Degree project

Efficient Monitoring of OSGi Applications

Author: Portero Aníbal
Supervisor: Abbas Nadeem
Date: 2013-08-27
Course Code: 2DV00E, 15 credits
Level: Bachelor
Department of Computer Science
Abstract

As software evolves and becomes more complex, self-adaptive systems become a more interesting solution. Self-adaptive software systems are capable to perform changes in themselves without human intervention. To make this possible it is necessary to perform a good observation of the system and its environment. This observation is made by a monitoring system.

In this paper, a framework for monitoring OSGi based applications is presented. OSGi is a module system and service platform for Java. This framework offers run-time information about OSGi modules, services and their behavior.

The first step is to make a state-of-the-art survey of existing methods to monitor in the field of self-adaptive systems and OSGi based applications. The survey reviews a set of articles in the area. It is performed to discover what are the common objectives and problems that any monitoring system faces. After that, the requirements for the framework are stated. These requirements specify the functionality that the framework is required to provide, along with the quality attributes that it has to meet. To demonstrate use of the contributed monitoring framework, we have developed two example demonstrators. The objective of these demonstrators is to provide users of the framework with working examples, so that they can use the framework to develop their own monitoring systems.

Keywords: OSGi, monitoring system, self-adaptive software system
Table of content

1 Introduction ........................................................................................................................................ 1
   1.1 Background ................................................................................................................................... 1
       1.1.1 Self-Adaptive Software Systems ......................................................................................... 1
       1.1.2 Monitoring systems in Self-Adaptive Software Systems ..................................................... 1
       1.1.3 OSGi ....................................................................................................................................... 1
   1.2 The problem .................................................................................................................................... 1
   1.3 The process .................................................................................................................................... 2
   1.4 Restrictions ..................................................................................................................................... 2
   1.5 The structure of the report ............................................................................................................ 2
2 Monitoring Systems: State-of-the-Art Survey ...................................................................................... 3
   2.1 The process of the survey .............................................................................................................. 3
   2.2 Self-Adaptive Software Systems ................................................................................................... 3
   2.3 Monitoring in Self-Adaptive Systems ............................................................................................. 4
       2.3.1 [IMSERTM] ........................................................................................................................... 4
       2.3.2 [RBMSASO] ........................................................................................................................ 5
       2.3.3 [FRTMSO] .......................................................................................................................... 5
   2.4 Monitoring Systems in OSGi ......................................................................................................... 5
       2.4.1 [A-OSGi] .............................................................................................................................. 6
       2.4.2 [LLMOSGi] .......................................................................................................................... 7
       2.4.3 [RMVOSGi] .......................................................................................................................... 7
       2.4.4 [MMROSGi] .......................................................................................................................... 8
       2.4.5 [CMOSGi] ............................................................................................................................. 8
       2.4.6 [KOSGi] ................................................................................................................................ 8
       2.4.7 [OSGiCJRM] ......................................................................................................................... 9
       2.4.8 [TIAOSGi] ............................................................................................................................ 9
       2.4.9 [R-OSGi] ................................................................................................................................ 9
       2.4.10 [OSGiD] ............................................................................................................................. 9
   2.5 Results of the survey ....................................................................................................................... 10
       2.5.1 What to monitor ...................................................................................................................... 10
       2.5.2 Why to monitor ....................................................................................................................... 11
       2.5.3 When to monitor ..................................................................................................................... 11
       2.5.4 Where to monitor .................................................................................................................... 12
       2.5.5 How to monitor ....................................................................................................................... 13
   2.6 Conclusions from the survey .......................................................................................................... 14
       2.6.1 Monitoring various components ............................................................................................ 14
       2.6.2 Get-based and Event-based monitoring .............................................................................. 14
       2.6.3 Decoupling ............................................................................................................................ 14
       2.6.4 API ......................................................................................................................................... 15
       2.6.5 Efficiency .............................................................................................................................. 15
       2.6.6 Extensibility ........................................................................................................................... 15
3 Requirements Specification .................................................................................................................. 16
   3.1 Overall description .......................................................................................................................... 16
   3.2 Functional requirements ................................................................................................................ 16
   3.3 Interface requirements .................................................................................................................... 16
   3.4 Quality attributes ........................................................................................................................... 17
   3.5 General requirements ...................................................................................................................... 17
   3.6 Design constraints .......................................................................................................................... 17
4 Implementation ..................................................................................................................................... 18
   4.1 Package organization ...................................................................................................................... 18
4.2 event package .............................................................................................................. 19
   4.2.1 Repository monitoring ......................................................................................... 19
   4.2.2 State monitoring .................................................................................................. 20
   4.2.3 Manifest monitoring ............................................................................................ 21
   4.2.4 Data field monitoring .......................................................................................... 22
   4.2.5 Service monitoring .............................................................................................. 23
4.3 event.filter package .................................................................................................. 24
   4.3.1 Bundle filtering .................................................................................................... 24
   4.3.2 Type filtering ....................................................................................................... 25
5 Results .............................................................................................................................. 26
   5.1 Demonstrator 1: bundles ......................................................................................... 26
       5.1.1 Repository monitoring ...................................................................................... 27
       5.1.2 State monitoring ............................................................................................... 28
       5.1.3 Manifest monitoring ......................................................................................... 29
       5.1.4 Data field monitoring ...................................................................................... 30
       5.1.5 Id filtering ......................................................................................................... 31
       5.1.6 Type filtering .................................................................................................... 32
   5.2 Demonstrator 2: services ......................................................................................... 34
       5.2.2 Service monitoring ............................................................................................ 36
   5.3 Traceability matrix .................................................................................................. 37
6 Conclusions ....................................................................................................................... 39
   6.1 Summary .................................................................................................................. 39
   6.2 Further work ............................................................................................................. 39
References ............................................................................................................................ 40
List of figures

Figure 2.1 – Self-adaptive software system structure
Figure 4.1 – Package organization of the monitoring framework
Figure 4.2 – EventMonitor class
Figure 4.3 – RepositoryMonitor class
Figure 4.4 – RepositoryEvent class
Figure 4.5 – StateMonitor class
Figure 4.6 – BundleEvent class
Figure 4.7 – ManifestMonitor class
Figure 4.8 – ManifestEvent class
Figure 4.9 – DataFieldMonitor class
Figure 4.10 – DataFieldEvent class
Figure 4.11 – ServiceRegistryMonitor class
Figure 4.12 – ServiceEventMod class
Figure 4.13 – IdFilterSet class
Figure 4.14 – NameFilterSet class
Figure 4.15 – TypeFilterSet class
Figure 5.1 – Demonstrator 1 GUI
Figure 5.2 – Monitor class
Figure 5.3 – Monitor class with repository monitoring
Figure 5.4 – Repository monitoring in demonstrator 1
Figure 5.5 – State monitoring initialization
Figure 5.6 – State monitoring in demonstrator 1
Figure 5.7 – Manifest monitor initialization
Figure 5.8 – Manifest monitoring in demonstrator 1
Figure 5.9 – Old manifest version
Figure 5.10 – New manifest version
Figure 5.11 – Data field initialization and services registration
Figure 5.13 – Data field services unregistration
Figure 5.12 – Data field monitoring in demonstrator 1
Figure 5.14 – Monitor class with id filter
Figure 5.15 – Id filtering in demonstrator 1
Figure 5.16 – Monitor class with type filter
Figure 5.17 – Type filtering in demonstrator 1
Figure 5.18 – Demonstrator 2 GUI
Figure 5.19 – Monitor class with bundle state monitoring and filters
Figure 5.20 – Monitor constructor with service monitoring
Figure 5.21 – Service monitoring in demonstrator 2
List of tables

Table 2.1 – A list of the non-OSGi articles that are going to be reviewed
Table 2.2 – A list of the OSGi articles that are going to be reviewed
Table 2.3 – A classification on what to monitor for each reviewed article
Table 2.4 – A classification of when to monitor for each reviewed article
Table 2.5 – A list of external tool that each reviewed article uses
Table 5.1 – Traceability matrix
List of Abbreviations

API: Application Programming Interface
ASM: Assembler
CPU: Central Processing Unit
GUI: Graphical User Interface
HTTP: Hypertext Transfer Protocol
iPOJO: injected Plain Old Java Application
JAR: Java ARchive
JMX: Java Management Extensions
JNI: Java Native Interface
JVM: Java Virtual Machine
JVMTI: Java Virtual Machine Tools Interface
OSGi: Open Service Gateway initiative
RMI: Remote Method Invocation
XMLRPC: XML for Remote Procedure Call
1 Introduction

Chapter 1 presents the introduction of the report, by explaining its background, purposes and goals, and what approach is going to be followed to achieve the solution. First, in 1.1, it presents the context for this report. Section 1.2 presents the main goal of this project. In Section 1.3, it continues explaining the process that is going to be followed to solve this problem. Section 1.4 enumerates the constraints of this project. Finally, in 1.5, it presents the structure of this report.

1.1 Background

To understand this work, there are three important concepts that have to be explained. The first one is self-adaptive software systems. The second important concept is monitoring in self-adaptive software systems. The third important idea is the environment in which we are going to work: OSGi (Open Service Gateway initiative).

1.1.1 Self-Adaptive Software Systems

As software evolves, it is necessary to develop new solutions for managing larger systems. Self-adaptive software system becomes one of the approaches for autonomous managing of large and complex software. This kind of software can perform changes on itself without any human intervention. Those changes are made accordingly to the observation of the software state itself and its context [1]. This observation is made by the monitoring system.

1.1.2 Monitoring systems in Self-Adaptive Software Systems

Monitoring system has to be continuously providing information, fetched from the base-level system, or managed systems, and its environment. The information is directed to the self-adaptive unit, or management system. It will make the appropriate decisions and effect changes in the base-level system, if it is necessary [2].

1.1.3 OSGi

OSGi (Open Service Gateway initiative) is a dynamic component system for Java. It provides a set of specifications that define a modular architecture for applications. Those modules are called bundles. A bundle is a plain old JAR (JAvA Archive) file, but OSGi hides everything in it except explicitly exported. Bundles can provide or consume services to build service-oriented applications. A service is an instance of a Java class, which is registered by a bundle. Services are managed by OSGi to provide its functionality to other bundles. Both bundles and services, called OSGi components, can be added, replaced or removed during run-time on the execution environment that OSGi provides [3].

Using this technology can reduce complexity, enable reuse of components, and ease maintenance, among other advantages, in large Java applications [4].

There are several OSGi implementations. Some of them are Knopflerfish, Hitachi Super J, Eclipse Equinox or Apache Felix [5].

1.2 The problem

As we have described in subsections 1.1.1 and 1.1.2, to achieve self-adaptation, it is necessary to monitor base-level system and its environment on run-time.

