonomic Policies* it has installed. The Autonomic Manager is implemented through Drools[3]: every time a request for a service is raised, the Autonomic Manager triggers the execution of a rule that verifies if the service is locally available or not and, if not, asks the *Negotiation Manager* to retrieve the Service and negotiate with the corresponding SelfLet a proper offer mode.
The *Behaviour Manager* controls the execution of the SelfLet's behaviour.
The *Internal Knowledge* is composed of four parts: Knowledge Base (contains

any kind of information needed by any of the SelfLet components), Service Repository (contains the services the SelfLet can offer to itself or to other SelfLets), Behaviours Repository (contains all the behaviours specifications the SelfLet is able to run) and Attribute Repository (stores descriptions about the SelfLet).
The *Ability Execution Environment* executes the Abilities, that can be activated as part of behaviours.
The *Performance Manager* offers a unified interface to monitor the execution of internal services.
A *Dispatcher* is in charge of receiving subscriptions for events and event publications between all the SelfLets: it asynchronously delivers all published events to those components that have subscribed to them. The Dispatcher is connected to the *Message Handler*, which manages the communication with the external environment and with other SelfLet through the REDS[16] middleware.

## 3.2 Conceptual Tools

This section presents some of the main conceptual tools that are used in order to realize the monitoring framework described in the next sections. Other design pattern as the *Singleton Pattern*, the *Builder Pattern*, the *Factory Method Pattern* or the *Observer Pattern*[19] are frequently used along the code of the framework, but doesn't represent a differentiation factor from all the other similar projects, so they were not explained in the current section: they were only some of the sufficient requirements for the realization of any project that aim to be easily maintained, understood and extended.

### 3.2.1 Aspect Programming

The cross-cutting problem is the basis of aspect-oriented programming (AOP). AOP is a methodology that provides separation of crosscutting concerns by introducing a new unit of modularization: *aspect*. Each aspect focuses on a specific crosscutting functionality. An aspect weaver composes the final system by combining the core classes and crosscutting aspects through a process called

*weaving*. Thus, AOP helps to create applications that are easier to design, implement, and maintain [22].

### Without AOP

Typically,the implementation of crosscutting concerns using OOP alone implies the adding the code needed for each crosscutting concern in each module, as shown in Figure 3.3. This figure shows how different modules in a system implement both core concerns and crosscutting concerns.

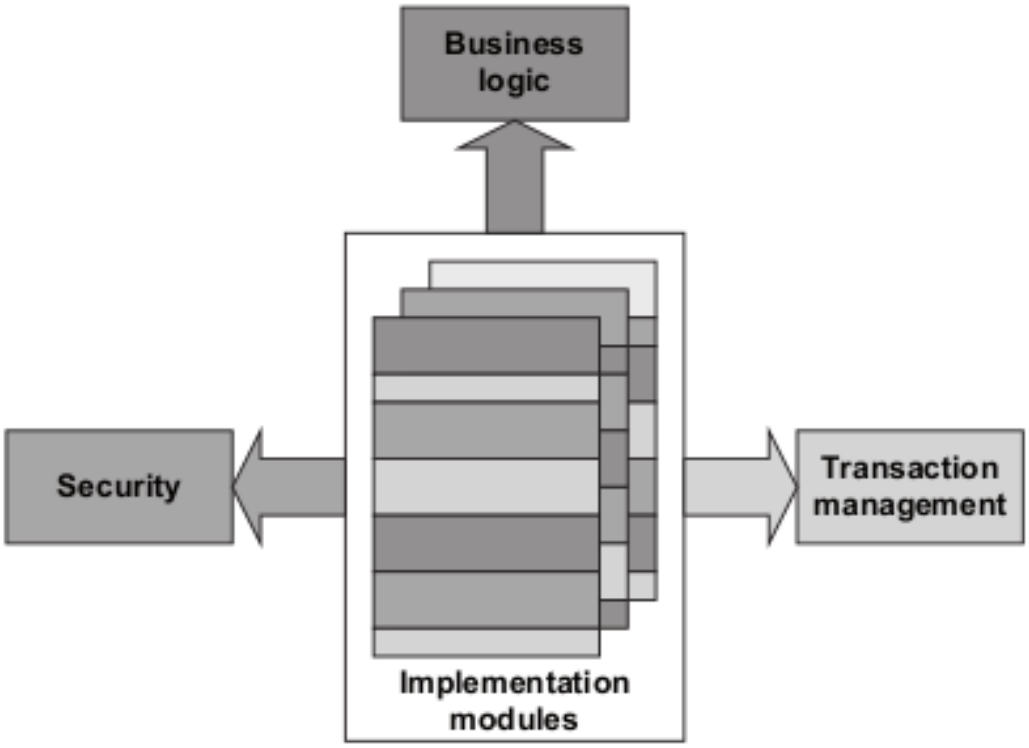


Figure 3.3: A system as a composition of multiple concerns.

Although the details will vary, the Figure 3.3 shows a common problem many developers face: a conceptual separation exists between multiple concerns at design time, but implementation mixes them together, breaking the *Single Responsibility Principle* (SRP) and thus the Open/Close principle. The overall consequence is a higher cost of implementing features and fixing bugs. With conventional implementations, core and crosscutting concerns are tangled

in each module. Furthermore, each crosscutting concern is scattered in many modules.

**Code tangling** Code tangling is caused when a module is implemented to handle multiple concerns simultaneously. Developers often consider concerns such as business logic, performance, synchronization, logging, security, monitoring and so forth when implementing a module. Obviously, such problems can be fixed within the bounds of OOP, but this leads to the simultaneous presence of elements from each concern's implementation and results in code tangling.
Another way to look at code tangling is to use the notion of a multidimensional concern space. If the application requirements are projected onto a multidimensional concern space, with each concern forming a dimension, all the concerns are mutually independent and therefore can evolve without affecting the rest. The Figure 3.4 shows a multidimensional concern space which collapses into a one-dimensional implementation space. Because the implementation space is one-dimensional, its focus is usually the implementation of the core concern that takes the role of the dominant dimension; other concerns then tangle the core concern. Although the individual requirements can be naturally separated into mutually independent concerns during the design phase, OOP alone doesn't permits the separation in the implementation phase.

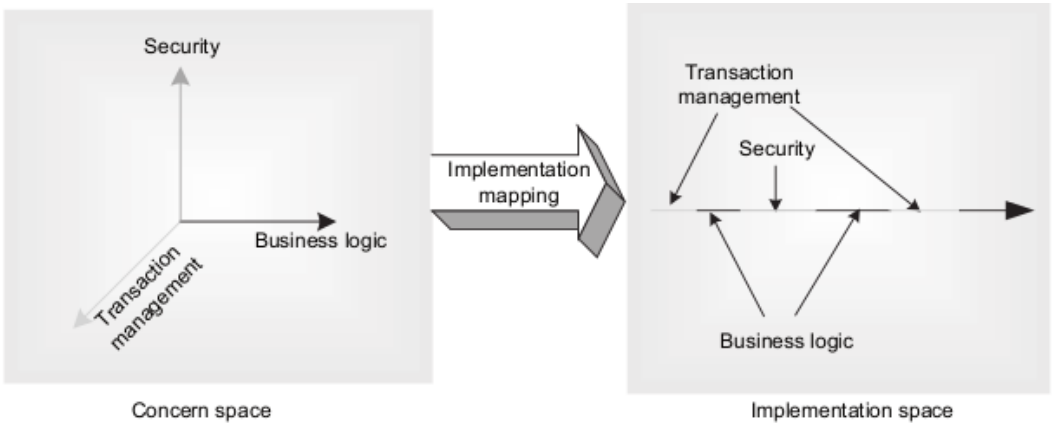


Figure 3.4: N-dimensional concern space using a one-dimensional language.

**Code scattering** Code scattering is caused when a single functionality is implemented in multiple modules. Because crosscutting concerns, by definition, are spread over many modules, related implementations are also scattered

over all those modules.
Code tangling and code scattering together impact software design and development in many ways: poor traceability, lower productivity, lower code reuse, poor quality, and difficult evolution. All of these problems lead to search for better approaches to architecture, design, and implementation. Aspect-oriented programming is one viable solution.

#### Modularizing with AOP

In OOP, the core concerns can be loosely coupled through interfaces, but there is no easy way to do the same for crosscutting concerns. This is because a concern is implemented in two parts: the server-side piece and the client-side piece. OOP modularizes the server part quite well in classes and interfaces. But when the concern is of a crosscutting nature, the client part (consisting of the requests to the server) is spread over all the clients. The terms server and client are used here in the classic OOP sense to mean the objects that are providing a certain set of services and the objects using those services.
The fundamental change that AOP brings is the preservation of the mutual independence of the individual concerns. Implementations can be easily mapped back to the corresponding concerns, resulting in a system that is simpler to understand, easier to implement, and more adaptable to changes.

### 3.2.2 Design Patterns

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. The design should be specific to the problem at hand but also general enough to address future problems and requirements. Experienced object-oriented designers state that a reusable and flexible design is difficult if not impossible to get "right" the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.
Yet experienced object-oriented designers do make good designs. Meanwhile new designers are overwhelmed by the options available and tend to fall back on non-object-oriented techniques they've used before. It takes a long time for novices to learn what good object-oriented design is all about. Experienced designers evidently know something inexperienced ones don't.
One thing expert designers know not to do is solve every problem from first

principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently, these solutions constitute recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.
Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems [19].

### Immutable Objects

An immutable class is simply a class whose instances cannot be modified. All of the information contained in each instance is provided when it is created and is fixed for the lifetime of the object. There are many good reasons for this: Immutable classes are easier to design, implement, and use than mutable classes. They are less prone to error and are more secure[28].
An immutable class follows these five rules [11]:

1. Any methods that modify the object's state aren't provided.

2. The class can't be extended. This prevents careless or malicious subclasses from compromising the immutable behavior of the class by behaving as if the object's state has changed. Preventing subclassing is generally accomplished by making the class final, but there is an alternative discussed later.

3. All fields are declared *final*. It is necessary to ensure correct behavior if a reference to a newly created instance is passed from one thread to another without synchronization, as spelled out in the *memory model*[28].

4. All fields of the class are *private*. This prevents clients from obtaining access to mutable objects referred to by fields and modifying these objects directly. While it is technically permissible for immutable classes to have public final fields containing primitive values or references to immutable

objects, it is not recommended because it precludes changing the internal representation in a later release.

5. Any mutable components have exclusive access. If the class has any fields that refer to mutable objects, clients of the class cannot obtain references to these objects.

Immutable objects are simple. An immutable object can be in exactly one state, the state in which it was created. If all constructors establish class invariants, then it is guaranteed that these invariants will remain true for all time, with no further effort by the programmer who uses the class. Mutable objects, on the other hand, can have arbitrarily complex state spaces. If the documentation does not provide a precise description of the state transitions performed by mutator methods, it can be difficult or impossible to use a mutable class reliably.
Immutable objects are inherently thread-safe; they require no synchronization [28]. They cannot be corrupted by multiple threads accessing them concurrently. This is the easiest approach to achieving thread safety. In fact, no thread can ever observe any effect of another thread on an immutable object. Therefore, immutable objects (and their internals) can be shared freely. Immutable classes should take advantage of this by encouraging clients to reuse existing instances wherever possible. One easy way to do this is to provide public static final constants for frequently used values.
An immutable class can provide static factories that cache frequently requested instances to avoid creating new instances when existing ones would do. Using such static factories causes clients to share instances instead of creating new ones, reducing memory footprint and garbage collection costs. Opting for static factories in place of public constructors when designing a new class gives the flexibility to add caching later, without modifying clients.
Immutable objects make great building blocks for other objects, whether mutable or immutable. It's much easier to maintain the invariants of a complex object if its component objects will not change underneath it. A special case of this principle is that immutable objects make great map keys and set elements: once they're in the map or set their values never change.
The only real disadvantage of immutable classes is that they require a separate object for each distinct value. Creating these objects can be costly, especially

if they are large. To summarize, classes should be immutable unless there's a very good reason to make them mutable. Immutable classes provide many advantages, and their only disadvantage is the potential for performance problems under certain circumstances.

