Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

An OSGi Implementation for Autonomous Goal-Oriented Deployment

João Paulo C. de Araujo

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof.ª Dr.ª Genaína Nunes Rodrigues

Brasília
2017

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Rodrigo Bonifácio de Almeida

Banca examinadora composta por:

Prof.ª Dr.ª Genaína Nunes Rodrigues (Orientador) — CIC/UnB
Prof. Dr. George Luiz Medeiros Teodoro — CIC/UnB
Prof. Dr. Felipe Pontes Guimarães — Agência Espacial Brasileira

CIP — Catalogação Internacional na Publicação

de Araujo, João Paulo C..

An OSGi Implementation for Autonomous Goal-Oriented Deployment / João Paulo C. de Araujo. Brasília : UnB, 2017.
55 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2017.

1. Dependabilidade, 2. Engenharia de Requisitos Orientada a Objetivos, 3. Modelagem de Contexto, 4. Recursos Heterogêneos, 5. Planejamento de Deployment, 6. OSGi

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

An OSGi Implementation for Autonomous Goal-Oriented Deployment

João Paulo C. de Araujo

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.ª Dr.ª Genaína Nunes Rodrigues (Orientador)
CIC/UnB

Prof. Dr. George Luiz Medeiros Teodoro
CIC/UnB

Prof. Dr. Felipe Pontes Guimarães
Agência Espacial Brasileira

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordenador do Bacharelado em Ciência da Computação

Brasília, 08 de Dezembro de 2017

Agradecimentos

Thanks to God, and my savior Jesus Christ, for giving me a purpose and meaning in life. What is the point of searching for academic excellence? Or at least of trying to develop a good final project? In five hundred of years, no one will ever remember of the young kid from Brasilia who tried to do his best, pulling dozens of all-nighters and leaving his social life behind, neither of his work, for it will be totally outdated. I ask again, what is the meaning of all of it? I found that meaning in something greater than me. By looking at life through this materialistic lenses, I saw myself in complete desperation, lacking for something that nothing in this world would never satisfy, since any satisfaction I'd find was only temporary. I've found rest only in the words of a Jew who said: "Everyone who drinks this water will be thirsty again, but whoever drinks the water I give them will never thirst." (John 4:13,14) He said that only him could provide me with everything I was searching for, namely, a purpose, and he provided it in the most unimaginable way: by excelling in all areas of his life here on earth and exchanging his perfect resume for my failed one in the cross, through sheer grace! He was everything I cannot be and achieved everything I cannot achieve. Now, I don't need to live by seeking for pleasures or finding significance through my performance, for I already have the ultimate pleasure in him and his perfect curriculum was graciously given to me. My life now is dedicated to follow him and to grow more likely to his character. If there is any aspect of this work that is good in any form, it is because of his kindness in providing me with wisdom and knowledge.

I also want to thank my girlfriend Luiza, for being the one person who, even though did not know a thing about programming, patiently and diligently listened to detailed descriptions of the problems I've faced during the execution of this project. Thanks to Genaína, my professor adviser, who was attentive and available, answering my emails almost in real time, always receiving me with a smile in her face. Thanks to Gabriel Rodrigues, author of the GoalD methodology, for being solicit and altruistically helping me with this project. Thanks also to my family for all the support, and to my friends Marcello Doudement, Érica Rios, Arnaldo Junior and Érica Lima who helped me keep my sanity during the college years.

Resumo

Com a expansão da tecnologia de Internet das coisas, novos desafios computacionais têm surgido. Estes possuem como característica principal seu alto grau de heterogeneidade de recursos, uma vez que são compostos pelos mais variados dispositivos, os quais se utilizam de uma infraestrutura de orientação a serviços para publicarem e descobrirem funcionalidades por meio de serviços. Tendo em vista a natureza complexa de tais sistemas, torna-se necessário o uso de ambientes de gerenciamento de deployment desses recursos heterogêneos. Dentre eles, um potencial framework é o padrão OSGi, que se caracteriza por ser um framework Java para desenvolvimento e deployment de programas modulares (em bundles). Nesse trabalho, será abordada a integração do OSGi ao GoalD, uma plataforma para deployment de recursos heterogêneos conforme a abordagem orientada a objetivos, por meio da descrição detalhada da implementação de cada uma das atividades do processo de deployment autônomo, definida pelo GoalD, utilizando os conceitos e técnicas apresentados pela tecnologia OSGi.

Palavras-chave: Dependabilidade, Engenharia de Requisitos Orientada a Objetivos, Modelagem de Contexto, Recursos Heterogêneos, Planejamento de Deployment, OSGi

Abstract

With the expansion of the Internet of Things technology, new computational challenges have risen. Their main characteristic is the high degree of resource heterogeneity, once they are composed by the most variant kinds of devices, which make use of a service-oriented infrastructure to publish and discover functionalities through services. Seeing the complex nature of such systems, it is necessary the use of deployment management environments to handle such heterogeneous resources. Amongst them, a potential framework is the OSGi standard, which is known for being a Java framework for the development and deployment of modular applications (bundles). In this work, it will be addressed the integration of OSGi to GoalD, a platform for the deployment of heterogeneous resources that follows the goal-oriented approach, through the detailed description of the implementation of each activity of the autonomous deployment process, defined by GoalD, by using the concepts and techniques presented by the OSGi technology.