To develop a self-adaptive software system in OSGi, we need to monitor different characteristics of OSGi applications and its environment. These characteristics or monitor objects are for example information about resources consumed by each bundle [6] or execution information about the behavior of a bundle [7]. As OSGi provides an
environment where bundles and services are added, modified and removed on run-time, it could also be interesting to provide updated information about the current state of these components [8].

OSGi framework [9] provides some tools to perform basic monitoring of its components. However, this functionality could be expanded by implementing a framework. This framework could provide useful, reusable and configurable methods to get information about OSGi components on run-time.

1.3 The process
The main objective of this project is to implement a Java framework to support development of component and behavior monitoring systems in OSGi applications, in the context of self-adaptive software systems. These monitoring systems will provide component and behavior information on run-time.

First step will be to make a state-of-the-art survey about monitoring systems, reviewing previous works in this area. This survey will explain the features that a monitoring system has to provide, such as what, when, where, why and how to monitor, as well as software attributes, API (Application Programming Interface) and performance.

Next step will be to write the requirements and design the prototype, taking the survey results and conclusions as the base to do so.

Finally, the framework will be implemented along with two demonstrator applications that will use the framework and will show how the tool works.

1.4 Restrictions
These restrictions are agreed with the stakeholders of this project. This project just focuses on monitoring systems, so the rest of the self-adaptive process is out of scope.

The tool will only perform component and behavior-based monitoring (see subsection 2.5.1).

The OSGi implementation selected for this project is Eclipse Equinox.

1.5 The structure of the report
This report is organized in six chapters. Chapter 1 is a presentation of the problem. Chapter 2 covers the preliminary state-of-the-art survey about monitoring systems. Chapter 3 contains the requirements of the framework. Chapter 4 shows the design and implementation details of the framework. Chapter 5 presents a guide on how to use the tool, using two demonstrator applications as support to do so. Chapter 6 explains the conclusions about the project.
2 Monitoring Systems: State-of-the-Art Survey

Chapter 2 presents the survey about monitoring systems in general and OSGi monitoring in particular.

In section 2.1, it introduces the process that is followed to make the survey. Section 2.2 presents self-adaptive software systems. Then, in 2.3, it explains monitoring systems with some examples of monitoring systems. Section 2.4 presents a list of previous works, in this case, about monitoring systems on OSGi. Section 2.5 contains the results of the survey. Finally, in 2.6, the conclusions of the survey are explained.

2.1 The process of the survey

The searching method for the articles reviewed in chapter 2 has been using Google Scholar, the strings used are “monitoring software”, “monitoring osgi”, “self-adaptive osgi” and “monitoring service-oriented” (OSGi can build service-oriented applications). Article [10] has been selected between other works because it is very flexible in its features, articles [11] and [12] have been selected because they are service-oriented and they could present various challenges relevant for this project. The rest of the works are OSGi related, and will be interesting for this survey. It does not pretend to be an exhaustive list, but an open list with some examples that are used to monitor using different procedures and objectives.

The articles reviewed for this survey are [10], [11] and [12] for monitoring of non-OSGi applications (section 2.2), and [4], [6], [7], [13], [14], [15], [16], [17], [18], [19] for monitoring of OSGi applications (section 2.4).

2.2 Self-Adaptive Software Systems

Self-adaptive software systems are able to adjust their behavior in response to their perception of the environment and the system itself [1]. A self-adaptive software system can perform changes on itself without any human intervention. Those changes are made according to the observation of the context of the software, and the software state itself.

![Figure 2.1 – Self-adaptive software system structure](image)

In Figure 2.1 we can see the structure of these systems. The Self-Adaptive Unit or Management System has to continuously monitor the behavior and environment of the
Base-Level or Managed System. After that, it will effect some changes in it if that is necessary.

2.3 Monitoring in Self-Adaptive Systems

Self-Adaptation can be achieved through monitoring the software system (self) and its environment (context) to detect changes, make appropriate decisions, and act accordingly [20]. The monitoring system has to be continuously providing information. This information is fetched from the base-level system and its environment. Then, it is directed to the self-adaptive unit so it can make the appropriate decisions and effect changes in the base-level system if it is necessary.

A similar definition is given in [2]. The monitoring system has to monitor the application and its environment, against some requirements. That means that the monitoring system has to direct the information to allow the self-adaptive system to satisfy some requirements.

The list of non-OSGi monitoring articles contains works introduced in section 2.1. These works are going to be briefly explained to show what kind of challenges faces a monitoring system in general, and what elements and features can be useful for this project. However, the implementation of these works is going to be skipped as it does not add anything important to this work since it is focused on OSGi.

<table>
<thead>
<tr>
<th>Short name</th>
<th>Author</th>
<th>Name of the article</th>
</tr>
</thead>
</table>

Table 2.1 – A list of the non-OSGi articles that are going to be reviewed

Table 2.1 gives the short names that these articles are going to be refereed as in the rest of the survey.

2.3.1 [IMSERTM]

Thomas Vogel et al. [10] propose a model-driven monitoring system, which means that it is going to be platform-independent. This system is oriented to transform the low level information from sensors, at the abstraction level of their APIs, to a higher level model, which will be utilized by the management system. This high level model can be used to analyze the behavior of the application for failure monitoring, and resource usages for performance monitoring. Sensors collect the information from the managed system in a pull-oriented or push-oriented way, which means that the updates can be triggered by some events or by the user. Layer architecture is used to transform the low level data from sensors to the high level. The information is transformed to make it independent from the managed system.

Thomas Vogel et al. [10] present several ideas that are going to be relevant for this project. The first one is the transformation of the data from the various types of sensors to a normalized format using layer architecture. This will give some flexibility and independence from self-adaptive unit to base-level system. The second concept could be what it can monitor, in this case is resource and fault monitoring. We can extract from this that there are several monitor objects and one tool can be used to monitor one or several of them. Another important thing is that this tool enables users to monitor the application using a pull-oriented or a push-oriented approach. That means that it enables
to extract the information when the user needs it or when there is some important change in the application. The combination of both types of extracting the information is a good solution that could give the user more flexibility.

2.3.2 [RBMSASO]
Harald Psaier et al. [11] introduce a monitoring system for service-oriented applications. It can monitor the behavior of services in an application. It is divided in two parts: Control Interface and Admin Tools. The Control Interface provides a functional grounding in which the user can add plugins. These plugins monitors the managed system itself. The Admin Tools have to be continuously collecting the information from the Control Interface and storing it in a log. It creates events of interest from the log and registers the behavior of the services and the interactions between them.

Harald Psaier et al. [11] present several characteristics that are going to be relevant for this project. In the first place, the divided architecture enables to have one part focused in the individuality of the base-level system while the other part normalizes the data for the self-adaptive unit. The plugin system is interesting, as it enables to expand the functionality of the monitoring system. Also, the event system used in this work can be an interesting approach. The log registers everything and can be used to create events taking just the relevant data from it.

2.3.3 [FRTMSO]
Cuiting Chen et al. [12] propose another monitoring system for service-oriented applications. It focuses on how services interact to create a runtime topology from the application and show dependencies between services. The information that it can monitor is the service's identification parameters, the operation that is performing and its parameters. That information is gotten by extending the service framework to obtain different data, so it can log the information when it performs an action. They also propose to extend functionality of the monitoring system by providing more data about the services.

The authors present some important relevant issues for this project. In the first place, it can monitor the runtime topology of a service-based application. They do this by monitoring the interaction between services. This could be a useful area to monitor in a service-based application. Moreover, the extension to the service framework is a good way to monitor an application without changing anything in its source code, as it provides a decoupled monitoring system from the application. However, this method creates a new dependency on a modified service framework. Finally, adding more information about the services, instead of only the interactions between services could provide a monitoring system with more possibilities.

2.4 Monitoring Systems in OSGi
The list of OSGi monitoring articles contains works introduced in the beginning of chapter 2. They are going to be reviewed to show what kind of challenges faces an OSGi monitoring system in particular, and what elements and features can be useful for this project. In some cases, it will include implementation details when they are relevant for us.
<table>
<thead>
<tr>
<th>Short name</th>
<th>Author</th>
<th>Name of the article</th>
</tr>
</thead>
<tbody>
<tr>
<td>[RMVOSGi]</td>
<td>Tuukka Miettinen</td>
<td>Resource monitoring and visualization of OSGi-based software components [13, 14]</td>
</tr>
<tr>
<td>[MMROSGi]</td>
<td>Ikuo Yamasaki</td>
<td>Monitoring and managing Computer Resource Usage on OSGi Frameworks [6]</td>
</tr>
<tr>
<td>[CMOSGi]</td>
<td>Tao Wang et al.</td>
<td>Component Monitoring of OSGi based Software [16]</td>
</tr>
<tr>
<td>[KOSGi]</td>
<td>-</td>
<td>Keywatch, an OSGi based open source monitoring system [17]</td>
</tr>
<tr>
<td>[OSGiCJRM]</td>
<td>Bruno Van Den Bossche et al.</td>
<td>An OSGi compatible implementation of a Java resource monitoring [18]</td>
</tr>
<tr>
<td>[TIAOSGi]</td>
<td>Nicolas Geoffray et al.</td>
<td>Towards a new Isolation Abstraction for OSGi [19]</td>
</tr>
<tr>
<td>[R-OSGi]</td>
<td>Jan S. Rellermeyer et al.</td>
<td>Building, Deploying, and Monitoring Distributed Applications with Eclipse and R-OSGi [8]</td>
</tr>
</tbody>
</table>

Table 2.2 – A list of the OSGi articles that are going to be reviewed

Table 2.2 gives the short names with that these articles are going to be refereed as in the rest of the survey.