### Decorators

Inheritance is a powerful way to achieve code reuse, but it is not always the best tool for the job. Used inappropriately, it leads to fragile software. It is safe to use inheritance within a package, where the subclass and the superclass implementations are under the control of the same programmers. It is also safe to use inheritance when extending classes specifically designed and documented for extension. Inheriting from ordinary concrete classes across package boundaries, however, is dangerous.
Unlike method invocation, inheritance violates encapsulation [29].
In other words, a subclass depends on the implementation details of its superclass for its proper function. The superclass's implementation may change from release to release, and if it does, the subclass may break, even though its code has not been touched. As a consequence, a subclass must evolve in tandem with its superclass, unless the superclass's authors have designed and documented it specifically for the purpose of being extended.
A related cause of fragility in subclasses is that their superclass can acquire new methods in subsequent releases. Suppose a program depends for its security on the fact that all elements inserted into some collection satisfy some predicate. This can be guaranteed by subclassing the collection and overriding each method capable of adding an element to ensure that the predicate is satisfied before adding the element. This works fine until a new method capable of inserting an element is added to the superclass in a subsequent release. Once this happens, it becomes possible to add an "illegal" element merely by invoking the new method, which is not overridden in the subclass. Both of the above problems stem from overriding methods.
There is a way to avoid all of the problems described earlier. Instead of extending an existing class, the new class must have a private field that references an instance of the existing class. This design is called *composition*[11] because the existing class becomes a component of the new

one. Each instance method in the new class invokes the corresponding method on the contained instance of the existing class and returns the results. This is known as forwarding, and the methods in the new class are known as forwarding methods. The resulting class will be with no dependencies on the implementation details of the existing class. Even adding new methods to the existing class will have no impact on the new class. The implementation will be broken into two pieces, the class itself and a reusable forwarding class, which contains all of the forwarding methods and nothing else. Besides being robust, this design is extremely flexible. Unlike the inheritance-based approach, which works only for a single concrete class and requires a separate constructor for each supported constructor in the superclass, the wrapper class can be used to *decorate* any implementation of the choosen interface and will work in conjunction with any preexisting constructor.
The class extending the reusable forwarding class is known as a wrapper class because each instance contains ("wraps") another instance of the right interface. This is also known as the *Decorator pattern* [19], because the wrapper class "decorates" the wrapped class by adding functionalities. Sometimes the combination of composition and forwarding is loosely referred to as *delegation*. Technically, it's not delegation unless the wrapper object passes itself to the wrapped object.

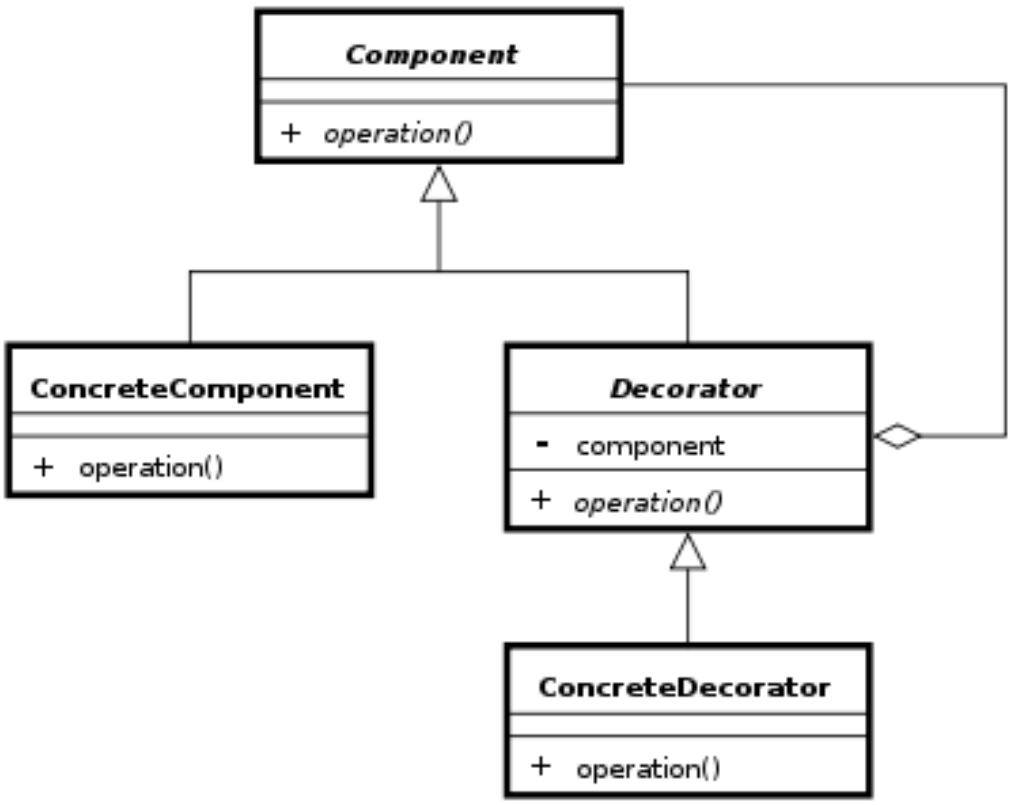


Figure 3.5: Decorator UML class diagram.

The disadvantages of wrapper classes are few. One caveat is that wrapper classes are not suited for use in callback frameworks, wherein objects pass self-references to other objects for subsequent invocations ("callbacks"). Because a wrapped object doesn't know of its wrapper, it passes a reference to itself (this) and callbacks elude the wrapper. This is known as the SELF problem [23] and in that case is preferred to use inheritance to maintain the right "identity" of the class to which the client invokes the functionalities.
Inheritance is appropriate only in circumstances where the subclass really is a subtype of the superclass. In other words, a class B should extend a class A only if an "is-a" relationship exists between the two classes.

### Active Objects

Active objects are very similar to traditional objects (also called passive objects in this context for a better differentiation). They have private fields and provide methods to operate on this data. The only important difference lies in the fact that each active object runs in its own thread of control and

that invoked methods do not block the caller but are executed asynchronously. Methods of active objects are always executed in a single thread and run sequentially with respect to each other, i.e. it is not possible that two method calls of one particular active object run at the same time. This simple condition automatically guarantees that the implementation of an active object does not require any additional mechanisms to ensure thread-safety for this particular object.

To understand how active objects work, it is important to differentiate between their public interface and their internal operation. The public interface of an active object is often called *proxy* and the internals are represented by a so called *servant* or *implementation*. The proxy is responsible for accepting method calls or requests from clients, other objects (passive or active) which utilize an active object. Public method requests to an active object and possible arguments are marshaled or converted into messages by the proxy and added to a message queue. A special object usually called dispatcher or scheduler which runs in the context of the active object thread dequeues and processes each incoming message and then invokes the actual methods of the servant. Possible results of asynchronous methods are returned in the form of *Future* objects. Figure 3.6 shows the corresponding message sequence chart for a typical request from a client to an active object.

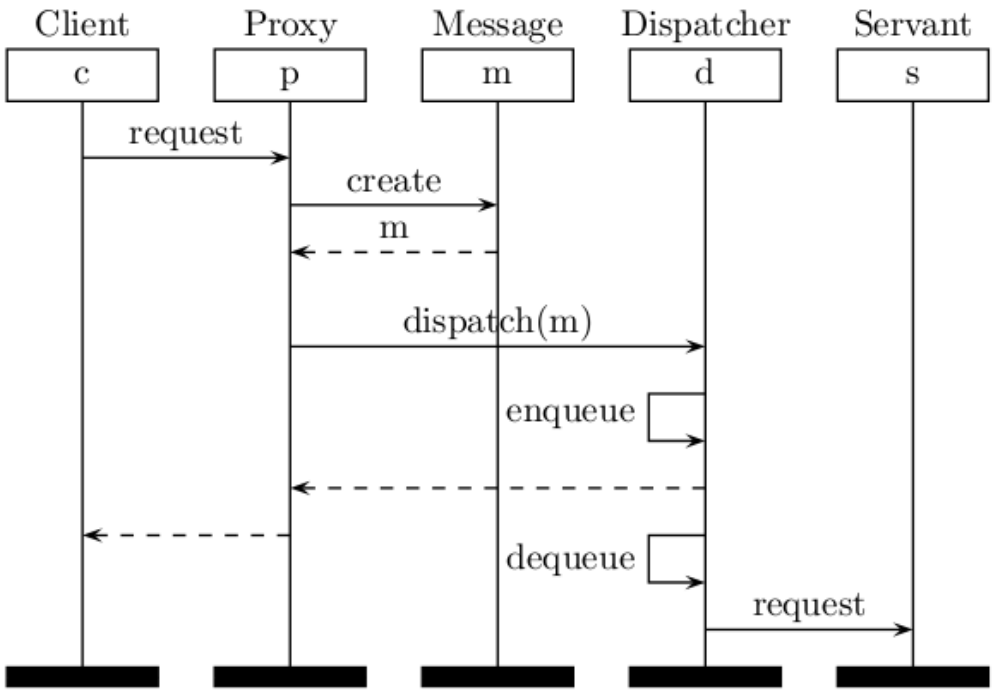


Figure 3.6: Active Object Behavior.

Two important things can be observed in this chart. First, it should become clear that the client does only interact with the public proxy interface of the active object and is not (or does not need to be) aware of the underlying message-passing semantics. Interaction and communication with the servant is achieved indirectly with request arguments and possible Future return values. Secondly, the dequeue and request operations on the lower right of the chart are processed asynchronously by the dispatcher while the client on the left can complete other tasks in the meantime.
In summary, active objects are characterized by the combination of the following three fundamental properties [28]:

1. **Message-based** Requests and possible arguments to an active object are converted into messages, forwarded to and eventually executed by the private active object implementation. Result messages are modeled as Future objects.

2. **Asynchronous** Requests to an active object are executed asynchronously in a private thread of control and only block the caller if the result is requested before it is available. Asynchronous execution of

requests is the property which lets active objects add concurrency and multi-threading to applications.

3. **Thread-safe** Active objects are inherently thread-safe by sequentially dequeuing and processing enqueued requests and always executing them in a single thread of control. The thread-safety of active objects largely removes the need for manual locking and mutual exclusion mechanisms.

## 3.3 Technologies adopted

In this section is presented the main "physical tool" that represents the core of the monitoring framework that i realized: through it, the monitoring system (the local part, especially) obtains a simple, well known and clear tool to capture all the informations choosen by the user of the framework, without modifying the original source of the monitored system, as described in the next sections.
Other "tools" were used to achieve all the main qualities (section 2.2.2) of a fully-functional Distributed Monitoring System; REDS[16] is already part of the monitored system (the SelfLet Framework) and were adapted to fullfill the needs of comunication (in particular, for the consistency of the messages passed) between the node of the monitoring system, without affecting the comunication channel of the monitored system.

### 3.3.1 AspectJ

AspectJ is the most complete implementation of the AOP model, supporting all its elements [22]. This section examines how AspectJ maps each model element into program constructs. Note that AspectJ offers two syntax choices: traditional and *@AspectJ*. This section uses the traditional syntax to study these building blocks. The crosscutting constructs in the AOP model can be classified as *common crosscutting constructs* (join point, pointcut, and aspect), *dynamic crosscutting construct* (advice), and *static crosscutting constructs* (inter-type declarations and weave-time declarations). These constructs form the building blocks of AspectJ.

### Common crosscutting constructs

AspectJ supports a few common constructs consisting of the join point, the pointcut, and the aspect. These constructs can be used with both dynamic and static crosscutting. In AOP, and therefore in AspectJ, join points are the places where the crosscutting actions take place.

**POINTCUT** A pointcut is a program construct that selects join points and collects context at those points. For example, a pointcut can select a join point that is an execution of a method. It can also collect the join-point context, such as the *this* object and the arguments to the method. It's possible to name a pointcut so that other programming elements can use it (and so that programmers can understand the intention behind the pointcut).

**ASPECT** The aspect is the central unit in AspectJ, in the same way that a class is the central unit in Java. It contains the code that expresses the weaving rules for both dynamic and static crosscutting. Additionally, aspects can contain data, methods, and nested class members, just like a normal Java class.

### Dynamic crosscutting construct: advice

AspectJ's dynamic crosscutting support comes in the form of advice. Advice is the code executed at a join point selected by a pointcut. Advice can execute before, after, or around the join point. The body of advice is much like a method body; it encapsulates the logic to be executed upon reaching a join point.
While dynamic crosscutting alters the program behavior, static crosscutting alters the programs structure.

### Static crosscutting constructs

Static crosscutting comes in the form of inter-type and weave-time declarations.

**INTER-TYPE DECLARATION** The inter-type declaration (ITD) is a static crosscutting construct that alters the static structure of the classes,

interfaces, and aspects in the system. In an ITD, one type (an aspect) declares the structure for the other types (classes, interfaces, and even aspects), hence the name. Member introduction is another form of ITD that offers a way to add new methods and fields to other types. ITD also offer a way to annotate program elements and deal with checked exceptions in a systematic manner. An important form of static crosscutting allows detecting and flagging the presence of join points, matching a pointcut during compilation.

**WEAVE-TIME DECLARATION** The weave-time declaration is another static crosscutting construct that allows to add weave-time warnings and errors when detecting certain usage patterns. Often, weaving is performed during compilation; therefore, these warnings and errors are issued when the classes are compiled. The weaver will report warnings when it detects the specified conditions along with other compile-time warnings such as use of a deprecated method.

### Weaving mechanisms

A weaver needs to weave together classes and aspects so that advice gets executed, inter-type declarations affect the static structure, and weave-time declarations produce warnings and errors. AspectJ offers three weaving models:

- Source weaving
- Binary weaving
- Load-time weaving

Regardless of the weaving model used, the resulting execution of the system is identical. The weaving mechanism is also orthogonal to the AspectJ syntax used; any combination of weaving mechanism and AspectJ syntax will produce identical results.