Keywords: Dependability, Goal-Oriented Requirements Engineering, Context Modelling, Heterogenous Resources, Deployment Planning, OSGi

Sumário

Lista de Figuras

Capítulo 1

Introduction

Thanks to the advance of technology, the second decade of the twentieth first century started with previsions for the future never imagined before. Computers, which have gotten smaller and cheaper, fitting in purses and pockets, now are being embedded into coffee machines, dish washers, street lamps, and any other device based on electricity that has an on/off switch. The advent of technologies such as Internet of Things [9], Ubiquitous Computing [11] and Opportunistic Computing [25] allowed for these devices to interact with each other, since, now, each one of them can be connected do the Internet and are able to form computational environments, providing its different resources as input for new possible solutions.

While this creates unimaginable possibilities for the development of new solutions that may ease the lives of millions of people around the globe, it also brings a few challenges. One of them is the difficulty found in developing softwares able to harvest the capability of these diverse environments. Such computing environments are defined as highly heterogeneous and are "formed by different sets of devices, with different resources, and which are only partially known at design-time"[24].

The challenge, thus, consists of analyzing the environment in run-time, in order to gather the available artifacts required for a given system to run properly. This involves an efficient deployment planning process for providing the needed artifacts, and the correct deployment of such artifacts, which is the process of getting these software artifacts ready to be used.

1.1 Problem Definition

After analyzing the current deployment methods in literature and finding none that correctly suits the described needs in an autonomous manner and without a previous knowledge of the environment, Rodrigues [24] describes a methodology intended to solve this challenge, named GoalD. His approach tackles three main obstacles to the deployment in such environments, which are: heterogeneity, uncertainty at design time and autonomous deployment. Initially, the heterogeneity is related to the broad range of resources

that can be found. Next, the uncertainty at design time considers that different configurations may emerge during the execution. Finally, the autonomous deployment specifies that the system must be planned and deployed without the assistance of a human.

Following this track, the GoalD approach revolves around getting to the bottom of these three obstacles by relying on Goal-oriented Requirements Engineering for modeling user's desires and providing a suitable configuration, for a given a set of available artifacts. This is done by performing a set of activities that can be divided into two stages: offline and online. The offline activities are: goal modeling, mapping goals to components and artifact packaging. The online activities consist of automated deployment planning and deployment execution.

Nevertheless, Rodrigues' project gives special attention to the autonomous deployment planning, providing details about the generation of the artifacts and their autonomous selection for meeting a specific user goal through a series of activities, leaving the process of getting the artifacts ready to run to be deeper explained in a future stage of his work. Consequently, he aims to verify GoalD's potential to the real world, thus, offering a proof of concept of his methodology. In the end, he asserts that his proposed deployment planning algorithm is reliable, efficient and scalable, without providing the assessment with an actual implementation integrated with a real framework for fetching and binding the artifacts. Therefore, his solution lacks a concrete manner of certifying the feasibility of GoalD's approach.

By the end of the description of the GoalD's activities, he indicates the OSGi technology as a viable solution for the integration with GoalD, since it allows for the dynamic fetch and bind of components in run-time. Having said that, the following research agenda remains: "Is OSGi an adequate solution for a concrete implementation of the GoalD's approach?"

1.2 Proposed Solution

The present work reasons about the feasibility of OSGi as a potential solution for a GoalD implementation. By focusing on providing GoalD with a fully functional implementation of its approach, this project integrates all of the online and offline activities with the OSGi approach. This is done by immersing in the OSGi technology concepts in order to bring light to how OSGi can be adapted in order to accommodate the steps performed in the GoalD methodology. This is accomplished through the mapping of the GoalD's terms to OSGi concepts in a twofold way: a description of each GoalD offline activity by using the technology offered by OSGi and the implementation of a framework for the online phase, which is responsible for acquiring the artifacts and binding them autonomously at runtime.

Firstly, GoalD's offline activities are mapped into OSGi elements. The first offline activity, goal modeling, is performed at an early stage of a system's development. The second offline activity considers OSGi services as goals and Java classes as components in order to conceptually match the methodology with the implementation. The third offline

activity packages GoalD's artifacts as OSGi bundles, and stores the results in OSGi Bundle repositories.

Lastly, an OSGi Framework was implemented for a complete integration of the online activities of fetching and binding bundles in an autonomous manner. The first online activity shows the integration of the Deployment Planning Algorithm developed by Rodrigues with the OSGi Framework, as to provide a deployment plan given a set of context resources. This activity has two main processes: the new goal registration and the re-planning process. The last activity, the Deployment Execution, explains how the processes of fetching and binding of OSGi bundles are performed in an highly heterogeneous environment using our developed framework.