2.4.1 [A-OSGi]
João Ferreira et al. [15] propose a framework named A-OSGi that leverages the features of the OSGi platform to support the implementation of self-adaptive systems. A-OSGi can monitor resource usage of a bundle, component availability and how services bind with each other at runtime. It interacts with the service and bundle layers, as well as with the JVM (Java Virtual Machine) to get all that information. It generates an event when a relevant change occurs in the system. Those events are sent to all components previously subscribed to the monitoring system. An HTTP (Hypertext Transfer Protocol) server enables A-OSGi to support deployment of web applications monitoring the requests received by the server. The interface of the monitoring system is exported as JMX (Java Management Extensions) Managed Beans. JMX Managed Beans is another Java technology for managing and monitoring applications. Its API is taken as a model in A-OSGi. It uses Apache Felix as OSGi platform. Monitoring of resources is doing by isolating bundles. That is, separate the resources consumed by each one of the bundles. This is achieved creating a hierarchy of threads named ThreadGroup, which relates each bundle with the threads it creates. For this reason, it was necessary to modify the life cycle layer of OSGi. Also, clients of a service call a proxy that executes the service in a thread associated with the bundle that registered the service. Once the isolation is assured, CPU (Central Processing Unit) and memory can be obtained using JVMTI (Java Virtual Machine Tools Interface). JVMTI is a tool that enables to monitor the JVM resource usage. The monitoring of services life cycle is done by interacting with the bundle and service layers as was pointed out before. For monitoring relations between services, A-OSGi uses iPOJO (injected Plain Old Java Application). iPOJO is a tool that allows monitoring and management of services binding in runtime.
João Ferreira et al. [15] present two important issues for an OSGi monitoring system. The first one is the monitoring object. It can monitor service bindings and life cycle, as well as resource usage of bundles. For the service monitoring it just uses the tools that the OSGi platform provides and iPOJO. Resource monitoring is more complicated. It needs isolation of bundles. Without this isolation, a client service could add the consumption of the services it consumes, merging measurements of various services. Isolation is provided by a structure that storage the relations between bundles and threads and by using proxies. This system has some limitations as it is explained in the paper, it does not isolates interactions that does not uses service interfaces (OSGi allow this) and it has an overhead for the system. For fetching the resource usage information, it uses JVMTI. The second issue that this article presents is the use of JMX Managed Beans as model for the software interface. This will allow compatibility with other tools that uses this system.

2.4.2 [LLMOSGi]
Giulio Caravagna et al. [7] present a tool for monitoring OSGi applications with low resource utilization. It can monitor the behavior of a bundle, inspecting its execution log, where the bundle can write a sequence of its operations. The monitoring system have scheduled when it has to monitor and will suspend the monitoring process the rest of the time. It uses Eclipse Equinox as OSGi platform. For the log system, each bundle has to use Commons Logging API. That tool offers a framework that enables to write an execution log in runtime. The monitoring system itself can be scheduled to read that log.

Giulio Caravagna et al. [7] introduce some new concepts. One of these concepts is that the monitoring object is the behavior of a bundle. For achieving this, each bundle has to register its actions in a log, using the Commons Logging API. Another concept that presented is the monitoring system that can be activated and suspended. This article claims to provide a lower overhead than other runtime methods thanks to that. However, it could miss relevant information while the monitoring system is suspended.

2.4.3 [RMVOSGi]
Tuukka Miettinen [13, 14] introduces a tool for monitoring of OSGi applications performance during runtime. It can monitor CPU and memory usage of an isolated bundle. It can get this data both in an event and get-based way. This implies that it can raise an event when there is any change in the system or when the user requests such information. It uses the Oscar platform. This tool uses a ThreadGroup object, similar to the João Ferreira et al. [15] proposed one, to identify and isolate bundle performance. To monitor services also uses a proxy that implements the same interface or interfaces of the bundle. Any component calling the methods of the bundle has to call the proxy first. To achieve both ThreadGroup and the proxies, it was necessary to modify Oscar. When a bundle is started, it registers it in the ThreadGroup object and creates its proxy. For getting the monitoring information itself, it uses JVMTI to monitor the CPU usage and byte-code instrumentation in all classes to get the memory consumption.

Tuukka Miettinen [13, 14] presents some relevant characteristics for performance monitoring. Some solutions are the same as in João Ferreira et al. [15], as the use of JVMTI and the bundle isolation system. The difference is the get-based and event-based monitoring system. This feature will provide flexibility to the user, and it is better than just use one of them. Finally, the byte-code instrumentation will help the JVMTI to monitor the memory usage, and it is a way to add instrumentation to the application without modifying the source code of it. Besides, it could be useful for other purposes as well.
2.4.4 [MMROSGi]

Ikuo Yamasaki [6] proposes a tool for monitor resource usage on OSGi applications. It can monitor CPU usage of an isolated bundle. Their preliminary requirements are three: the tool has to be lightweight, without rewriting source code and platform independent. This tool uses JMX Managed Beans to monitor every thread. To identify and isolate each bundle, it has to be registered in a structure named Bundle-Thread-Tree, similar to the João Ferreira et al. [15] or Tuukka Miettinen [13, 14] ones. When a bundle is created, it has to be reported to the Thread Manager. It uses Byte Code Weaving to not alter the source code as in the Tuukka Miettinen [13, 14] article, but in this case, it uses AspectJ for byte-code instrumentation. It gets the monitoring data of the application periodically.

The Ikuo Yamasaki [6] approach is similar to the rest of the tools for performance monitoring. In this case it uses JMX Managed Beans, which allows management and monitoring of a target previously registered. The most relevant thing this project presents is the requirements of the tool. First, the monitoring system has to add minimal overhead to the base-level application. Also, any bundle has to be able to be monitored without altering anything in its source code. Finally, it has to run on any OSGi framework. These requirements are very desirable for any monitoring system.

2.4.5 [CMOSGi]

Tao Wang et al. [16] focus on a tool for monitoring OSGi components. It can monitor CPU and memory usage of an isolated bundle, and the interaction between bundles. To monitor the interaction between bundles it registers the creation, interaction and invocation frequency of its services. It uses Apache Felix platform as OSGi implementation. Its structure has four parts: JVM Monitor, OSGi Extension, Bundle Tracker and Bundle Console. JVM Monitor is implemented with native code. It tags threads, objects and calculate resources using JVMTI, like the rest of the tools for resource monitoring that we have seen. The monitoring data is stored in an array. OSGi Extension is based in Felix kernel. It generates a proxy using ASM (Assembler) to insert byte-codes to trace the thread transfer between bundles. Proxies are used as interfaces for the services. Bundle Tracker is meant to allow communication between the Java application and the JVM Monitor using JNI (Java Native Interface). JNI will allow us to match the JVM Monitor, which is programmed in C, with the rest of the Java application. Finally the Bundle Console is used for visualization and user interaction.

The authors present an interesting point for the implementation of this this project. This is the interaction with the service registry. This will be useful for monitoring of OSGi services.

2.4.6 [KOSGi]

Keywatch [17] is an OSGi monitoring system. It uses a plugin system. Those plugins provides the monitoring functionalities. Keywatch is focused on the extensibility and flexibility that the plugin system provides. Different plugins use normalized events which contain the monitoring information. To manage the diverse plugins, users can use the Task Controller to start, stop or schedule plugins. There are three kinds of plugins that can interact with the Event Server: Providers, Clients and Rules. Providers send events to the Event Server. Clients are the event consumers from the Event Server. Rules can be used to analysis the events from the Event Server. The Keywatch implementation includes a client and a provider samples. Client is a web client built with Ajax. Clients function is to get access to the services from Event Server. The XMLRPC (XML for Remote Procedure Call) provider is a plugin included in the core
product. It makes it easy for the user to start using Keywatch. XMLRPC enables to run Perl scripts to do the observation and provides a Java OSGi agent that reports if any bundle is started or stopped.

Keywatch [17] architecture could be interesting for this project. The plug-in system for clients and providers is a good system that enables extension of functionalities to the user, instead of focusing on the system in the observation of one or two properties of the system.

2.4.7 [OSGiCJRM]
Bruno Van Den Bossche et al. [18] propose an API for developing platform independent monitoring systems. It can monitor resource usage of a bundle. It does not specify an OSGi platform. It is composed by two parts. The first one is an API that provides read access to the resources, with a single interface for each resource. The second one uses the resources API and raise and event when some conditions meet. A listener can be registered to the monitor and see when those events rise. The monitoring framework uses several event-based conditions. One can raise an event if a resource usage crosses a threshold. Another one when there is a certain change in resource usage. A third one can raise an event when a specified property of the resource changes.

In Bruno Van Den Bossche et al. [18] article, we found again an extensible system like the resources API, which actually provides a different API per resource. Another interesting feature is the event system, which can be triggered when one of these conditions is met: the resource crosses a threshold, it suffers an important change or it simply changes. These events are directed to a previously registered listener.

2.4.8 [TIAOSGi]
Nicolas Geoffray et al. [19] propose a method to isolate OSGi components that can be used for monitoring them. It modifies the compiler to add a new parameter to each Java method without changing the source code. With this parameter, the method knows its domain and its identifier. Each class has an array with its private instances to fetch instances and private Java static variables. To fetch strings it uses ldc for byte-code instrumentation. The compiler is modified to add local variables to Java methods, these variables will point to the domain the method belongs to.

Nicolas Geoffray et al. [19] do not present a monitoring system itself. However, they introduce a method for isolation of OSGi components that can be useful to monitor resource consumption of such components.

2.4.9 [R-OSGi]
Jan S. Rellermeyer et al. [8] present an Eclipse plugin to monitor OSGi applications named R-OSGi. It can monitor the current state of bundles and services, changes in their states, and communication between services. R-OSGi can monitor any change in services and bundles. It is oriented to distributed applications. It uses an agent to communicate with the OSGi framework. These agents are contacted to get the information. Changes in the state of bundles and services are propagated via OSGi EventAdmin service [5].

The authors identify the need of having a tool in Eclipse that helps the OSGi developers to have information about the topology of the application. This is solved with an Eclipse plugin. However, they do not provide the implementation details.

2.4.10 [OSGiD]
Alexandre de Castro Alves [4] uses the JMX tool for monitoring OSGi applications. With this tool, it can monitor the state of bundles, services and data about imported and
exported packages. This can be done both when the user of the framework requires the information and when a notification triggers. To monitor the different components, it is necessary to indicate the information about what to monitor to a server named MBeans. This server works as a local agent that interacts with the managed system. A remote client can retrieve the information by various transport protocols, such as RMI (Remote Method Invocation).

Alexandre de Castro Alves book [4] is not focused on monitoring system, but dedicates a chapter to it. In this work we can see some of the recurring topics in the rest of articles: get-based, event-based monitoring and JMX.

2.5 Results of the survey
This is a summary of the survey. The information extracted will be gathered to answer the questions what, why, when, where and how to monitor. This is done by comparing the different problems and solutions.

2.5.1 What to monitor
What to monitor covers the different objectives the articles focus on. That is, the various characteristics of base-level application and environment information that the systems can get. The tools that have similar monitoring objectives, faces the same problems and features similar or different solutions. We are going to see the problems first in this section, while solutions are in subsection 2.5.5, on how to monitor.

The articles reviewed can be gathered in four types on what to monitor: resource, component, behavior monitoring and other. The first three types and the special cases are explained after Table 2.3. The other category is included because, as it can be seen in the survey, what to monitor is an open subject and some of the articles are focused on extensibility of their features.