**Source weaving** In source weaving, the weaver is part of the compiler. The input to the weaver consists of classes and aspects in source-code form. The aspects can be written in either the traditional syntax or the @AspectJ syntax.

The weaver, which works in a manner similar to a compiler, processes the source and produces woven byte code. The byte code produced by the compiler is compliant with the Java byte-code specification, which any standard compliant VM can execute. Essentially, when used in this manner, ajc replaces javac. But unlike javac, ajc requires that all sources be presented together.

**Binary weaving** In binary weaving, input to the weaver (classes and aspects) is in byte-code form. The input byte code is compiled separately using the Java compiler or the AspectJ compiler. An extension of binary weaver is load-time weaving.

**Load-time weaving** A load-time weaver takes input in the form of binary classes and aspects, as well as aspects and configuration defined in XML format. A load-time agent can take many forms: a Java VM Tools Interface (JVMTI) agent, a classloader, or a VM-specific class preprocessor, which weaves the classes as they're loaded at run-time into the VM. The Figure 3.7 explains better the process.

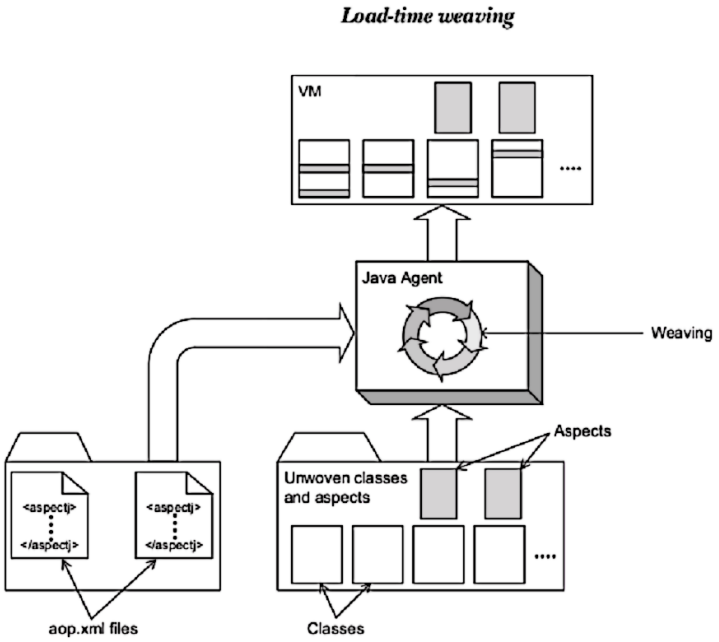


Figure 3.7: Load-time weaving schematic.The *aop.xml* files provide information about the aspects and classes participating in load-time weaving. The weaver intercepts loading of any class to weave in appropriate aspects.

## 3.4 The SelfLetMonitor Framework

In this section is presented the structure of the *SelfLetMonitor framework* , an implementation of an on-line Distributed Monitoring System[27], designed in order to achieve the needs of a lightweight, extendible, robust and accurate debugging instrument for the SelfLet framework.

### 3.4.1 Architecture

As described in the previous sections, a monitoring system that operates over a distributed system is itself a distributed system, divided in smaller local monitoring nodes attached to the nodes of the monitored application (as suggested by the J-OMIS approach[26]).
The local monitored systems have to intercept a customizable set of events (and/or data modifications) of the application and comunicate to the monitoring tools all the informations collected in a real-time-fashion, in order to represent in any choosen form (graphically, textually and so on) the overall state of the system. One of the most important requirements of a monitoring system is its "plugin" form: the capability to be completely decoupled in term of development and deployment cycle. For this reason, the SelfLetMonitor system uses *@AspectJ* with Load-time weaving (section 3.3.1) for the front-end part of the local monitors. Indeed, a developer doesn't have to learn a new language (the original AspectJ form): the java classes used to interact with the SelfLet Framework are "decorated" with simple annotations that instruct a java agent (the AspectJ weaver) to treat the classes annotated as Aspects and not as simple Java Classes. This solution impose an important constraint on the framework; the needs of a JVM (Java virtual Machine) updated to the 1.5 version. In any case, considering that in the framework are used other useful characteristics of such version of the JVM (better cuncurrent API, generics support and others), it doesn't represent a heavy-weight constraint in front of the achieved improvements. Over the classes "@AspectJ-annotated" there is a thread-safe wrapper layer that translate the informations collected by the aspects in an immutable form that an Active Object that decorates a REDS[16] dispatcher uses to send at the remote tool responsible to show the state of the overall selfLet network. The Immutable form of the data created by the wrap-

pers and sended by the "active sender", permits a simple mechanism of instance sharing either in the local nodes (the wrappers and the "Active Sender" and their respective threads can share freely all the information collected, without corrupting it[28]) and in the remote tool, as explained in the next sections.
A conceptual representation of the architecture of the monitoring system is presentend in Figure 3.8.

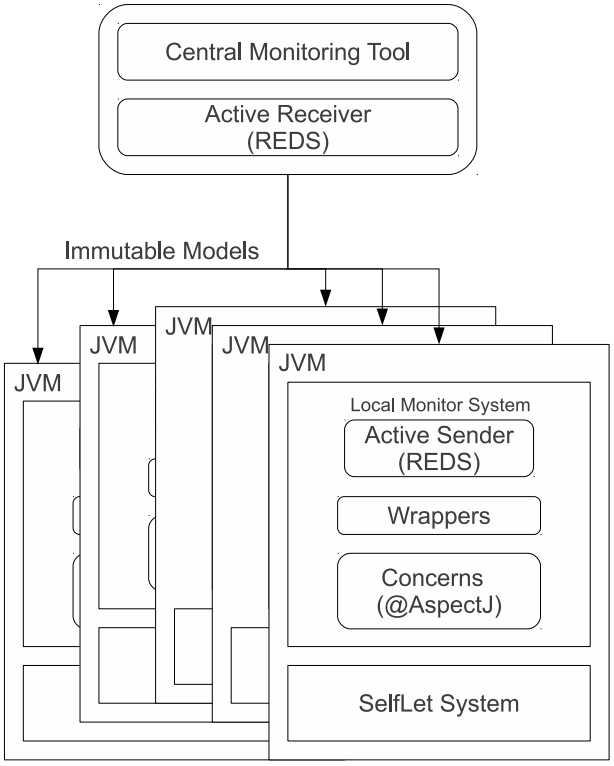


Figure 3.8: Overall architecture of the SelfLetMonitor framework.

### 3.4.2 Under the hood

Increasing the detail of the view of the system and going inside its current implementation is possible to consider 3 important parts that constitute it:

1. the **SelfLetMonitor API**'s: the API used either by the remote tools and the local monitoring nodes to create and send the informations through the comunication channel, decoupled by the REDS messagging system.

2. the **SelfLetClientMonitor**: i.e. the local monitoring part, relies on the interfaces exposed by the SelfLetMonitor API's and on the interfaces

of che monitored concerns. Collects all the information through a layer of aspects, which, in turn, use the interfaces provided by the wrapping layer to create the messages. Through another crosscutting concern on the monitoring system itself, all the relevant creations of messages will be captured and delivered to the remote tools.

3. the **SelfLetServerMonitor**: i.e. the tools that realize some kind of aggregation (e.g. a Swing user interface) between all the messages sended by the local monitoring systems. Relies only on the model and comunication channels provided by the common monitoring API's.

### SelfLetMonitorAPI

The core part of the architecture are the APIs, divided in two main packages, *model* and *comunication*: the former provides all the primitives to manipulate (through a Builder pattern to ensure the thread-safety of the creation[11]) the immutable models (shown in Figures A.2, A.3 , A.4) used by the local monitors and the remote tools, while the latter permits to establish a general channel in which the models, encapsulated in proper messages (Figure A.1), could travel and be retrieved in different manners.
All the comunication part of the APIs hiddens completely its real implementation through the Bridge pattern[19], permitting to the users of the APIs only to extends some well-known points, a group of safe-extendable classes[3], as shown in Figures A.5, A.6, A.7. This defensive approach is used in order to prevent the violations of encapsulation caused by the wrong usage of the inheritance mechanism[11].

### SelfLetClientMonitor

The local monitors are structured in two important parts: *concerns* and *wrappers*. The first is responsible to analyze[4] the crosscutting choosen informations of the running monitored system (e.g modifications in the internal knowledges,

[3]All the classes named with the prefix of *Forwardable* were let visible to the API user, and are safely-extendable.

[4]Not only the monitored system. But there are aspects (Figure B.1) used to monitor both the on-line local monitoring system and the monitored system for off-line performance evaluation purposes.

running Services or SelfLet properties) through the expressivity of the AspectJ language inserted in proper Java annotations, while the second one is used by the "aspects" to decoupled the arise of some kind of event in the SelfLet engine from the creation of the right immutable model.
The aspects layer presents, for all the working aspects, a skeletal (i.e. abstract) class that represent only the abstract concern that the local monitor is designed to capture. So, with this "skeletal aspect" the local monitor system could easily evolve to permit new kind of interactions with che monitored system (e.g. becoming a debug tools capable to modify the flow of informations in the SelfLet engine [5]). All the concrete aspects refers to the right wrapper only by its interface, so the realization of new implementations of the wrapper would be simple and without effects on the aspect layer (the Bridge pattern helps in this). All the current wrapper implementations use factories to be instantiated and the only wrappers type hierarchy visible outside the packages are represented by public interfaces, as shown in figures B.2, B.3, B.4, B.5, B.6).
Instead, internally, the wrappers layer uses a combination of decorators[11][19] and normal inheritance from skeletal (i.e. abstract) classes to provide all the services promised by their interfaces and to hide externally the others required by the classes belonging to the same layer (e.g. the Observer/Observable relationship between classes of the same layer). Basically, all the classes capable to manage any new service without encountering the SELF problem[23] use the Decorator pattern to provide extendibility and skeletal (and high-constrainted) classes to manage all the other cases. Every implementation provides a separate factory to manage the instantiation outside the package and various instance manager systems to minimize the number of real instantiations[6]. The figures B.7,B.9 and B.8 represent the UML class diagrams of parts of the layer that prefer the mechanism of decoration instead of that of inheritance, while, in the figures B.2 and B.3, the class hierarchies presented underline the choice of the static inheritance.

[5] In addition, from a deployment point of view, the aspects could be in turn disabled or enable by a user simply modifing the XML document named *aop.xml* that must be set over the CLASSPATH of the application (see section 3.3.1 for the explanation).

[6] The Multiton pattern is used instead of plain Singleton pattern if the creation of an instance depends on any parameter,

Every wrapper implementation uses the APIs early mentioned to instantiate the right immutable model, so the thread-safeness of the classes is simply accomplished by simple *volatile* variables that refer to the model currently created.

### SelfLetServerMonitor

The last part of the framework, the "server monitor" (*SelfLetServerMonitor*), relies completely on the inferfaces exposed by the APIs and its current implementation is designed upon the *Model-View-Presenter-Controller* pattern (MVPC)[20]. In addition, an active comunication receiver, created upon the the APIs early mentioned, is connected to the main system via an Observer/Observable pattern. These two component are contained in the packages *model*, *views*, *controllers* for the former (figures C.1, C.2) and *comunication* for the latter.
All the components of the system uses a *thread-confinement*[28] approach to solve the thread-safeness requirement imposed by the utilization of the Swing[8] GUI framework choosen for the view part and by the mechanism used to retrieve che monitor messages from the local monitors. In order to improve the flexibility of the behavior of the receiver, its instantiation depends on the strategy choosen for it (as shown in Figure C.3).
The strategies defined in the package of the same name could be decorated to add new behaviors to the receiving component. The architecture of the strategies is shown in Figure C.4.
The views of the system are divided in two logical parts, depending on what kind of informations the end user want to know.
One related to the global state of the SelfLet network (figures 3.9,3.10)

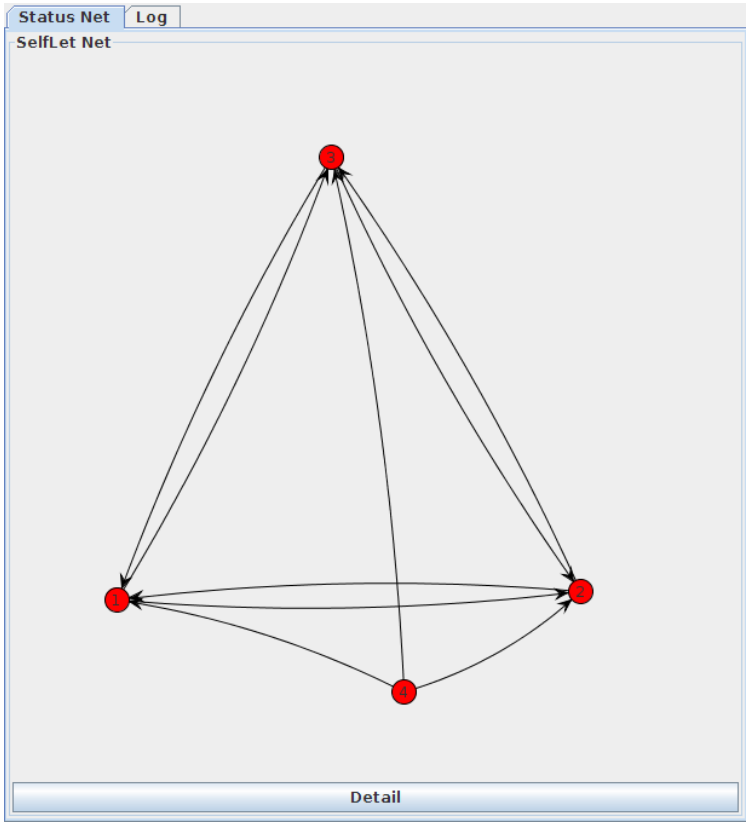


Figure 3.9: Main screen of the SelfLetServerMonitor system.

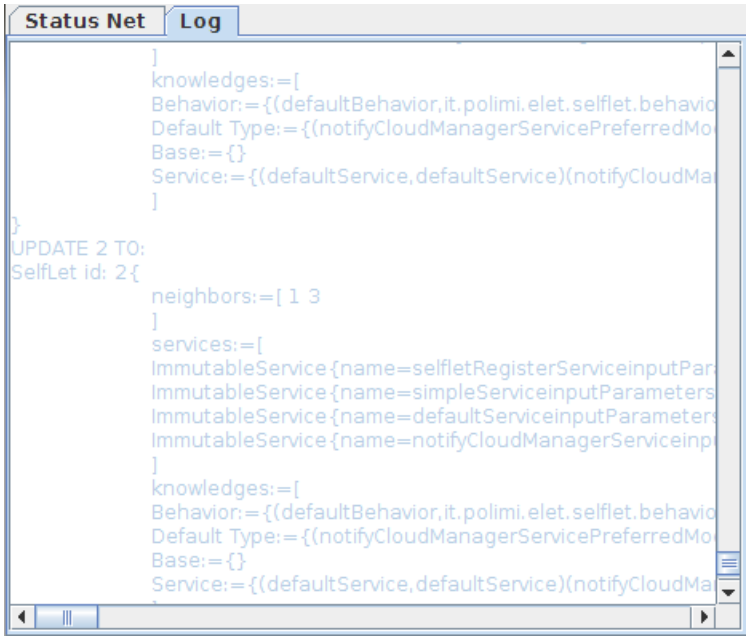


Figure 3.10: Log screen of the SelfLetServerMonitor system.

and a further detail on the state of a single SelfLet (figures 3.11, 3.12, 3.13

and 3.14) belonging to the network.

For the representation of the overall network is been choosen the open-source framework Jung[6] (i,e Java Universal Network/Graph Framework), thanks to its full support to the latest features of the Java language version 1.6, the integration with the Swing framework and all the tools of networks analysis it provides.

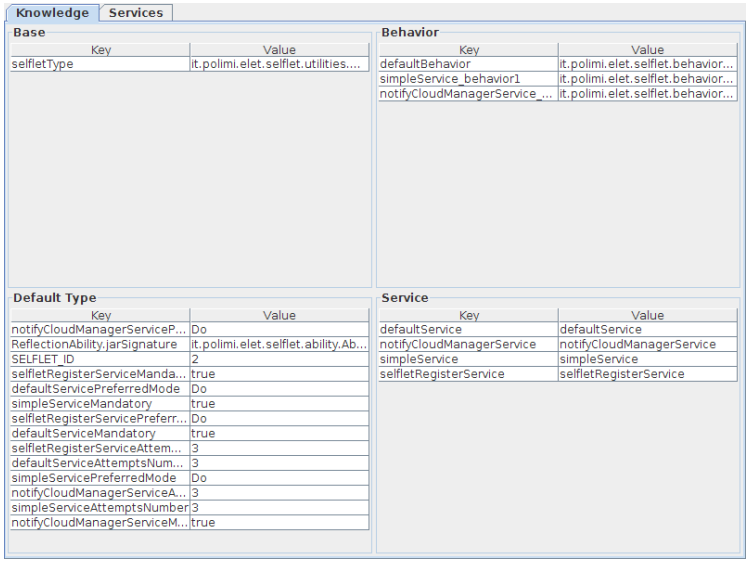


Figure 3.11: Internal Knowledge details of a selected SelfLet.

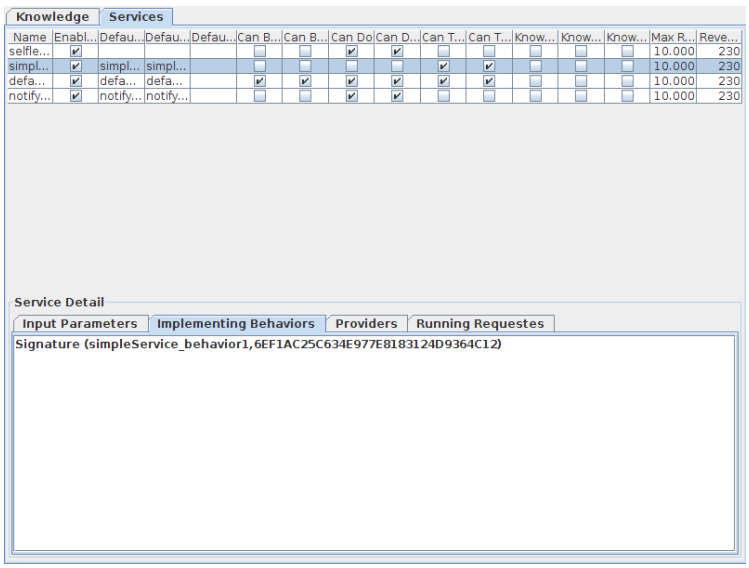


Figure 3.12: Services details of a selected SelfLet.

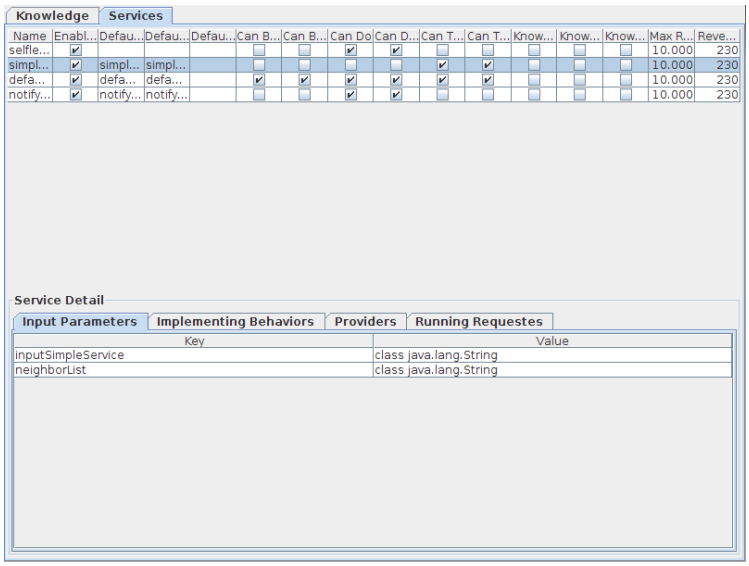


Figure 3.13: Services details of a selected SelfLet.

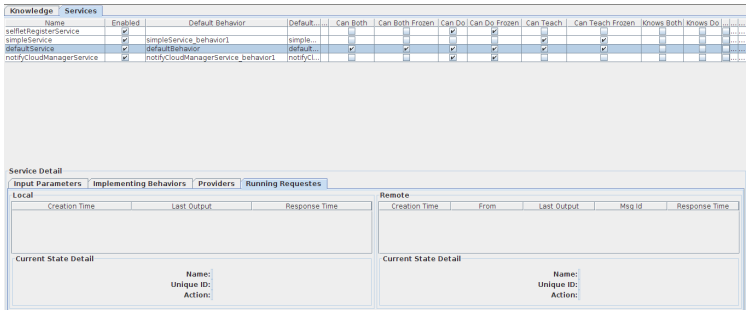


Figure 3.14: Services details of a selected SelfLet.

# Chapter 4

# Case study analysis and results

## 4.1 Objectives

The objective of these case studies is to validate the developed framework in terms of performance impact and in terms of contribution to the tools available for the SelfLet framework.
Since no previous similar works exists on the SelfLet framework, initial efforts have been directed in the understanding of the proper methodology to adopt. Measuring the impact of the monitoring tools requires the development of new case studies involving the exploitation of the characteristics to be tested.
In order to do that it has been necessary to create case studies that would cause state changes in the monitored concerns of the SelfLet Framework and a successive reaction of the (local part of) monitoring system.
As it has been explained in the previous chapter, these state changes would lead the local monitors to create and send proper messages to a REDS broker: these two reactions of the monitoring system are the objectives of the evaluation.
In order to validate the proposed approach, it is also necessary to extract relevant numerical data concerning the impact of the feature being developed. As a further requirement there is the representativeness of the developed case studies; indeed, since it is not possible to cover all the possible space of experiments, it is necessary to find what are the most relevant ones to which other case studies can be referred to. In this way an approximate estimate of the impact of this new feature can be evaluated in other case studies.

## 4.2 Methodology

In order to validate the implemented feature, the experiments have been organized in two different case studies. The rationale behind the two case studies is to create the conditions for an intervention of the monitoring framework.
More precisely, all the experiments involve a comunication between two selfLet, analyzing the different performances in term of messages sent or services requests accomplished. Each case study has been tested for different time intervals on the same "virtual" machines[1]. This also gave the opportunity to test with a real system, with real lags caused by an unpredictable network latencies dependent by the distance of the cloud machines used.
To run an experiments it's necessary to start (through SSH[7]) on the Cloud instances the REDS middleware with the monitor APIs (*SelfLetMonitorAPI.jar*) on its CLASSPATH first so that the SelfLets can connect to it using a common IP address (obtained by the public DNS of the machine hosting the REDS broker). The second step depends on which kinds of samples are needed to be taken. The three test cases involve three possible monitor configurations (i.e. three different families of experiments) of the two selfLets (named, for brevity, *S1* and *S2*). In subsections 4.2.1 and 4.2.2 these configurations are explained in more detail.
All the experiments share the same SelfLets behaviors configurations:

- S1 (Figure 4.1) is capable to accomplish locally the request of the service *simpleService* in *do* mode and without any optimization policy
- S2 (Figure 4.2) requests periodically (every 2000 milliseconds) the execution of a remote service called *simpleService* in *do* mode and without any optimization policy

The action contained in the state of the elementary behavior of S1 is actually a very simple one; indeed, it just produces a predefined string as output. The result of executing the service is thus obtaining this string; the actual value is irrelevant to the example.

[1]The machines are offered by the EC2 Cloud service of Amazon[1]

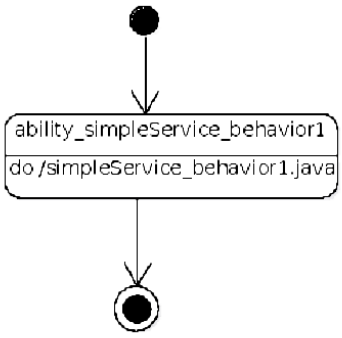


Figure 4.1: Main behavior of S1

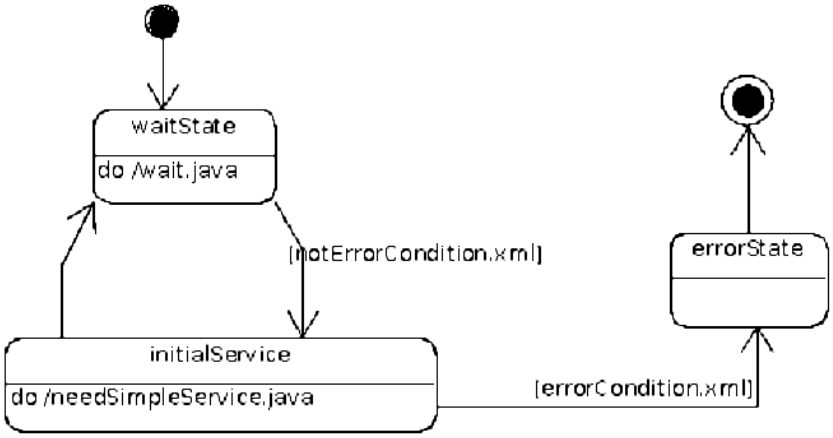


Figure 4.2: Main behavior of S2

The next subsection focuses on the kind of data collected and the used tools, parts of the monitor framework itself.