<table>
<thead>
<tr>
<th>Article</th>
<th>Resource monitoring</th>
<th>Component monitoring</th>
<th>Behavior monitoring</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>[IMSSERTM]</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>[RBMSASO]</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>[FRTMSO]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[A-OSGi]</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[LLMOSGi]</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>[RMVOSGi]</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[MMROSGi]</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[CMOSGi]</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>[KOSGi]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>[OSGiCJRM]</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[TIAOSGi]</td>
<td>?</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>[R-OSGi]</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[OSGiD]</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

Table 2.3 – A classification on what to monitor for each reviewed article

In resource monitoring are all the articles that can provide data about the system resources that the application is consuming. For João Ferreira et al. [15] article, it is CPU and memory; for Tuukka Miettinen [13, 14] paper it is the same, CPU and
memory; Ikuo Yamasaki [6] just CPU; Tao Wang et al. [16] CPU and memory; Keywatch [17] is a special case and will be discussed later in this subsection; Bruno Van Den Bossche et al. [18] approach use an open system, it includes CPU and memory, but some others can be added, as network utilization; Nicolas Geoffray et al. [19] article is also a special case.

In component monitoring are all the articles that can give information about the OSGi components (services and bundles) and the relations between them. João Ferreira et al. [15] article focus on monitoring both bundle and service availability and the relations between them; Tao Wang et al. [16] work is a similar case, and is also focused on monitoring the interaction between bundles; Keywatch [17] is a special case and will be discussed later in this subsection; Jan S. Rellermeyer et al. [8] exclusively monitor the topology of the application, which is services and relations between them; Alexandre de Castro Alves [4] system can monitor information about bundles and services, and also importing and exporting of packages; Cuiting Chen et al. [12] article is non OSGi, but is service-oriented, it monitors the topology of the application and the relation between services.

In behavior monitoring are the tools that can monitor the execution of a bundle, therefore, the actions that they perform. Giulio Caravagna et al. [7] article focus in an execution log of a bundle; Harald Psai er et al. [11] paper is non OSGi, but it also monitors the execution log of a service; Keywatch [17] is a special case.

The special cases are Keywatch [17] and the work by Nicolas Geoffray et al. [19] Keywatch [17] is focused on extensibility. Its plugin system, explained in 2.4.1, can add functionalities on what to monitor. With these plugins, it can monitor any of the types explained and even others. The work of Nicolas Geoffray et al. [19] does not specify what to monitor, its functionality could be used for resource monitoring, but it could be used for other methods as well.

2.5.2 Why to monitor

Why to monitor is partially answered in subsection 2.2. Self-adaptive software systems are able to make changes in themselves accordingly to the observation of the system and its environment [1]. Therefore, monitoring process is necessary to take the appropriate decisions in each case. However, this explanation can be extended for each kind of monitoring explained in 2.5.1.

In the case of the resource monitoring articles, Ikuo Yamasaki [6] explains that this information is necessary to avoid that some threads could deplete the computer resources. Especially when those threads comes from services that we do not control, and that could be malicious or poorly implemented.

In the case of component monitoring, João Ferreira et al. [15] explain that it could support to describe the autonomic behavior of an OSGi application. This information will allow understanding the topology of an OSGi application. That is, the bundles, services and the relations between them. Having control over the different components, that are part of it and their life cycle, can help to manage bundles and services as Jan S. Rellermeyer et al. [8] states.

In the case of behavior monitoring, Giulio Caravagna et al. [7] explain that it could be used to store a user interaction log, analyze that log, and detect security violations. Harald Psai er et al. [11] explain that an application behavior log could be used to detect failures and malfunctions too.

2.5.3 When to monitor

When to monitor is going to cover the decision about when the monitoring system has to extract data about the base-level system and its environment. This is solved mainly in
two distinct ways in the articles: get-based monitoring and event-based monitoring. These two are explained after Table 2.4. The other type is for the non-applicable cases.

<table>
<thead>
<tr>
<th>Articles</th>
<th>Get-based monitoring</th>
<th>Event-based monitoring</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>[IMSERTM]</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>[RBMSASO]</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>[FRTMSO]</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>[A-OSGi]</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>[LLMOSGi]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[RMVOSGi]</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>[MMROSGi]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[CMOSGi]</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>[KOSGi]</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>[OSGiCJRM]</td>
<td>?</td>
<td>?</td>
<td>x</td>
</tr>
<tr>
<td>[TIAOSGi]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[R-OSGi]</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>[OSGiD]</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4 – A classification of when to monitor for each reviewed article

In get-based monitoring are all the articles that can perform the monitoring in a previously set time. This can be a period like in the article by Ikuo Yamasaki [6], or with an asynchronous method, like the pull-oriented monitoring of Thomas Vogel et al. [10], where the user can decide when to get the information.

In event-based monitoring are all the articles that perform the monitoring when some event happens. There can be several kinds of events, depending of the change in the system that raised an event, like in Bruno Van Den Bossche et al. [18] work. This work present the most complete solution of all the event-based articles, featuring several types of triggers: one can rise an event if a resource usage crosses a threshold, another one when there is a certain change in resource usage and a third one when a specified property changes.

Thomas Vogel et al. [10] or Alexandre de Castro Alves [4] give a flexible solution, as they allow the user to monitor with an event-based system, but also provide to the user the possibility of get the information asynchronously when the user wants. The special cases are again Keywatch [17] and Nicolas Geoffray et al. [19] article. Keywatch [17] does not specify it, but it could support both kinds of monitoring. In Nicolas Geoffray et al. [19] article, this feature is not specified, but it could support both of them.

2.5.4 Where to monitor

In where to monitor, we are going to present methods for instrumentation. As Ikuo Yamasaki [6] states in their requirements, it is important to monitor an application without changing its source code.

Sometimes, the tools that the OSGi platform and the JVM provide are not enough to perform the monitoring process. Because of that, some works like Tuukka Miettinen [13, 14], Ikuo Yamasaki [6], Tao Wang et al. [16] and Nicolas Geoffray et al. [19] use byte-code instrumentation. That enables to add instrumentation at compiler time or in
runtime to perform actions as logging or raising events without changing the source code. Those tools perform these operations using an external framework as AspectJ, ASM or ldc.

### 2.5.5 How to monitor

In *how to monitor*, techniques, architectures and tools of the articles are going to be discussed. First, we are going to explain the common techniques between the different articles. For this, we are going to separate the tools according to the classification in 2.5.1 about *what to monitor*.

Tools that perform the resource monitoring faces the same problem, to monitor each bundle it is necessary to isolate that bundle. Tuukkan Miettinen [13, 14] explains that this is mandatory if the monitoring tool has to get the resource of just that bundle, and not append the resources of the services that it consumes. To make the resource monitoring, this system has to be aware of the bundles and the threads that it creates. This information is stored in a data structure, named *ThreadGroups* by João Ferreira *et al.* [15], *ThreadGroup* by Tuukkan Miettinen [13, 14] or *Bundle-Thread-Tree* by Ikuo Yamasaki [6]. To provide the isolation, these tools also use a proxy system. The proxy system implements the same interface as the service that is isolating. With this system, the monitoring system can separate the resources that the bundle is consuming from the resources that other services are consuming.

The tools that perform component monitoring fetch that information from the OSGi bundle, services and lifecycle layers. In the case of João Ferreira *et al.* [15] work, it interacts with the service and bundle layers. Tao Wang *et al.* [16] system interacts with the service layer, specifically with the service registry.

The tools that perform behavior monitoring like the one presented by Harald Psaier *et al.* [11], or Giulio Caravagna *et al.* [7], uses instrumentation to register changes in their execution logs.

Some of the articles present interesting architectures. In special, the extensible architecture presented by Bruno Van Den Bossche *et al.* [18] and Keywatch [17]. In these cases, it is easy to add more functionality to the monitoring system, to provide more resource objectives in the first case, or more objectives of any type in the second one. The other tools are focused on one kind of monitoring objective. Harald Psaier *et al.* [11] system is divided in two parts, and the first one is a plugin system like the one that Keywatch [17] uses, that part interacts with the base-level system, the second part manages the data taken from the first part and normalizes it. Thomas Vogel *et al.* [10] presents a layer architecture in which the low level information taken from the sensors is transformed to a higher level, that high level information is normalized and sent to the management system.

Finally, the tools that are used are: JMX Managed Beans for providing resource monitoring of threads and a standard API; JVMTI for JVM resource monitoring; iPOJO for service binding monitoring; Common Logging API for logging of bundles; AspectJ, ASM and ldc for byte-code instrumentation; JNI to communicate C code with Java; Google Web Toolkit and Jetty Servlet Container for creating a client-server architecture; and RMI to create a client-server architecture allowing to call methods remotely.
<table>
<thead>
<tr>
<th>Articles</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>[IMSERTM]</td>
<td>-</td>
</tr>
<tr>
<td>[RBMSASO]</td>
<td>-</td>
</tr>
<tr>
<td>[FRTMOSO]</td>
<td>-</td>
</tr>
<tr>
<td>[A-OSGi]</td>
<td>iPOJO, JVMTI and JMX Managed Beans</td>
</tr>
<tr>
<td>[LLMOSGi]</td>
<td>Common Logging API</td>
</tr>
<tr>
<td>[RMVOSGi]</td>
<td>JVMTI</td>
</tr>
<tr>
<td>[MMROSGi]</td>
<td>JMX Managed Beans and AspectJ</td>
</tr>
<tr>
<td>[CMOSGi]</td>
<td>JVMTI, ASM and JNI</td>
</tr>
<tr>
<td>[KOSGi]</td>
<td>Google Web Toolkit and Jetty Servlet Container</td>
</tr>
<tr>
<td>[OSGiCJRM]</td>
<td>-</td>
</tr>
<tr>
<td>[TIAOSGi]</td>
<td>ldc</td>
</tr>
<tr>
<td>[R-OSGi]</td>
<td>-</td>
</tr>
<tr>
<td>[OSGiD]</td>
<td>JMX Managed Beans and RMI</td>
</tr>
</tbody>
</table>

Table 2.5 – A list of external tool that each reviewed article uses

### 2.6 Conclusions from the survey

The conclusions from the survey can be split in various areas that will be relevant for designing our framework. These areas are extracted from results of the survey in section 2.5. Subsection 2.6.1 covers what to monitor. Subsection 2.6.2 comprises when to monitor. Subsection 2.6.3 applies to where to monitor. Finally, Subsections 2.6.4, 2.6.5 and 2.6.6 cover how to monitor.

#### 2.6.1 Monitoring various components

A system that is able to monitor different OSGi components at the same time is desirable for most of the articles. An important feature that JMX allows, and it is explained in Alexandre de Castro Alves [4] is that the monitored OSGi components can be specified by the user. Therefore, it is important to support mechanisms that can get information about one or several components.