### 4.2.1 First case study: evaluating speed of execution

In this case study there are two kind of experiments with the same selfLets, but different monitoring configurations:

1. S1 with all the concerns of the local monitor enabled, while S2 with no monitoring at all

2. S1 and S2 either without local monitoring

Several sessions of each kind of experiments are started indipendently (i.e. on different Cloud instances) and at the end a Ruby[10] script which analyzes all the logs created by S1 in the two kind of experiments is launched. The script creates two different files (one for each configuration tested) readable by Gnuplot[4] which represent a table having, for each row, two different values: the *timestamp* in milliseconds in which a single request of simpleService execution is accomplished and the related *total number of requestes accomplished*. The comparison between the two files may help to better understand the computational overhead introduced by the local monitor to the original engine.

### 4.2.2 Second case study: evaluating messages overhead

In this case study the system is launched several times with only one monitor configuration: S1 with all the concerns of the local monitor enabled plus the messages tracer and S2 with no monitoring at all. An aspect collects informations about the activities of every REDS's *DispatchingService* instantiated and at the end of the tests is possible to read the two files created by the aspect readable by Gnuplot that show the number of messages sent by the local monitor compared to that sent (or replied) by the SelfLet itself.
The analysis of this comparison may help to know the overhead in term of network bandwidth allocation introduced by the monitoring system in a condition in which the internal state of the selfLet is updated frequently plus verified the entity of the initial (and predictable) big flow of monitoring informations sent during the configuration and initialization of the selfLet.

## 4.3 Results

The two test cases described above were tried a meaningful number of time in order to verify the validity of the results shown and every time all the results converge to the results shown below. In particular the first case study reports the graphs represented in figure 4.3.

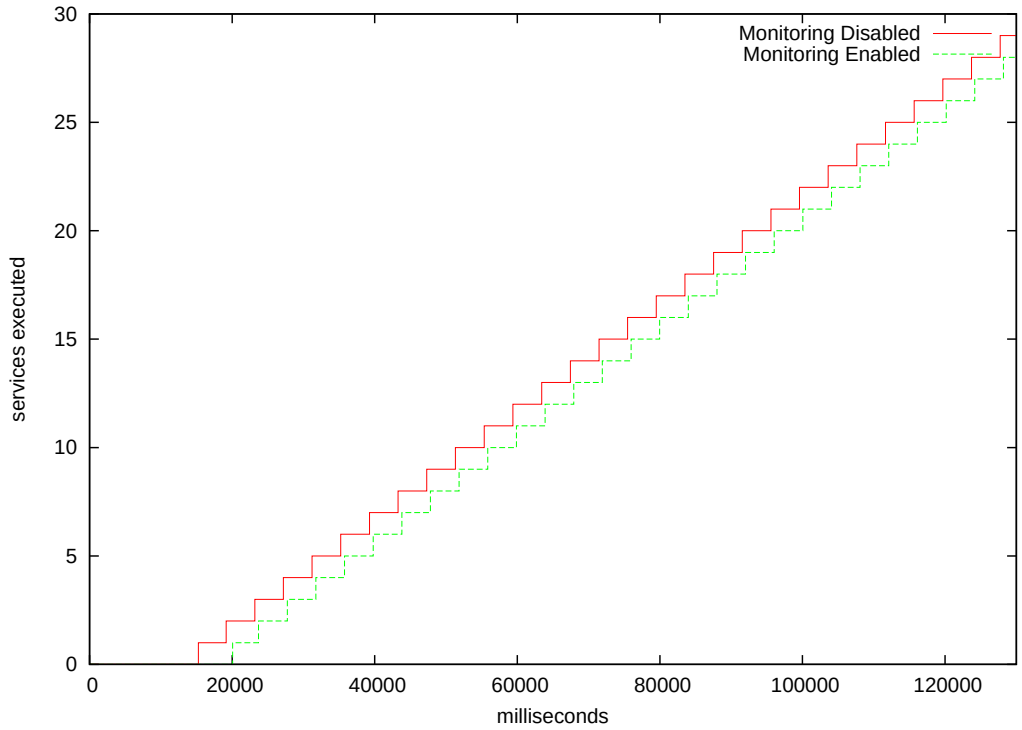


Figure 4.3: Result of the first case study tests

From this test is possible to evaluate the real impact on the performance in term of services accomplished rate of the system, and the figure 4.3 shows that the "real" overhead introduced is only at the starting part of the life-cicle of the SelfLet, right after the initialization of the SelfLet. In this part the monitoring system is sending all the informations to the remote Monitor. Anyway, the overall impact is lesser than 5% and the future trend of the gap seems to be the same. To improve the performance of the local monitor the available solutions appear to be:

1. to optimize the aspects or the wrappers in order to collect information in better ways i.e. improving the speed of creation of the model that will be sent and/or choosing better pointcuts in which the monitor aspects use the wrapper to obtain the parameters for the translation of the SelfLet state in the immutable models to be sent.

2. to eliminate the sending of the monitor messages during that initialization phase, but deleting that comunications from the knowledge of the remote Monitor, which will receive the actual state of the SelfLets after the real start of the engines.

However, an evaluation of these proposals will be done at the end of the section, because the results of the second test study could help to choose one or the other approach.
The results of second test case show the trend of the message sending rate of the local monitoring system compared with that of the SelfLet engine. The trends are related to the same configuration and are presented in Figure 4.4.

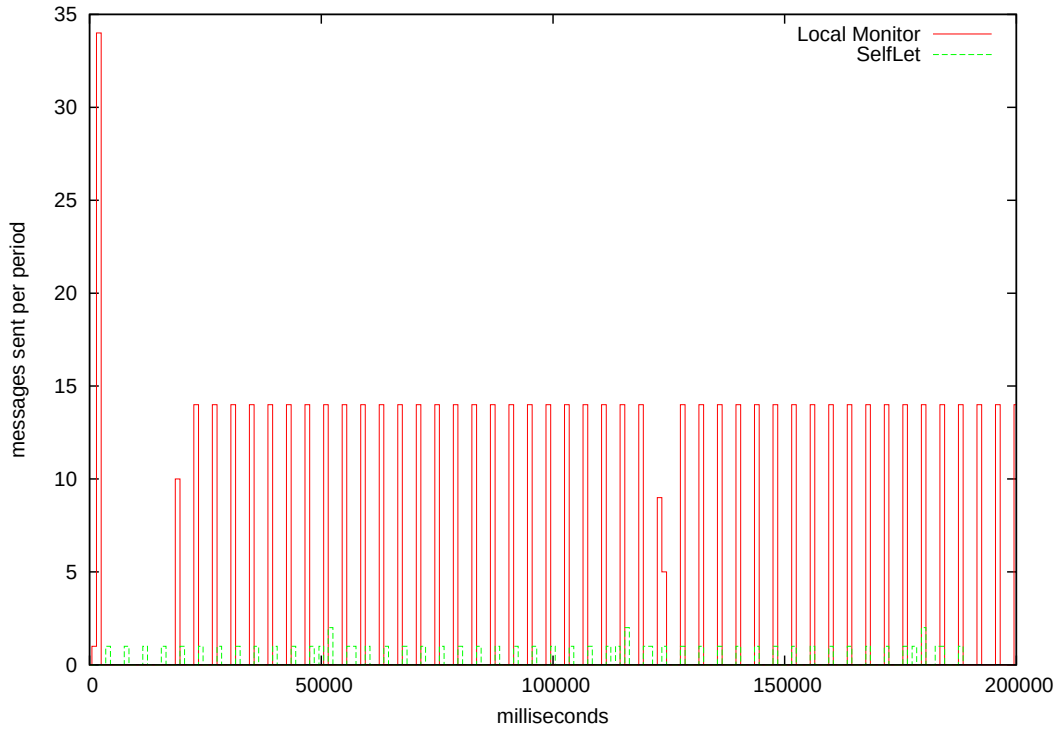


Figure 4.4: Result of the second case study tests

From the figure, is possible that the reason of the bad performance of the monitored system in the initialization phase could be explained by the presence of a big quantity of variations in the state of the monitored system during that phase. The comparison evidentiates that the messages sent by the monitor after that initial part are about seven time the messages sent by the selflet itself in the same interval and in the same interval the service performance of the SelfLet suffers a delay, becoming less responsive. The value of the delay seems influenced by the instantiations of the new remote running services requested by the other SelfLet and the Figure 4.3 confirms this idea. These evaluations suggest the thesis that might exist a correlation between the number of messages sent and the performance of the SelfLet. Assuming this idea as correct, one possible way to affect the weight of this dependency

in the overall performance of the system is to cut off every comunication with the Monitor server during the initialization phase. This approach changes the policy of the local monitor and affects the nature of monitoring too, so the other solution is preferable to improve the performance of the local monitor without affecting the quality of the informations delivered.
Only for the sake of completeness, the Figure 4.5 shows the same comparison of the figure 4.4, but in the case of "lazy"comunication with the server. The current local monitor implementation supports this kind of configuration too but, by default, is adopted the one that comunicates every data changes.

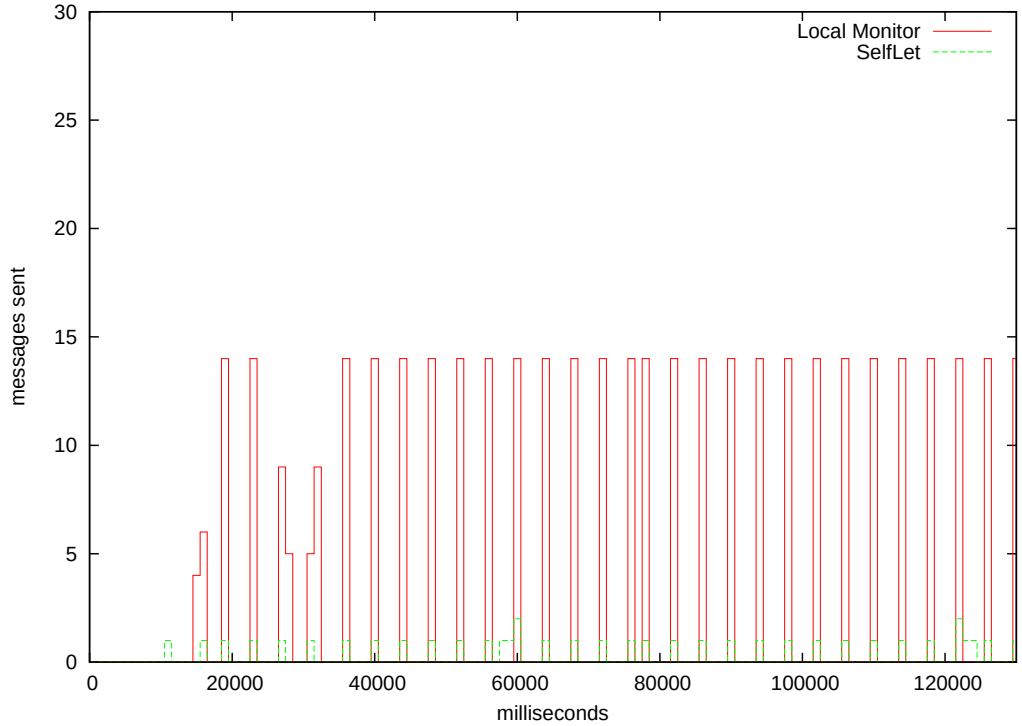


Figure 4.5: Result of the second case study tests adopting the "lazy" comunication approach

The Figure 4.6 shows, instead, the related gain in term of service performance, adopting the "lazy" approach, compared with the performance without every kind of monitoring.

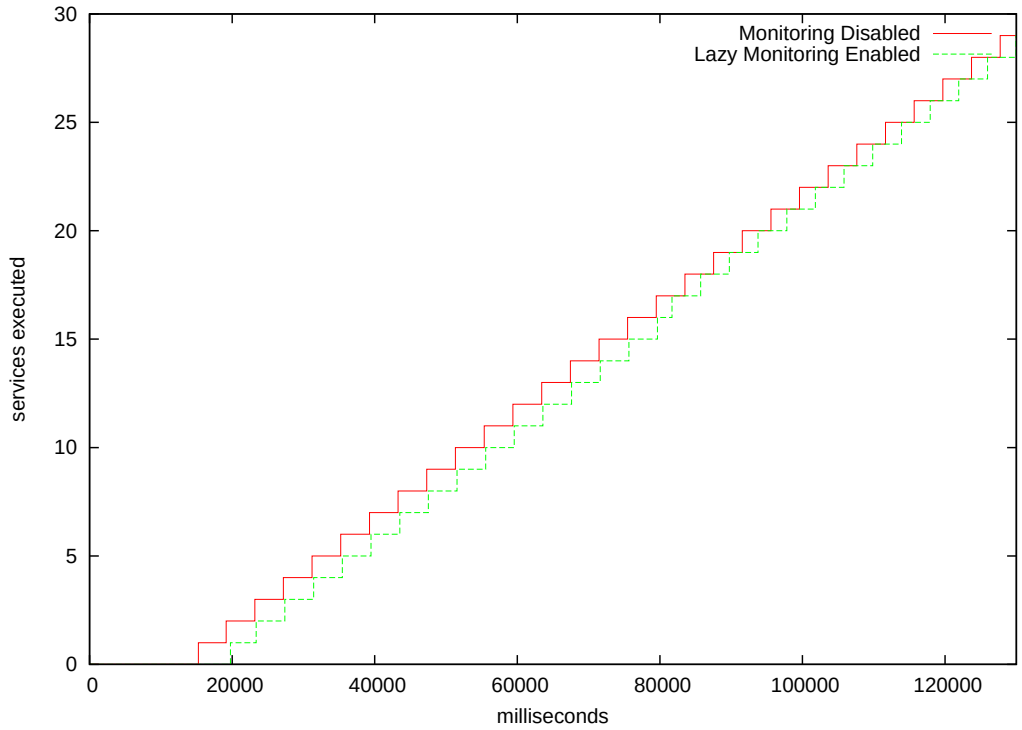


Figure 4.6: Result of the first case study tests adopting the "lazy" comunication approach

The final verdict about the solution of cutting off the flow of messagges is that the system obtains only a negligible performance boost compared to the "full comunication" default solution, so the choice of what kind of monitoring to adopt should be guided only by constraints of the network bandwidth.
In conclusion, the default version of the local monitor has shown a negligible[2] computational impact over the monitored system and a predictable comunication overhead, tunable only at the price of choosing a different monitoring policy.

[2]Not influenced by the choice to adopt AspectJ and load-time weaving[9].

# Chapter 5

# Conclusions and Future Works

## 5.1 Conclusion

The presented work started by setting three main goals:

1. to realize a development framework for the on-line distributed monitoring of the SelfLets
2. to find a solution at the problem of making it simple to maintain and extend without affecting the SelfLet framework
3. to realize a case study to demonstrate the effectiveness of the solution adopted

It is now pointed out how these objectives have been achieved throughout this work.
The first and the third objective are covered respectively in the chapters 3 and 4, while the second one is better explained below.
The main difference between the approach adopted in the realization of the framework from other solutions is about the choice of the Java language for the specification of all the parts of the framework. Other approaches relies on declared flexible, solid architecture[26] and great support for the developer in almost every functionality required by such a kind of tools, but when these frameworks need to be extended in order to express new type of "points of interest" in the monitored system, the vision changes. The choice of the *@AspectJ* syntax as the point of contact with the monitored system makes the difference from these solutions: it's flexible, robust, powerful (as shown in chapter 4) and

expressable in simple Java (with annotations). All the other qualities of the monitoring system realized in this thesis are not more than good cuncurrent programming practices, oriented to the production of an high-performance, thread-safe and defensive API. So, the second goal is fully achieved: no one users of the presented API will need to modify a *java agent*[5] in *C++* with all the risks of this kind of operations[18].

## 5.2 Future work

The future work on the presented framework is mostly signed and can be summarized as follows:

- add the support for the run-time configuration of the local monitors by the remote monitor
- add the support for the run-time modification of the systems monitored in the points expressed by the remote monitor
- find better models/istantiation technique of the models, to improve the performance of the local monitor during the collection of the informations

Some of these goals could be currently achieved relying on the features of the presented implementation of the framework:

- the double-way comunication possible with Reds[16] combined with the State pattern[19] applied to the wrappers, in order to change dinamically the behavior of the local monitor
- the use of the *@Around* construct of *@AspectJ*[22] in order to permit modifications to the flow of execution of the monitored system

In addition, other features of this framework could be transferred to the SelfLet framework itself: the autonomic qualities of the framework would be better expressed if the autonomic layer (which provides a form of on-line monitoring) were to be applicable to every kind of Java application, even if legacy and not only those expressed following the SelfLet paradigm.

A similar solution, which adopts tha same technology of the monitoring framework presented in this thesis, was presented by IBM[15], but with the distribuited qualities of the SelfLet solution, it will became one of the best expressions of the Autonomic vision[21].