#### 2.6.2 Get-based and Event-based monitoring

In subsection 2.5.3 about when to monitor, we present get-based and event-based monitoring. There are some works, for example Ikuo Yamasaki [6] or Thomas Vogel et al. [10], which use Get-based monitoring. It enables to get the monitoring information when the user wants. Also, it can be used to get this data periodically. This method gives flexibility to the user, but it could be inefficient if we wanted updated information to be available as soon as some change happens. On the other hand we have event-based monitoring. Works like Bruno Van Den Bossche et al. [18] use this method. It has the advantage of being faster. As it triggers events as soon as these events happen. A more flexible solution is used in Thomas Vogel et al. [10] or Alexandre de Castro Alves [4], which consists on offering both get-based and event-based monitoring.

#### 2.6.3 Decoupling

One of the requirements of the tool that Ikuo Yamasaki [6] presents is that the base-level application does not have to be changed to implement the monitoring system. Generally, most of the works try to do so. When it is inevitable, they use byte-code
instrumentation. Tuukka Miettinen [13, 14], Ikuo Yamasaki [6], Tao Wang et al. [16] and Nicolas Geoffray et al. [19] works use it. It allows inserting instructions into the bundles without changing the source code. Lose coupling with the base-level is going to be another requirement, using byte-code instrumentation when necessary.

2.6.4 API
The API that the framework offers is important. API has to offer data in a normalized way. That is, the information has to be brought into conformity with a standard that will be used, in this case, by the management system. Works like Keywatch [17] transform raw information into events. Thomas Vogel et al. [10] also transform data from the sensors and normalize it for the management system.

2.6.5 Efficiency
Efficiency is a common concern in almost all the works. The insertion of a monitoring system in an application could disturb the information gathered by this system [21]. For example, a poorly implemented resource monitoring system may alter the resource consumption. Works like Ikuo Yamasaki [6] emphasizes the need of creating a monitoring system that adds low overhead to the managed system. This is not only applicable in resource monitoring systems. Other types of monitoring like the one proposed by Harald Psaier et al. [11] needs to provide an efficient system. Some works even have resource consumption tests, like Giulio Caravagna et al. [7] or Tuukka Miettinen [13, 14]. So efficiency is also a desirable feature for this project. That means low resource consumption, low overhead to the managed system and fast response.

2.6.6 Extensibility
Some of the works like Bruno Van Den Bossche et al. [18], Keywatch [17] or Harald Psaier et al. [11] present an extensible and flexible architecture that allows expanding their functionalities. On the other hand, articles like Tuukka Miettinen [13, 14] or Ikuo Yamasaki [6] focus on one kind of monitoring. Extensibility and flexibility is going to be a desirable property for this project. There are various types of monitoring in subsection 2.5.1 classification, like resource monitoring. Inside this type, there are several monitoring possibilities like CPU, memory or network. Using an extensible architecture like Keywatch [17] could be interesting because functionalities of a monitoring system could grow in that way.
3 Requirements Specification

Chapter 3 presents the requirements of the monitoring framework. These requirements have been mainly proposed by the stakeholders of this project, while some has been extracted from results and conclusions of the survey, presented in Chapter 2.

Section 3.1 starts with a general description of the framework. Section 3.2 contains a list of the functionalities for this framework. Section 3.3 presents the interface requirements. Section 3.4 lists the performance requirements. Section 3.5 contains the requirements of the demonstrator applications. Finally, section 3.6 presents the constraints for this framework.

3.1 Overall description
This project is about developing a framework for monitoring of OSGi applications, to support development of self-adaptive systems. Therefore, this framework has to provide an API, or set of classes and interfaces, for an OSGi developer.

To show that the framework meets the requirements, two demonstrator applications will be implemented. These demonstrators will cover all features specified in chapter 3. Demonstrator 1 will cover bundle monitoring features. Demonstrator 2 will cover services, bundles and component communication monitoring features.

3.2 Functional requirements
This section contains a list of functionalities for the framework.
3.2.1. The framework provides support to monitor any change in the bundle repository, or bundle installation directory, and reports if any of the installed bundles is removed, added, or replaced (updated).
3.2.2. The framework provides support to monitor any change in the state of the bundles, and reports if any bundle is installed, started, stopped, updated or uninstalled.
3.2.3. The framework provides support to monitor any change in the installed bundles MANIFEST.MF properties, and reports if there is any change in them.
3.2.4. The framework provides support to monitor attributes of different components, for example a java class, within a bundle, and reports if there is any change in its data fields.
3.2.5. The framework provides support to monitor if there is any change in the service registry and reports if any service is registered or unregistered. For example, if a bundle is consuming a weather forecast service, and a new implementation for that service becomes available, then the monitoring should be able to report to that bundle that a new weather service implementation is now available.
3.2.6. The framework provides support to monitor if there is any communication, exchange of messages, between two OSGi components (bundles or services). This is an optional requirement.
3.2.7. The user is able to specify what type or types of events will be monitored. For example, the user can specify that the framework allows removed events and blocks the rest of events while bundle repository monitoring is working.
3.2.8. The user is able to specify what OSGi component or set of components will be monitored.

3.3 Interface requirements
This section presents the software interface requirements.
3.3.1. The framework provides a well-defined API for OSGi developers.
3.3.2. The framework API is completely documented. Considering OSGi developers as users of the framework. This means that the documentation should enable the developers to understand it, and use it to develop their own monitoring systems of OSGi based applications.

3.4 Quality attributes
This section contains the quality attributes: a list of requirements that the framework has to meet.
3.4.1. Different monitors supported by the framework should be able to detect and report changes in the object that they are monitoring in less than 45 seconds.
3.4.2. It is independent from the base-level system. It is not allowed to change the source code of it to implement the monitoring system.
3.4.3. It is extensible: Considering OSGi developers as users of the framework, the user can add new functionality.

3.5 General requirements
This section presents the requirements for two demonstrator applications that will use the framework.
3.5.1. Demonstrators 1 and 2 provide a GUI (Graphical User Interface).
3.5.2. Demonstrator 1 shows an updated report of the information specified in 3.2.2, 3.2.4 and 3.2.5.
3.5.3. Demonstrator 1 enables the user make the changes specified in 3.2.2, 3.2.2, 3.2.4 and 3.2.5.
3.5.4. Demonstrator 2 shows an updated report of the information specified in 3.2.6 and 3.2.6.
3.5.5. Demonstrator 2 enables the user make the changes specified in 3.2.6 and 3.2.6.
3.5.6. Demonstrators 1 and 2 enable the user to specify what components and types of events to monitor, as it is specified in 3.2.7 and 3.2.8.

3.6 Design constraints
This section contains the design constraint. It is made to narrow the implementation problems and focus on the functionality.
3.6.1. It works on the Equinox OSGi implementation.
4 Implementation

Chapter 4 explains the design and implementation of the framework. The framework is implemented following the requirements of chapter 3. All the implementation decisions are justified in chapter 4 following those requirements, and explained using conclusions of chapter 2 when necessary. However, chapter 4 will not check if the requirements are met correctly. This will be done in chapter 5.

Firstly, section 4.1 presents the package structure, software interfaces and architecture of the framework. Sections 4.2 and 4.3 explain the two packages available for the user, and the features that they offer.

4.1 Package organization

This framework is organized in two packages: osgi.framework.monitoring.event and osgi.framework.monitoring.event.filter. In the first one, we can find all the classes related to the monitoring process. In the second one, we can find the filter classes, which will allow the user to filter events in different ways.

Figure 4.1 – Package organization of the monitoring framework

In figure 4.1, we can see the package organization of the framework, and the classes within the two packages. Classes are represented by rectangles with its name for simplifying reasons. Arrows represent the inheritance relation between classes.

Let us start with the event package (osgi.framework.monitoring.event). In this package, there are two kinds of classes: the monitor classes and the event classes.

All monitor classes extends from EventMonitor (see Figure 4.2), a class with common functionality for all monitor classes. This is done for extensibility purposes (requirement 3.4.3).
**class** EventMonitor **extends** Observable

```java
EventMonitor(BundleContext bundleContext);
void setTypeFilterSet(TypeFilterSet fs);
TypeFilterSet getFilterSet();
void setBundleFilterSet(BundleFilterSet fs);
BundleFilterSet getBundleFilterSet();
BundleContext getBundleContext();
void notify(EventObject eo);
void checkUpdate(EventObject eo, int type, Bundle bundle);
```

Figure 4.2 – EventMonitor class

*EventMonitor* class extends from *Observable* class. That means that all monitor classes are *Observable* as well. This will be important when using the framework. It will be necessary to use the *Observer* pattern [23], as will be seen in the next chapter. Each monitor class focuses on one of the features from requirements 3.2.2 to 3.2.6. They are implemented to follow the quality requirements of section 3.4. Also, an event approach has been selected (see subsection 2.6.2). Monitor classes perform the monitoring process and raise an event object if necessary.

There is one event class for each monitor class. Event classes extend from *EventObject*, a *java.util* class, and will contain monitoring data. This is done for normalization purposes (see subsection 2.6.4). Both monitor and event classes will be presented and explained in section 4.2, as well as their implementation details.

The other package, *event.filter* (*osgi.framework.monitoring.event.filter*), contains the filter classes. There are two types of filter classes: bundle filter classes and a type filter class. Each type focuses on one of requirements 3.2.7 and 3.2.8. Bundle filter classes extend from a parent class named *BundleFilterSet*. This is done for extensibility purposes (requirement 3.4.3). Bundle filter classes can filter event objects depending on the bundle that raised it.

In the other hand, type filter class can filter events depending on the type of the event. For instance, a type filter class can disable all the events but the stopped and started ones in a state monitor. The state monitor will therefore raise just those kinds of events.

Filter classes will be presented and explained in section 4.3, as well as their individual implementation details and how they are related to the *event* package classes.

### 4.2 event package

Event package contains all classes related to the monitoring process.

This section is divided by features extracted from requirements 3.2.2 to 3.2.6. Each section contains an explanation of the feature, the classes from the framework that are available for the user, with a brief explanation of the important methods, and its internal implementation details.