# Appendices

# Appendix A

# SelfLetMonitorAPI UML 2.0 Class Diagrams

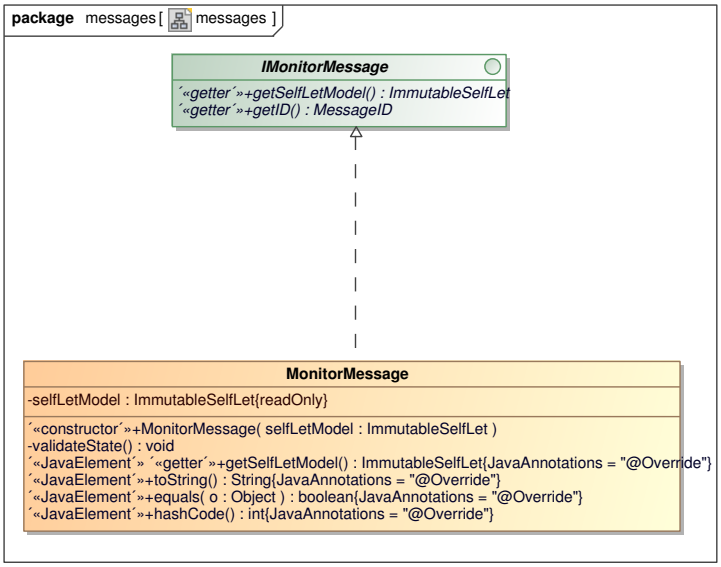


Figure A.1: UML Class diagram of part of the system

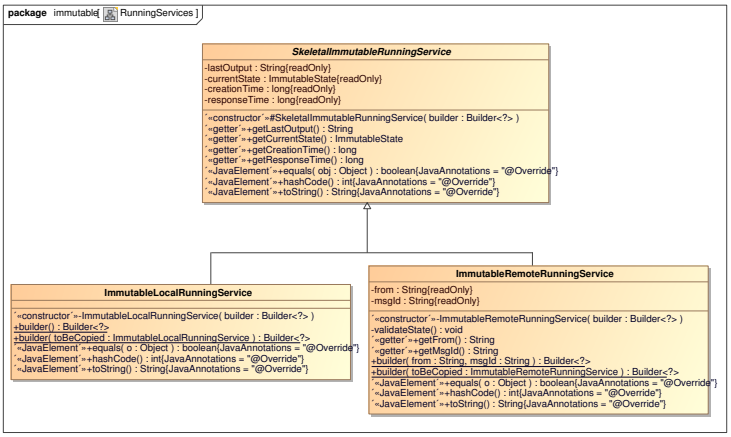


Figure A.2: UML Class diagram of part of the system

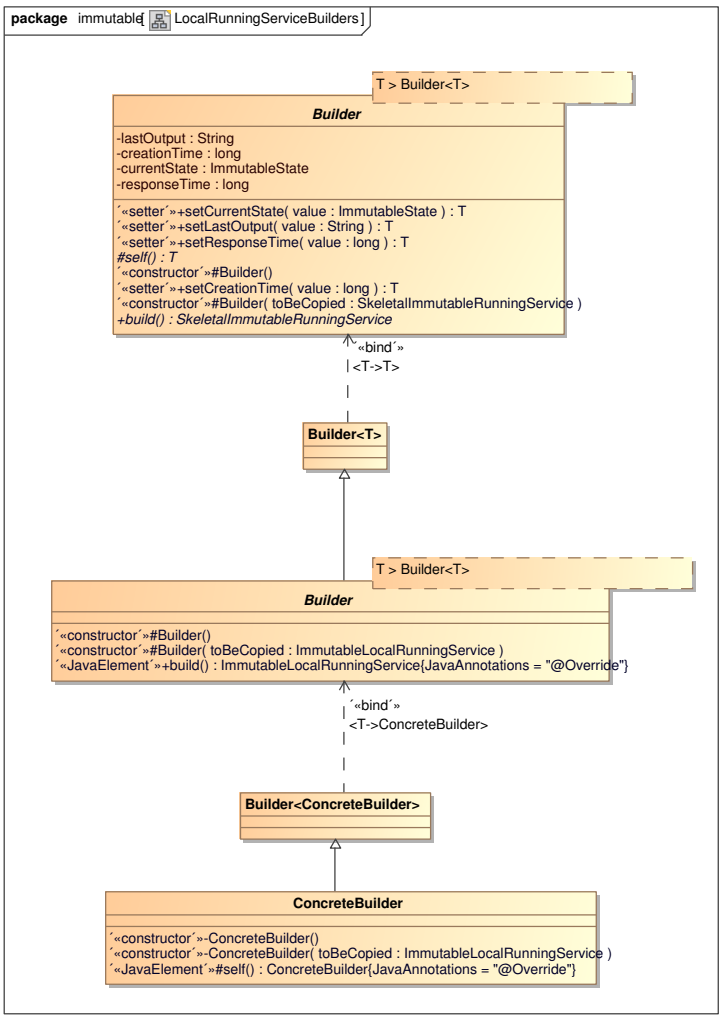


Figure A.3: UML Class diagram of part of the system

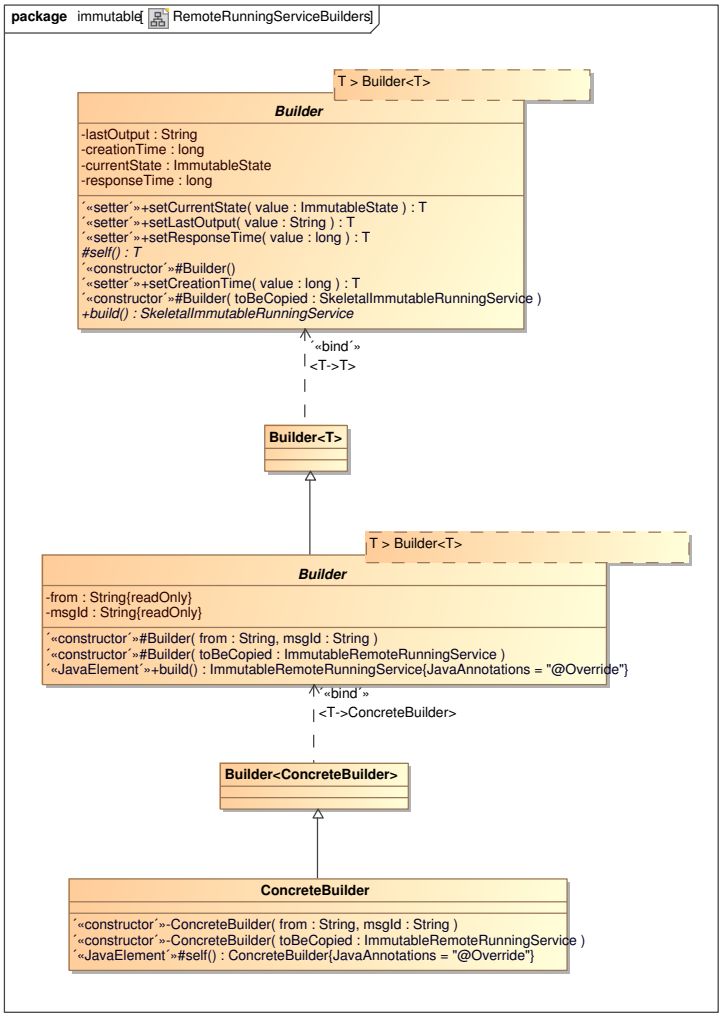


Figure A.4: UML Class diagram of part of the system

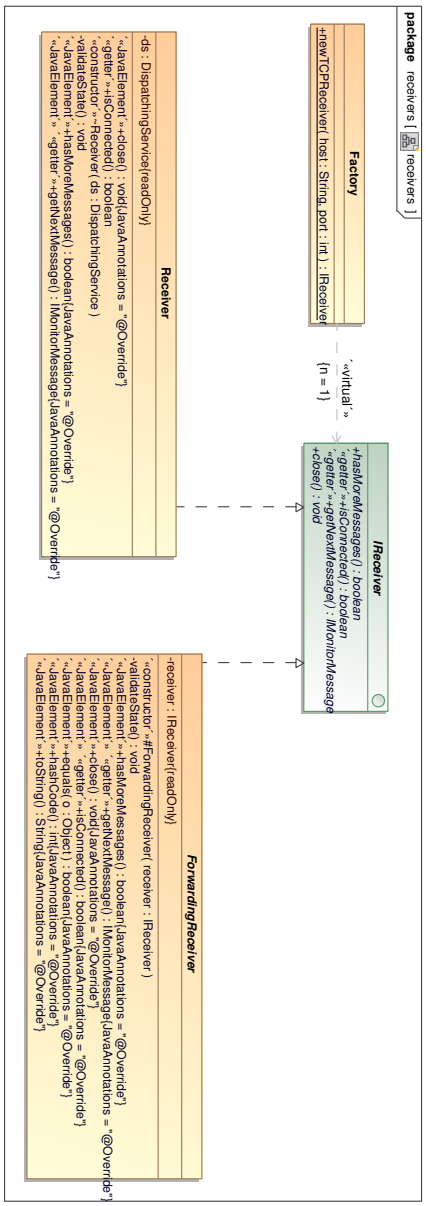


Figure A.5: UML Class diagram of part of the system

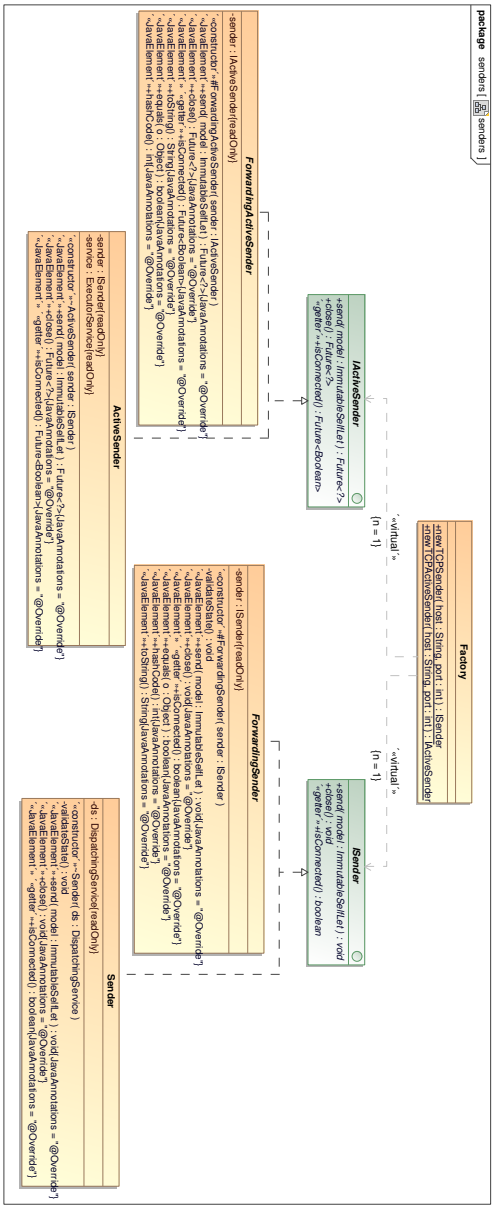


Figure A.6: UML Class diagram of part of the system

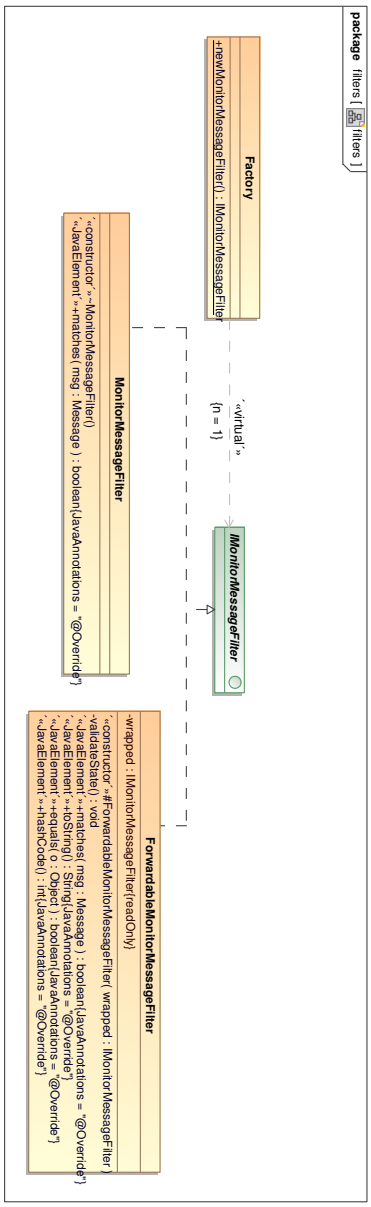


Figure A.7: UML Class diagram of part of the system

# Appendix B

# SelfLetClientMonitor UML 2.0 Class Diagrams

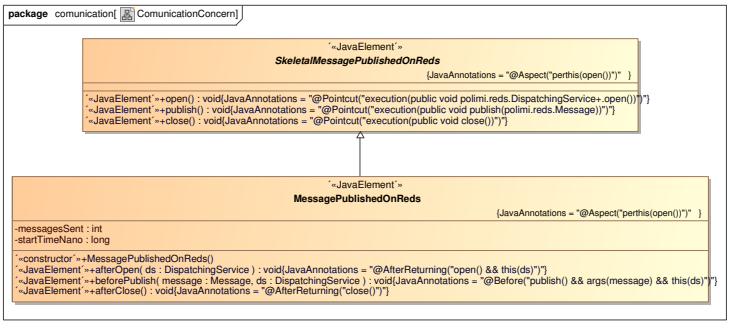


Figure B.1: UML Class diagram of part of the system

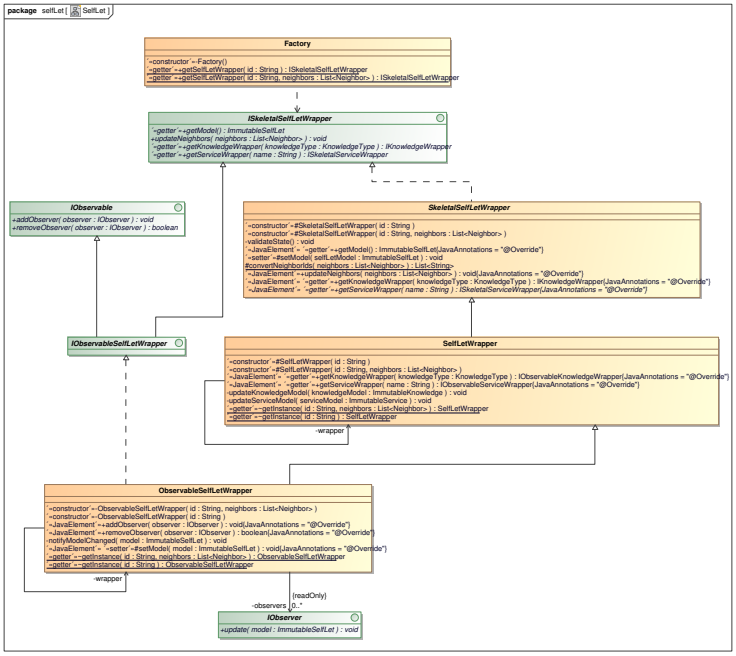


Figure B.2: UML Class diagram of part of the system

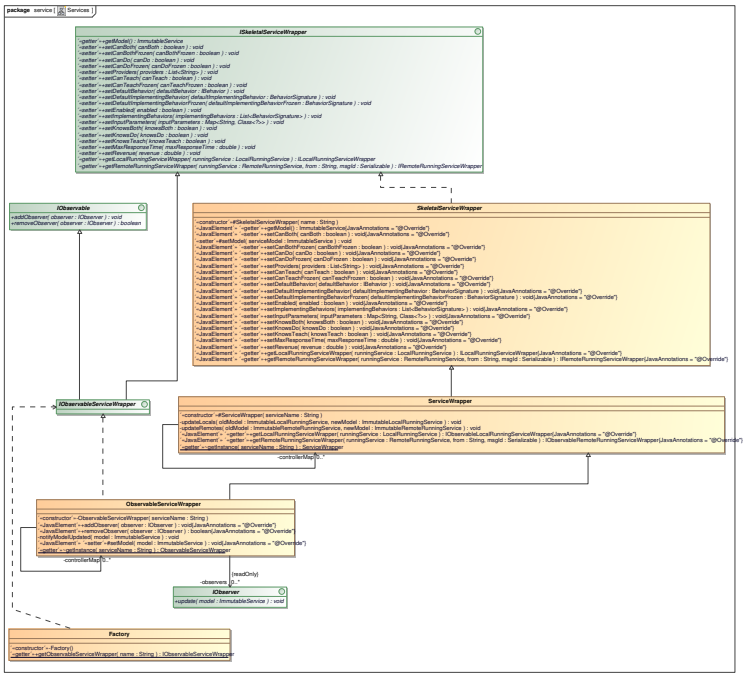


Figure B.3: UML Class diagram of part of the system

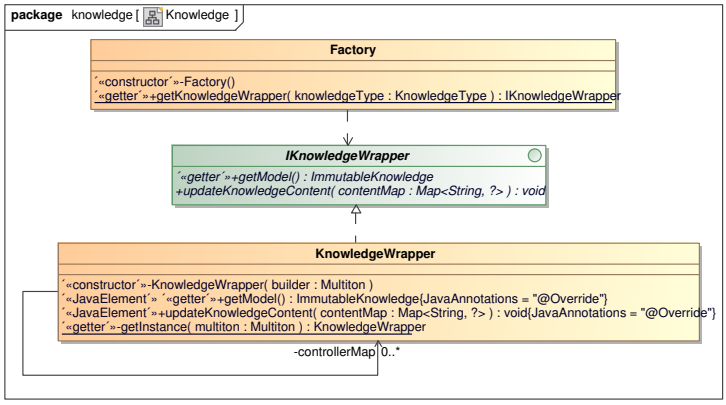


Figure B.4: UML Class diagram of part of the system

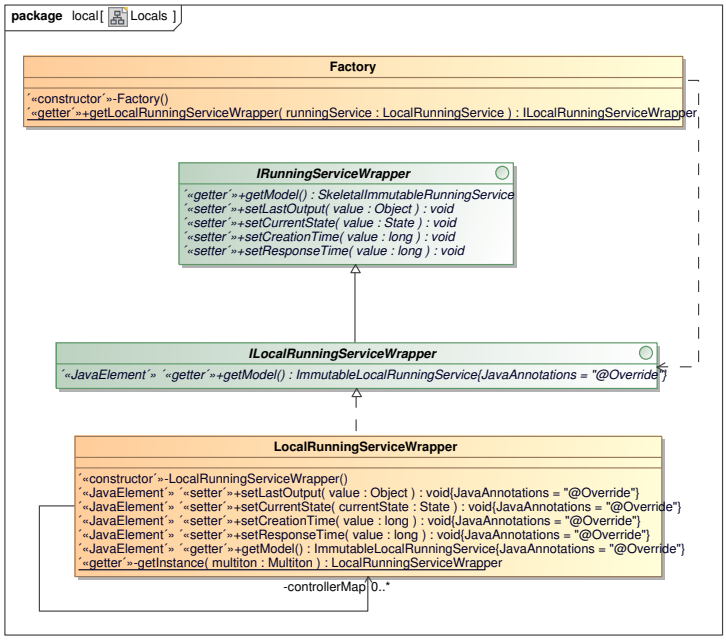


Figure B.5: UML Class diagram of part of the system

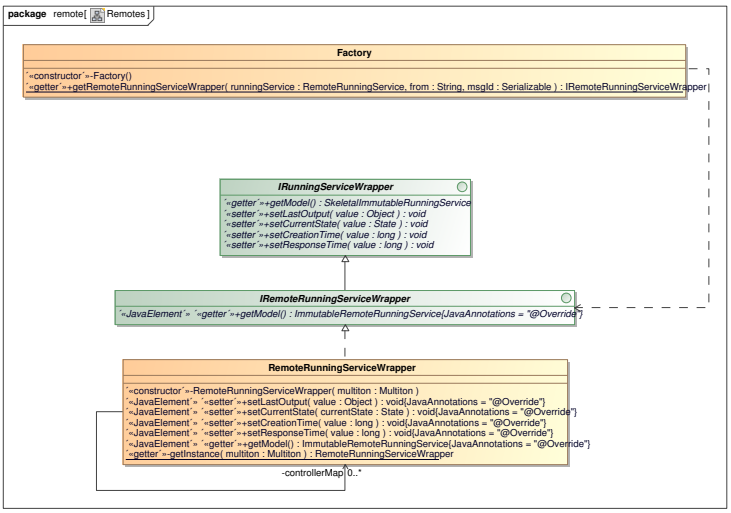


Figure B.6: UML Class diagram of part of the system

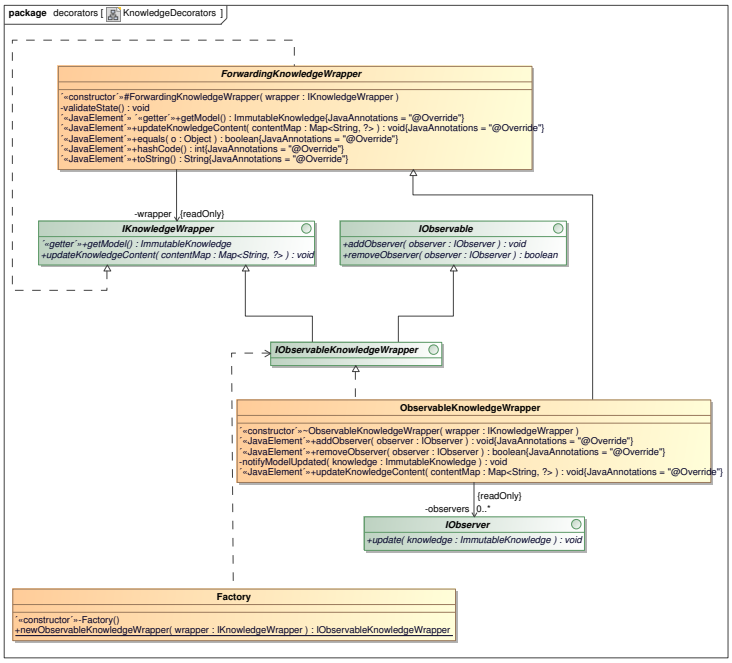


Figure B.7: UML Class diagram of part of the system

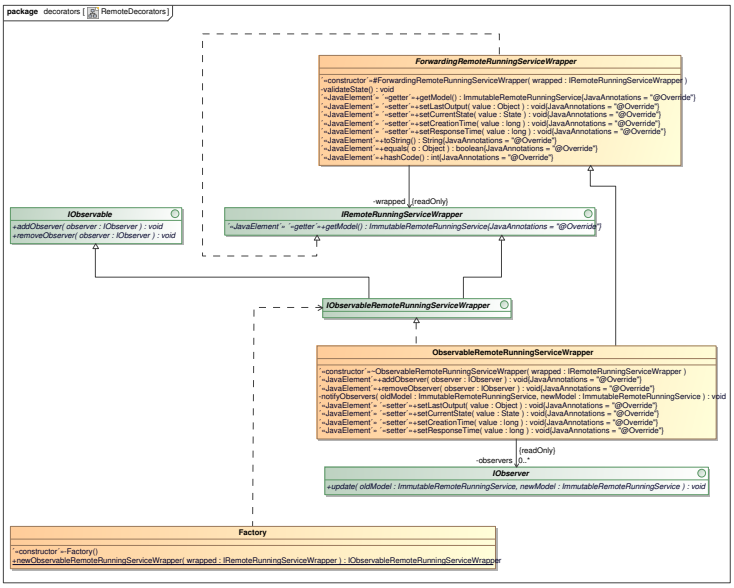


Figure B.8: UML Class diagram of part of the system

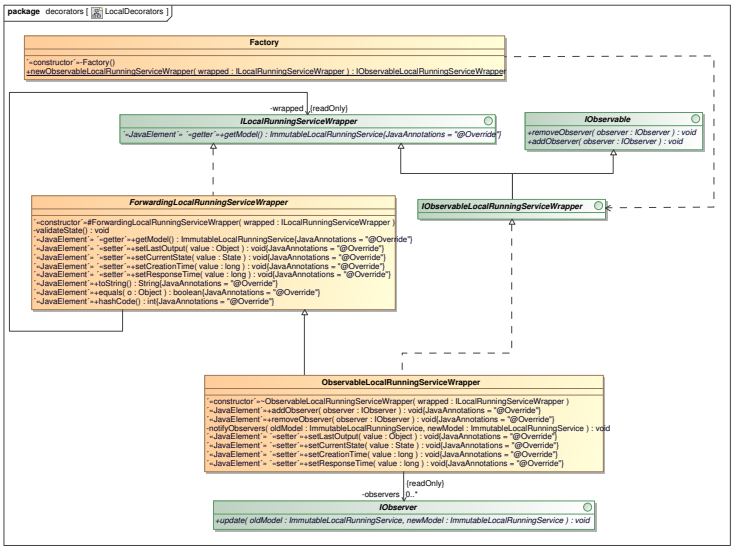


Figure B.9: UML Class diagram of part of the system

# Appendix C

# SelfLetServerMonitor UML 2.0 Class Diagrams

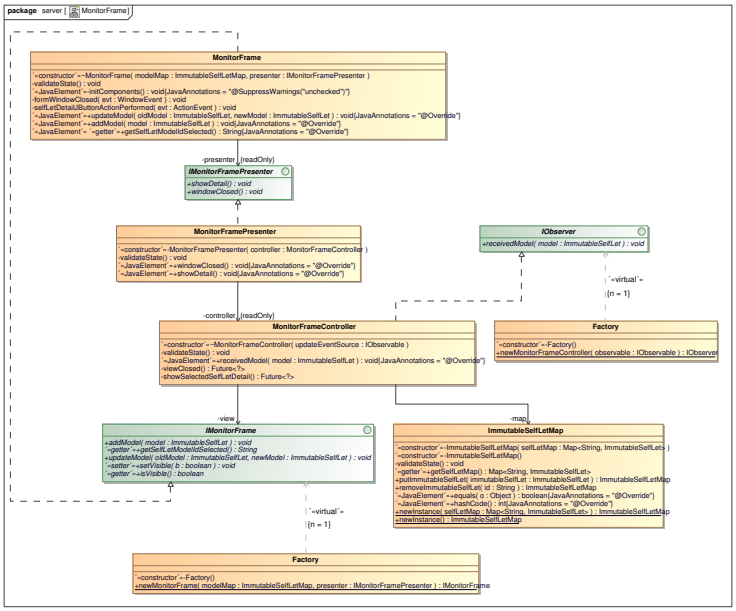


Figure C.1: UML Class diagram of part of the system

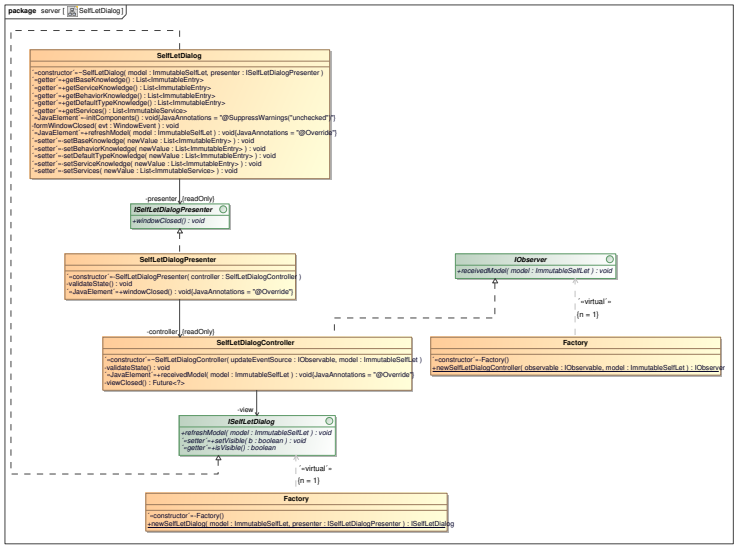


Figure C.2: UML Class diagram of part of the system

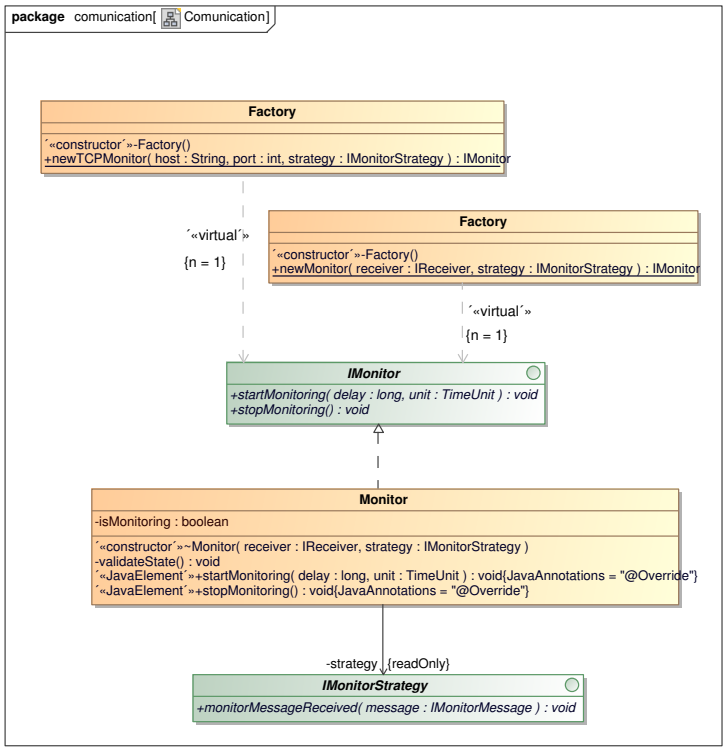


Figure C.3: UML Class diagram of part of the system

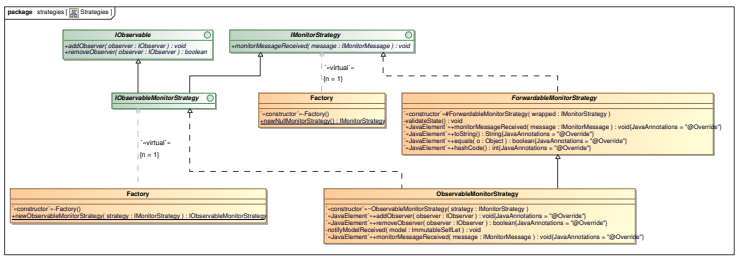


Figure C.4: UML Class diagram of part of the system

# Bibliography


[1] Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

[2] Argouml modeling tool. http://argouml.tigris.org/.

[3] Drools. http://www.jboss.org/drools.

[4] Gnuplot, a portable command-line driven graphing utility. http://www.gnuplot.info/.

[5] Java platform debugger architecture. http://download.oracle.com/javase/6/docs/technotes/guides/jpda/architecture.html.

[6] Jung, java universal network/graph framework. http://jung.sourceforge.net/index.html.

[7] Openssh. http://www.openssh.com/.

[8] Package javax.swing. http://download.oracle.com/javase/6/docs/api/javax/swing/package-summary.html.

[9] Performance overhead of aspectj. http://www.eclipse.org/aspectj/doc/released/faq.php#q:effectonperformance.

[10] Ruby homepage. http://www.ruby-lang.org/en/.

[11] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[12] Marian Bubak, Wlodzimierz Funika, Marcin Smetek, Zbigniew Kilianski, and Roland Wismüller. Event handling in the j-ocm monitoring system. In Roman Wyrzykowski, Jack Dongarra, Marcin Paprzycki, and Jerzy

Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 3019 of *Lecture Notes in Computer Science*, pages 344–351. Springer Berlin / Heidelberg, 2004.