#### 4.2.1 Repository monitoring

This feature is performed by using two classes within the *event* package: *RepositoryMonitor* and *RepositoryEvent*. *RepositoryMonitor* will monitor the bundle repository, and will raise an event if any JAR file of any installed bundle is added, modified (updated) or deleted. The bundle repository will be automatically set from the current bundle folder. That means that it will identify the directory in which the repository monitor is. This feature corresponds with requirement 3.2.2 from chapter 3.
**class** RepositoryMonitor **extends** EventMonitor **implements** Observer
{
    RepositoryMonitor(BundleContext bundleContext);
    void typeFilterSetDefaultConfiguration(TypeFilterSet tf);
    void update(Observable obs, Object obj);
}

Figure 4.3 – RepositoryMonitor class

RepositoryMonitor extends from EventMonitor class (see section 4.1). Therefore, it is an Observable class. RepositoryMonitor will notify its Observer classes when there is a change in the bundle repository, and will send them a RepositoryEvent object as a parameter.

class RepositoryEvent **extends** EventObject
{
    static final int ENTRY_ERROR = 0;
    static final int ENTRY_CREATE = 1;
    static final int ENTRY_MODIFY = 2;
    static final int ENTRY_DELETE = 3;

    RepositoryEvent(Object obj, Bundle bundle, String name);
    Bundle getBundle();
    String getName();
    int getType();
}

Figure 4.4 – RepositoryEvent class

From this class, the user can get the bundle which JAR file has been changed, name of the JAR file and type of event. RepositoryEvent class also offers constant values for the different types of events that RepositoryMonitor can rise.

The observation of the directory is implemented by using WatchService, from java.nio.file package. A watch service monitors objects like folders for changes and events [23]. This service is not an OSGi service and does not have to be confused with one of them.

RepositoryMonitor extends from EventMonitor, so it will support the filter system that will be explained in section 4.3.

4.2.2 State monitoring

This feature is performed by using one class within the event package: StateMonitor. In this case, the framework uses BundleEvent from the OSGi framework as its correspondent event class. BundleEvent will monitor the OSGi BundleContext and will raise an event if there is any change in the state of any bundle. BundleContext is an OSGi class which objective is to provide interaction with the OSGi framework [9]. Therefore, StateMonitor will raise an event if any bundle is installed, started, stopped and so on. This feature corresponds with requirement 3.2.2 from chapter 3.

class StateMonitor **extends** EventMonitor **implements** BundleListener
{
    StateMonitor(BundleContext bundleContext);
    void typeFilterSetDefaultConfiguration(TypeFilterSet tf);
    void bundleChanged(BundleEvent be);
}

Figure 4.5 – StateMonitor class
StateMonitor extends from EventMonitor, so it is an Observable class. StateMonitor will notify its Observer classes when there is a change in the state of any bundle, and will send them an OSGi BundleEvent object as a parameter.

class BundleEvent extends EventObject
{
    static final int INSTALLED = 1;
    static final int LAZY_ACTIVATION = 512;
    static final int RESOLVED = 32;
    static final int STARTED = 2;
    static final int STARTING = 128;
    static final int STOPPED = 4;
    static final int STOPPING = 256;
    static final int UNINSTALLED = 16;
    static final int UNRESOLVED = 64;
    static final int UPDATED = 8;

    BundleEvent(int type, Bundle bundle);
    BundleEvent(int type, Bundle bundle, Bundle origin);
    Bundle getBundle();
    Bundle getOrigin();
    int getType();
}

Figure 4.6 – BundleEvent class

From this class, the user will be able to get the bundle and type of event. BundleEvent class also offers constant values for the various types of events.

This feature implementation uses the BundleListener class of the OSGi framework to perform the monitoring process. This class monitors the BundleContext and notifies its Observer classes if there is any change of state in any bundle [9].

Although BundleMonitor does not add any functionality to the OSGi BundleListener implementation, it extends from EventMonitor, so it will support the filter system that will be explained in section 4.3.

4.2.3 Manifest monitoring
This feature is performed by using two classes within the event package: ManifestMonitor and ManifestEvent. ManifestMonitor will monitor the installed bundles manifest and will raise an event if any entry of the manifest of any updated bundle is created, modified or deleted. This feature corresponds with requirement 3.2.4 from chapter 3.

class ManifestMonitor extends EventMonitor implements Observer
{
    ManifestMonitor(BundleContext bundleContext);
    void typeFilterSetDefaultConfiguration(TypeFilterSet tf);
    public void update(Observable obs, Object obj);
}

Figure 4.7 – ManifestMonitor class

ManifestMonitor extends from EventMonitor, so it is an Observable class. ManifestMonitor will notify its Observer classes when there is any change in any bundle manifest and will send them a ManifestEvent object as a parameter.
class ManifestEvent extends EventObject
{
    static final int CREATED = 1;
    static final int MODIFIED = 2;
    static final int DELETED = 3;

    ManifestEvent(Object source, Bundle bundle, String header, int type);
    Bundle getBundle();
    String getHeader();
    int getType();
}

Figure 4.8 – ManifestEvent class

From this class, the user will be able to get the bundle, the name of the modified header and type of event. ManifestEvent also offers constant values for each type of event.

ManifestMonitor implementation uses the StateMonitor class of subsection 4.2.2 for detecting if any bundle is installed, updated or uninstalled. ManifestMonitor stores all manifest data. If any bundle is installed, it will add its manifest information. If it is updated, it will compare the new manifest with the old one. If there is any change it will generate a ManifestEvent and substitute the old manifest with the new one.

ManifestMonitor extends from EventMonitor, so it will support the filter system that will be explained in section 4.3.

4.2.4 Data field monitoring
This feature is performed by two classes within the event package: DataFieldMonitor and DataFieldEvent. DataFieldMonitor will monitor different class attributes within a bundle and will raise an event if any previously specified primitive data field is modified. Modifications of a data field can be done with the =, ++ or -- Java operators. This feature corresponds with requirement 3.2.5 from chapter 3.

class DataFieldMonitor extends EventMonitor
{
    DataFieldMonitor(BundleContext bundleContext);
    boolean addDataField(String className, String fieldName);
    void registerWeavingService();
    void unregisterWeavingService();
    void update(Object field, String className, String fieldName);
}

Figure 4.9 – DataFieldMonitor class

DataFieldMonitor works in a different way as the rest of the monitor classes. To make it work, it is necessary to call the method registerWeavingService(). This method will register the necessary services for this monitor to work. We will talk about these services later. unregisterWeavingServices() will perform the opposite action. There is also another method, addDataField, which will specify a new class and field that will be monitored. DataFieldMonitor extends from EventMonitor, so it is an Observable class and will notify its Observer classes when a specified field is modified, and will send a DataFieldEvent object as a parameter.
class DataFieldEvent extends EventObject
{
    DataFieldEvent(Object obs,
                    Object field,
                    String className,
                    String fieldName);

    Object getField();
    String getClassName();
    String getFieldName();
}

Figure 4.10 – DataFiedlEvent class

From this class, the user will be able to get the modified field value, as well as the
name of the class and field. As it can be noticed, this class does not provide constant
values as the rest of the previous event classes. The reason for this is that there is only
one type of event in data field monitoring: modified.

This feature implementation presented more complications to meet requirement
3.4.2 than the rest of features. As it consists in monitoring classes data fields within
bundles. It was necessary to get private data fields within classes within bundles. The
solution to this is byte-code instrumentation (See subsection 2.6.3). This
implementation uses OSGi WeavingHook class. Bundles registering a WeavingHook
service can access any class before it is loaded by the OSGi framework [9]. Once it has
the class that is going to be loaded. DataFieldMonitor uses Javassist. Javassist is a Java
library that permits to modify byte-code of Java classes. This implementation therefore
combines WeavingHook and Javassist to modify new loaded classes. It must perform
two changes in the previously specified classes: get a service registered by
DataFieldMonitor that acts as an interface between DataFieldMonitor and the
instrumented bundle; the second change is to call a method within that service when a
specified data field is written. This method will call to DataFieldMonitor that will raise
the event, and will have the object value as a parameter. This implementation permits to
monitor data fields without changing the source code, and therefore respects
requirement 3.4.2.

As this type of monitoring does not focus on OSGi bundles and does not have
different types of events, these classes are not compatible with the filter system that will
be presented in section 4.3.

4.2.5 Service monitoring
This feature is performed by two classes within event package: ServiceRegistryMonitor
and ServiceEventMod. ServiceRegistryMonitor will monitor the OSGi service registry
and will raise an event if any service is registered, modified or unregistered. This
feature corresponds with requirement 3.2.6 from chapter 3.

class ServiceRegistryMonitor extends EventMonitor implements ServiceListener
{
    ServiceRegistryMonitor(BundleContext bundleContext);
    void typeFilterSetDefaultConfiguration(TypeFilterSet tf);
    void serviceChanged(ServiceEvent se);
}

Figure 4.11 – ServiceRegistryMonitor class

ServiceRegistryMonitor extends from EventMonitor, so it is an Observable class
and will notify its Observer classes when a there is any change in the service registry,
and will send a DataFieldEvent object as a parameter.
```java
class ServiceEventMod extends EventObject {
    static final int MODIFIED = ServiceEvent.MODIFIED;
    static final int MODIFIED_ENDMATCH = ServiceEvent.MODIFIED_ENDMATCH;
    static final int REGISTERED = ServiceEvent.REGISTERED;
    static final int UNREGISTERED = ServiceEvent.UNREGISTERING;

    ServiceEventMod(Object source, int type,
            ServiceReference<?> reference,
            Object service);
    int getType();
    ServiceReference<?> getServiceReference();
    Object getService();
}
```

Figure 4.12 – `ServiceEventMod` class

`ServiceEventMod` class is a modification of `ServiceEvent` class from the OSGi framework. From this class, the user can get the service reference (and the registering bundle from it) and the type of event [9]. In addition, `ServiceEventMod` allows retrieving the object registered as a service. `ServiceEventMod` also offers constant values for the different types of events as `ServiceEvent` does.

`ServiceRegistryMonitor` extends from `EventMonitor`, so it will support the filter system that will be explained in section 4.3.

### 4.3 event.filter package

This package contains all classes related to the monitoring process and was presented in section 4.1.

Section 4.3 is divided in two subsections that correspond with requirements 3.2.7 and 3.2.8. Each subsection has an explanation of the feature, the classes that are available for the user and the implementation of those classes.

#### 4.3.1 Bundle filtering

This feature is performed by two classes within the `event.filter` package: `IdFilterSet` and `NameFilterSet`. These classes filter events depending on the bundle that generated it. Both classes extend from `BundleFilterSet`. This abstract class does not add any functionality. But, it is used to group the bundle filter classes. These classes are meant to satisfy requirement 3.2.8.

```java
class IdFilterSet extends BundleFilterSet {
    IdFilterSet();
    void addEntry(long idBundle, Boolean mode);
    boolean isOpen(long idBundle);
}
```

Figure 4.13 – `IdFilterSet` class

The most important method that this class offers is `addEntry`. It will add or modify the mode (opened or closed) associated with an id bundle. An opened value will allow all events related to that id, while a closed one will block them. This class is implemented by a Java `Map` that stores ids of bundles and a correspondent open or closed value.
**class** NameFilterSet **extends** BundleFilterSet
{
    NameFilterSet();
    void addEntry(String nameBundle, Boolean mode);
    boolean isOpen(String nameBundle);
}

Figure 4.14 – *NameFilterSet* class

The most important method in this class is also *addEntry*. It has the same function as in *IdFilterSet* class. In this case, *NameFilterSet* stores strings with bundle names instead of ids.

As we can see in Figure 4.2, we can set one of the bundle filter class to an *EventMonitor* class (or any other class that extends from it). *EventMonitor* class will manage to use the filter in case of event for each one of them. If the bundle is opened in the filter and there is an event, it will allow it and will notify its *Observer* classes. If it is closed or it does not exist, it will not raise the event.

**4.3.2 Type filtering**

Type filtering is similar than bundle filter classes. The class that performs this functionality is *TypeFilterSet*. It can block or allow events depending on their type. This class is meant to satisfy requirement 3.2.7.

**class** TypeFilterSet
{
    TypeFilterSet();
    void addEntry(int type, boolean mode);
    void resetTypeFilter(boolean mode);
    void openTypeFilter();
    boolean isOpen(int type);
    int size();
    void clear();
}

Figure 4.15 – *TypeFilterSet* class

The most important method is *addEntry* again. It will add or modify the mode (opened or closed) associated with a type of event. An opened value will allow all events of that kind, while a closed one will block them. It is also meant to work with all monitor classes. Section 4.2 contains the different kind of events for each type of monitoring, providing constants for each one of them.

*TypeFilterSet* works, as the previous filtering classes, with an *EventMonitor* class (or any other class that extends from it). *EventMonitor* class will manage the type filter itself. If a new event has an opened type associated, it will raise the event. If not, it will block it.
5 Results

Chapter 5 presents the two demonstrator applications that use the framework. Their requirements are specified in section 3.5. It is meant to be a tutorial on how the framework works. It also shows that the framework meets its requirements. Demonstrator applications are two simple applications with a GUI. Their objective is to exemplify how the users can use the framework to develop their own monitoring systems.

Section 5.1 presents the first demonstrator. Section 5.2 explains the second demonstrator. Finally, section 5.3 presents a traceability matrix to show the relation between the requirements and implementation.

5.1 Demonstrator 1: bundles

Using this framework is going to need an initial configuration in OSGi. This framework has been tested in eclipse equinox 3.7.1 but it should work in other versions as well. Before using this framework, it is necessary to install javassist 3.10.0.GA and the framework itself. With these bundles installed in OSGi, we can start using the framework.

Demonstrator 1 focuses in all bundle monitoring features of the framework. It is going to meet requirements 3.5.1, 3.5.2, 3.5.3 and 3.5.6. So, in the first place it is going to provide a GUI as requirement 3.5.1 states.

![Figure 5.1 – Demonstrator 1 GUI](image)

Figure 5.1 shows demonstrator 1 GUI. Installed bundle table and Monitoring log list are going to show updated information about the changes of bundles (requirement 3.5.2). The buttons on the right are going to allow the user to perform changes on the bundles (requirement 3.5.3). Filters menu, Monitor and Unmonitor buttons permit the user to specify what events are going to be shown (requirement 3.5.6). Monitors menu allow changing what kind of monitoring is going to be performed: repository monitoring for requirement 3.2.2 function, state monitoring for requirement 3.2.2, manifest monitoring for requirement 3.2.4 and Data field monitoring for requirement 3.2.5. Instrumentation menu is related to data field monitoring. All these features are going to be explained in the following subsections.

Now that the GUI is explained, let us continue with how it is implemented. To simplify, we are going to see only the part of the program that interacts directly with the
framework, showing only the most interesting piece of code for each feature. In demonstrator 1, the interaction with the framework is made in a class named Monitor.

```java
public class Monitor extends Observable implements Observer {
    public void update(Observable obs, Object obj) {
        setChanged();
        notifyObservers(obj);
    }
}
```

Figure 5.2 – Monitor class

This class extends from Observable to notify the view of changes for the bundle list and the log. It implements Observer because all monitor classes in the framework are Observable classes too. The functionality for this class is going to be added step by step in the next chapters.

5.1.1 Repository monitoring

Repository monitoring feature, as we saw in section 4.2.1, will monitor changes in the bundle repository for installed bundle JAR files. That means that if a JAR file is created, modified or deleted, it will show a message in the log. First of all, let us take a look at the code added to the Monitor class:

```java
public class Monitor extends Observable implements Observer {
    ArrayList<EventMonitor> bundleMonitors;

    public void Monitor(BundleContext bundleContext) {
        bundleMonitors = new ArrayList<EventMonitor>();
        bundleMonitors.add(new RepositoryMonitor(bundleContext));
    }

    public void setMonitorOn(int index) {
        bundleMonitors.get(index).addObserver(this);
    }

    public void setMonitorOff(int index) {
        bundleMonitors.get(index).deleteObservers();
    }

    @Override
    public void update(Observable obs, Object obj) {
        setChanged();
        notifyObservers(obj);
    }
}
```

Figure 5.3 – Monitor class with repository monitoring
As we can see in the previous figure, we are going to have an array with all monitor classes. We can group them because they all extend from `EventMonitor` class. The first class added to the array is `RepositoryMonitor`. `setMonitorOn` and `setMonitorOff` methods just activate or deactivate the monitoring of a class within the array. This is done by adding or deleting the `Monitor` class to the specified `EventMonitor` class. If there is any change in the repository, the `update` method will notify the `Observer` (The view, in this case). The object that is sent to the `Observer` will be an object of `RepositoryEvent` class. It will contain all necessary information about the event as we can see in subsection 4.2.1.

![Figure 5.4 – Repository monitoring in demonstrator 1](image)

In figure 5.4 we can see how repository monitor shows its messages in the GUI. This is the output after installing a bundle, moving its JAR file outside the repository, copying it back and rewriting it again.

### 5.1.2 State monitoring
State monitoring is activated by default in demonstrator 1, as we could see in Figure 5.4. This feature will raise an event if any bundle state is changed, as it is explained in section 4.2.2. That means that if any bundle is installed, uninstalled, started, stopped or updated, it will show a message in the log and it will change its state in the installed bundles table. Let us start with the code that we have to add in `Monitor` class constructor:

```java
bundleMonitors.add(new StateMonitor(bundleContext));
```

![Figure 5.5 – State monitoring initialization](image)

We have just added a `StateMonitor` object to the array. The activation and deactivation of the monitor will work with `setMonitorOn` and `setMonitorOff` methods too. These methods will require an `index` parameter. This parameter corresponds with the `EventMonitor` class position within the `bundleMonitor ArrayList`. On the other hand, the `update` method will raise a `BundleEvent` object in case of event with all the relevant data. We can see the methods that it provides in section 4.2.2.
Figure 5.6 shows how state monitoring displays its messages in demonstrator 1. This is the output after performing some operations such as Install, Start, Stop and Update a bundle.

5.1.3 Manifest monitoring

Manifest monitoring will raise an event if any updated bundle presents any modification in its manifest. That means if any entry is created, deleted or modified. As in the previous case, the additions in the Monitor constructor are fairly simple:

```java
bundleMonitors.add(new ManifestMonitor(bundleContext));
```

Figure 5.7 – Manifest monitor initialization

The update method will raise a ManifestEvent object in this case. We can see the information that we can get form it in section 4.2.3.

In figure 5.8 we can see how manifest monitor shows its updates in demonstrator 1. This is the output after installing a bundle, changing its manifest and updating it. The changes in the manifest were made so it would show the three possible types of changes. These are the two versions of the manifest that we have used for this example:
As we can see, the created-by entry has been added, the import-package entry modified and the export-package entry deleted. This corresponds with the changes shown by the log list in figure 5.8.

5.1.4 Data field monitoring

Data filed monitoring will raise an event if a previously specified data field is changed. That means if a primitive data field is written with the =, ++ or -- operators. As we saw in subsection 4.2.4, this feature is different of the rest of monitoring features, as it monitors a data field instead of an entire bundle. These are the changes in the Monitor constructor:

```java
bundleMonitors.add(new DataFieldMonitor(bundleContext));
((DataFieldMonitor)bundleMonitors.get(3)).registerWeavingService();
```

Figure 5.11 – Data field initialization and services registration

Below the initialization of DataFieldMonitor it is the registerWeavingService() method. This method will register the two services that data field monitoring needs to work (see subsection 4.2.4). Also, we are going to add a new method in Monitor:

```java
public void unregisterServices() {
    ((DataFieldMonitor)bundleMonitors.get(3)).unregisterWeavingService();
}
```

Figure 5.12 – Data field services unregistration

This method will be used to unregister the services after performing data field monitoring. In demonstrator 1, this method is called when the user stops data field monitoring in the Monitors menu.

In this case, the update method will raise a DataFieldEvent object in case of event with all the necessary data. We can see the methods that it provides in section 4.2.4.
Figure 5.13 shows how data field monitor deploys its updates in demonstrator 1. This is the output after instrumenting a field named `humidity` within `DataWeather` class. Then, we have made some changes in its bundle execution. These operations have changed the value of the data field as Figure 5.13 shows. The user only has to introduce the class and field name using the Instrumentation menu. The bundle must be updated after that. The rest of the process is hidden from the user.

5.1.5 Id filtering
Id filter set objective is to block or allow events from `EventMonitor` classes depending on the bundle that trigged them. We can use the same id filter for different monitors. We can specify rules for several bundles in the same id filter.

```java
public class Monitor extends Observable implements Observer {
    private ArrayList<EventMonitor> bundleMonitors;
    private IdFilterSet idFilter;

    public Monitor(BundleContext bundleContext) {
        bundleMonitors = new ArrayList<EventMonitor>();
        idFilter = new IdFilterSet();

        bundleMonitors.add(new RepositoryMonitor(bundleContext));
        bundleMonitors.add(new StateMonitor(bundleContext));
        bundleMonitors.add(new ManifestMonitor(bundleContext));
        bundleMonitors.add(new DataFieldMonitor(bundleContext));

        for (int i = 0; i < bundleMonitors.size() - 1; i++) {
            bundleMonitors.get(i).setBundleFilterSet(idFilter);
        }

        ((DataFieldMonitor)bundleMonitors.get(3)).registerWeavingService();
        setMonitorOn(1);
    }
```
private void setId(long idBundle, boolean mode)
{
    idFilter.addEntry(idBundle, mode);
}

public void setIdOpened(long idBundle)
{
    setId(idBundle, true);
}

public void setIdClosed(long idBundle)
{
    setId(idBundle, false);
}

Figure 5.14 – Monitor class with id filter

Figure 5.14 only shows the new data field, the complete constructor and the new methods for simplifying reasons. As we can see, we have set the filter to all monitor classes in the constructor but DataFieldMonitor, as it monitors classes, not bundles. setIdOpened and setIdClosed methods add or modify rules in the id filter. We can see the implementation of IdFilterSet in section 4.3.1 and the implementation of EventMonitor in section 4.1, in which is implemented the interaction between filters and monitor classes.