[13] Marian Bubak, Wlodzimierz Funika, Marcin Smetek, Zbigniew Kilianski, and Roland Wismüller. Request processing in the java-oriented omis compliant monitoring system. In Roman Wyrzykowski, Jack Dongarra, Marcin Paprzycki, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 3019 of *Lecture Notes in Computer Science*, pages 352–359. Springer Berlin / Heidelberg, 2004.

[14] Nicolò M. Calcavecchia, Danilo Ardagna, and Elisabetta Nitto. The emergence of load balancing in distributed systems: the selflet approach. In Danilo Ardagna and Li Zhang, editors, *Run-time Models for Self-managing Systems and Applications*, Autonomic Systems, pages 97–124. Springer Basel, 2010.

[15] Hoi Chan and Trieu C. Chieu. An approach to monitor application states for self-managing (autonomic) systems. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 312–313, New York, NY, USA, 2003. ACM.

[16] Gianpaolo Cugola and Gian Pietro Picco. Reds: a reconfigurable dispatching system. In *Proceedings of the 6th international workshop on Software engineering and middleware*, SEM '06, pages 9–16, New York, NY, USA, 2006. ACM.

[17] Davide Devescovi, Elisabetta Di Nitto, and Raffaela Mirandola. An infrastructure for autonomic system development: the selflet approach. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 449–452, New York, NY, USA, 2007. ACM.

[18] Wlodzimierz Funika and Pawel Swierszcz. Dynamic instrumentation of distributed java applications using bytecode modifications. In *International Conference on Computational Science (2)*, pages 534–541, 2006.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[20] Martin Hunter. Tidying the house: The mvpc software design pattern. `http://www.martinhunter.co.nz/articles/MVPC.pdf`, 2006.

[21] IBM. The autonomic computing manifesto. `http://www.research.ibm.com/autonomic/manifesto`.

[22] Ramnivas Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications Co., Greenwich, CT, USA, 2nd edition, 2009.

[23] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 214–223, New York, NY, USA, 1986. ACM.

[24] Thomas Ludwig and Roland Wismüller. Omis 2.0 - a universal interface for monitoring systems. In *Proceedings of 4th European PVM/MPI Users' Group Meeting*, pages 267–276. Springer Verlag, 1997.

[25] Thomas Ludwig, Roland Wismüller, and Michael Oberhuber. Ocm - an omis compliant monitoring system. In *Parallel Virtual Machine – EuroPVM'96*, pages 81–90. Springer Verlag. http://wwwbode.informatik.tu-muenchen.de/ wismuell/pub/europvm96b.ps.gz, 1996.

[26] Wl odzimierz Funika, Marian Bubak, and Marcin Smetek. Monitoring system for distributed java applications. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra, editors, *Computational Science - ICCS 2004*, volume 3038 of *Lecture Notes in Computer Science*, pages 472–479. Springer Berlin / Heidelberg, 2004.

[27] Castle Park, Cambridge Cb Rd, Yigal Hoffner, and Yigal Hoffner. Monitoring in distributed systems monitoring in distributed systems, 1994.

[28] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[29] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 38–45, New York, NY, USA, 1986. ACM.

[30] The OSGi Alliance. OSGi service platform core specification, release 4.1. `http://www.osgi.org/Specifications`, 2007.