Figure 5.15 – Id filtering in demonstrator 1

In figure 5.15 we can see how id filter works. The id filtering can be performed by using the installed bundles list and the Monitor or Unmonitor buttons. This is the output when we install two bundles, select the second one and press the Unmonitor button. Then, we have performed the same operations for both of them, with repository and state monitoring activated.

5.1.6 Type filtering
Type filtering is similar than id filtering. In this case, it blocks or allows events depending on the type. We are going to use the same filter class for every type of monitoring. But, we are going to specify the instructions for different types of monitoring in different filter objects.
public class Monitor extends Observable implements Observer {

    private ArrayList<TypeFilterSet> typeFilters;

    public Monitor(BundleContext bundleContext) {
        bundleMonitors = new ArrayList<EventMonitor>();
        typeFilters = new ArrayList<TypeFilterSet>();
        idFilter = new IdFilterSet();
        bundleMonitors.add(new RepositoryMonitor(bundleContext));
        bundleMonitors.add(new StateMonitor(bundleContext));
        bundleMonitors.add(new ManifestMonitor(bundleContext));
        bundleMonitors.add(new DataFieldMonitor(bundleContext));

        for (int i = 0; i < bundleMonitors.size() - 1; i++) {
            typeFilters.add(new TypeFilterSet());
            bundleMonitors.get(i).setTypeFilterSet(typeFilters.get(i));
            bundleMonitors.get(i).setBundleFilterSet(idFilter);
        }

        ((DataFieldMonitor) bundleMonitors.get(3)).registerWeavingService();
        setMonitorOn(1);
    }

    private void setType(TypeFilterSet typeFilterSet, int type, Boolean open) {
        typeFilterSet.addEntry(type, open);
    }

    public void setTypeOpened(int index, int type) {
        setType(typeFilters.get(index), type, true);
    }

    public void setTypeClosed(int index, int type) {
        setType(typeFilters.get(index), type, false);
    }

    
    Figure 5.16 – Monitor class with type filter

    The transformation is similar than the id filter one. We have set in the constructor a type filter object for each monitor class. setTypeOpened and setTypeClosed have the same function as setIdOpened and setIdClosed. We can see the implementation of TypeFilterSet in section 4.3.2 and the implementation of EventMonitor in section 4.1, in which is implemented the interaction between filters and monitor classes.

    33
In figure 5.17 we can see how type filter works. Type filter is performed by using the installed bundles list and the Filters menu. This is the output when we install a bundle, deactivate the stopped events and start and stop the bundle. The stopped message does not appear in the log while the started does.

5.2 Demonstrator 2: services
For this demonstrator, we must use the same initial configuration as in demonstrator 1. This configuration is explained in the beginning of section 5.1.

Demonstrator 2 focuses in all bundle monitoring features of the framework. It is going to meet requirements 3.5.1, 3.5.4, 3.5.5 and 3.5.6. So, it has to a GUI as requirement 3.5.1 states.

The GUI is similar than demonstrator 1 one. But there are a few differences. There is a new table for displaying services knowledge named Services available (requirement 3.5.4). Also, the Monitor menu now has just two types of monitoring: state monitoring for requirement 3.2.2 for displaying the information about Installed bundles and service
monitor for requirement 3.2.6. Filters menu is also changed so it shows the new types of events. In this case, we are just going to explain the new service feature, as the rest of functionality (state monitoring, id and type filters) is explained in section 5.1.

The implementation is also similar, with a Monitor class. With the bundle state monitor and the filters already implemented, it is going to look like this:

```java
public class Monitor extends Observable implements Observer {
    private ArrayList<EventMonitor> bundleMonitors;
    private ArrayList<TypeFilterSet> typeFilters;
    private IdFilterSet idFilter;

    public Monitor(BundleContext bundleContext) {
        bundleMonitors = new ArrayList<EventMonitor>();
        typeFilters = new ArrayList<TypeFilterSet>();
        idFilter = new IdFilterSet();
        bundleMonitors.add(new StateMonitor(bundleContext));
        for (int i = 0; i < bundleMonitors.size() - 1; i++) {
            typeFilters.add(new TypeFilterSet());
            bundleMonitors.get(i).setTypeFilterSet(typeFilters.get(i));
            bundleMonitors.get(i).setBundleFilterSet(idFilter);
        }
        setMonitorOn(0);
    }

    public void setMonitorOn(int index) {
        bundleMonitors.get(index).addObserver(this);
    }

    public void setMonitorOff(int index) {
        bundleMonitors.get(index).deleteObservers();
    }

    private void setType(TypeFilterSet typeFilterSet, int type, Boolean open) {
        typeFilterSet.addEntry(type, open);
    }

    public void setTypeOpened(int index, int type) {
        setType(typeFilters.get(index), type, true);
    }

    public void setTypeClosed(int index, int type) {
        setType(typeFilters.get(index), type, false);
    }
}
```
private void setId(long idBundle, boolean mode) {
    idFilter.addEntry(idBundle, mode);
}

public void setIdOpened(long idBundle) {
    setId(idBundle, true);
}

public void setIdClosed(long idBundle) {
    setId(idBundle, false);
}

@Override
public void update(Observable obs, Object obj) {
    setChanged();
    notifyObservers(obj);
}

Figure 5.19 – Monitor class with bundle state monitoring and filters

5.2.2 Service monitoring
Service monitor will raise an event if there is any change in the service repository. That is, if any service is registered, modified or unregistered. To simplify, we are going to just show the constructor of Monitor. As it will be the only method changed:

public class Monitor extends Observable implements Observer {

    public Monitor(BundleContext bundleContext) {
        bundleMonitors = new ArrayList<EventMonitor>();
        typeFilters = new ArrayList<TypeFilterSet>();
        idFilter = new IdFilterSet();
        bundleMonitors.add(new StateMonitor(bundleContext));
        bundleMonitors.add(new ServiceRegistryMonitor(bundleContext));

        for(int i = 0; i < bundleMonitors.size() - 1; i++) {
            typeFilters.add(new TypeFilterSet());
            bundleMonitors.get(i).setTypeFilterSet(typeFilters.get(i));
            bundleMonitors.get(i).setBundleFilterSet(idFilter);
        }
        setMonitorOn(0);
    }

    Figure 5.20 – Monitor constructor with service monitoring
As we can see, we have just added a `ServiceRegistryMonitor` object to the monitor array. The update method will raise a `ServiceEventMod` object in case of event with all the necessary data. We can see the methods that it provides and how `ServiceRegistryMonitor` is implemented in section 4.2.5.

![Figure 5.21 – Service monitoring in demonstrator 2](image)

In figure 5.21 we can see how service monitor shows its messages in the GUI, and change the service table. This is the output when we install two different versions of the same bundle. This bundle offers a service named `Soil`. As we can see, the two services appear, but with different versions.

### 5.3 Traceability matrix
Now that we have tested our framework, we can make a traceability matrix. With this table we are going to relate the requirements from chapter 3 with the different classes from chapter 4 and demonstrator applications from chapter 5.
<table>
<thead>
<tr>
<th>Req</th>
<th>Classes</th>
<th>Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EM</td>
<td>RM</td>
</tr>
<tr>
<td>3.2.2</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3.2.2</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>3.2.4</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>3.2.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3.1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3.3.2</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3.4.1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3.4.2</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3.4.3</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>3.5.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6.1</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 5.1 – Traceability matrix

The Req rows contain all requirements from chapter 3. The Classes columns contain all classes from the software interface of the framework from chapter 4. Legend:
- EM: EventMonitor
- RM: RepositoryMonitor
- RE: RepositoryEvent
- SM: StateMonitor
- SE: BundleEvent
- MM: ManifestMonitor
- ME: ManifestEvent
- DM: DataFieldMonitor
- DE: DataFieldEvent
- SM: ServiceRegistryMonitor
- SE: ServiceEventMod
- TF: TypeFilterSet
- BF: BundleFilter (and its subclasses)

The Apps columns contain the demonstrator applications from chapter 5:
- D1: Demonstrator 1
- D2: Demonstrator 2

Each requirement (Req row) will be checked with an ‘x’ if it is implemented in the correspondent class (Classes column) or application (Apps column). Each class or application will be checked with an ‘x’ if its implementation features the correspondent requirement.
6 Conclusions

Chapter 6 contains the conclusions about the project. It comprises achievements, failures and future work.

Section 6.1 presents the summary. Section 6.2 explains further work.

6.1 Summary

Summing up, we have implemented a framework to support development of monitoring systems, in the context of self-adaptive systems. First, we have made a survey state of the art about monitoring systems. This survey reviews 16 articles. These articles have helped us to extract some common features concerning monitoring of OSGi applications. After that we have specified the requirements for the framework. The requirements have been mainly proposed by the stakeholders of the project. The requirements specify a component and behavior monitoring system, using an event based approach, following the classification extracted from the survey results. After that, we have implemented the framework, according to the functional requirements and following the specified quality attributes. To meet these requirements, it has been necessary to take different approaches. Most of monitoring features were implemented by interacting with the OSGi framework, as in bundle or service monitoring. However, some of them took other solutions such as WatchService for manifest monitoring, or byte-code manipulation for data field monitoring. Then, we have implemented two demonstrator applications that use the framework. Source code for both framework and demonstrators is available on GitHub [24].

One of the achievements of the implementation part has been to meet the quality attributes. These attributes were presented in the requirements. They were three desirable properties: create an efficient, extensible and decoupled from the base-level framework. Besides, all mandatory functional requirements have been completed and the filter system offers a flexible solution for the different features that can perform.

However, there is one notable failure, as optional requirement 3.2.6 has not been implemented. This requirement consisted on monitor exchange of messages between two OSGi components using a communication framework like ECF. This could be one subject for further work.

6.2 Further work

In addition, there are some other improvements that could be done. The filter system could be expanded to offer more functionality. For example, a filter associated with bundles of an application within the OSGi platform, or all bundles within a Java package. Also, the survey part is opened, and more works could be added. Finally, more monitoring features could be added to the framework. As we have seen in the survey, there is unlimited data that can be extracted from the base-level system. That means that there is always the possibility to expand the features inside the component and behavior monitoring, and event outside these types of monitoring.

These future features could be done according with an evaluation of the framework, and the needs of the users.
References

[3] OSGi Alliance; http://www.osgi.org; Last date of use: 9-8-2013
[9] OSGi core release 4; http://www.osgi.org/javadoc/r4v43/core/; Last date of use: 9-8-2013
[23] Observer; http://docs.oracle.com/javase/7/docs/api/overview-summary.html; Last date of use: 9-8-2013