1.3 Structure

This work is organized as follows: Chapter 2 gives an introduction on the chief concepts of the theoretical background that outlines this work. Chapter 3 describes the Filling Station Advisor, the motivating example used for illustrating the ideas presented. Next, Chapter 4 provides the explanation of the solution in a thorough way, starting with an description of each GoalD activity realized by the OSGi technology, followed by a detailed explanation of each process done by the Framework implemented. Afterwards, in Chapter 5, the solution is evaluated considering performance and efficiency. Lastly, Chapter 6 lays out the conclusions of the work and proposes future projects.

Capítulo 2

Background

This chapter will present the main concepts related to Goal-Oriented Requirements Engineering, the GoalD approach, the OSGi framework and the Maven project. All of those are key to a better understanding of the solution proposed by this project.

2.1 Goal-Oriented Requirements Engineering (GORE)

A goal is defined by Van Lamsweerde as an "objective the system should achieve through cooperation of agents in the software-to-be and in the environment,"[18] and has a significant importance to the requirements engineering area due to the aid it provides in regard to solving human agents viewpoint conflicts and to the comprehension of the existence and the responsibilities of the requirements components[14], which allows the linkage of low-level details to high-level concerns.

The Goal-Oriented Requirements Engineering approach (GORE) was developed to take advantage of these benefits. It alludes to the usage of goals for every activity of the Requirements Engineering area and aims to acquire and represent systems and software requirements at the intentional level [28]. Modeling requirements as goals, or Goal Modeling, is the technique of representation of those systems that provides a powerful way of understanding the needs of the stakeholders besides figuring out the motives behind the development of a piece of software. In the end, a goal model revolves around laying out user goals and ways to meet them [1]. The GORE approach has a major impact in the development of systems, specially if they must adapt to different situations like environment changes, system capability changes and changes in the problem to be solved [15].

A goal model usually utilizes a directed graph tree to delineate the goals in a top-down manner so that goals can be successively refined via AND/OR decompositions and ultimately satisfied by the leaf nodes, which represent tasks that must be performed by actors. On the one hand, AND-refinements tie in a goal with a group of subgoals, and the fulfillment all of the subgoals is the only sufficient condition in regard to satisfying the goal. On the other hand, the OR-refinement relates a goal to a set of alternative

subgoals, meaning that meeting one of the subgoals is enough for satisfying the goal. The refinement comes to an end when each subgoal is utterly achievable by some particular agent capable of monitoring and controlling it [28]. Figure 2.1 depicts both AND- and OR-Refinements.

Amongst these methodologies, TROPOS [12] holds particular promise. In its heart, there is his modeling language, which is the base for constructing conceptual goal models. It rests in the concepts of actor, goal, plan, resource, dependency, capability and belief. An Actor is an entity that possess intentionality and represents agents, roles or positions. An actor can have none to multiple goals. A Goal is related to the interests of a particular actor, and is divided in softgoals and hard goals, meaning goals whose definition and satisfiability criteria are unclear, for the former, or trenchant, for the latter. A Plan can be defined as "a way of doing something"[12], and as a declaration of which refinement paths should be taken in the directed tree in order to achieve the main goal. A Resource corresponds to an entity with physical or informational attributes. Dependencies are subordination relationships between two actors which point out that one depends on the other, the depender and the dependee, severally. The Capability express an actor's competency to define, choose and execute a plan in order to satisfy a goal. Finally, a Belief declares the world's knowledge of an actor.

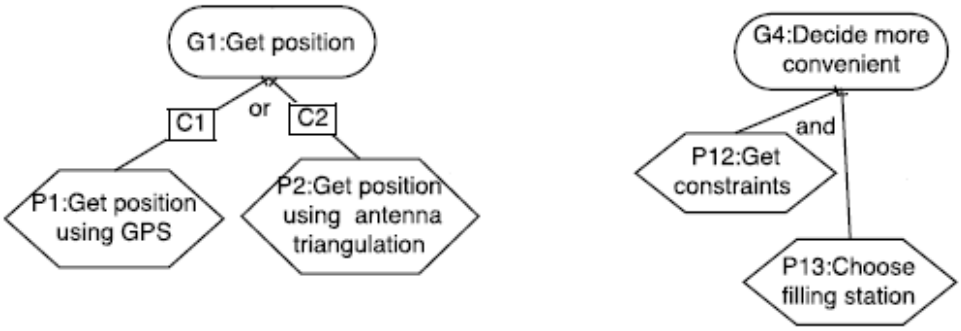


Figura 2.1: AND/OR-Refinements [24]

Moreover, Ali et al. [1] defined the concept of Context, in order to take into account the variability that may be found in a setting. A Context is "a partial state of the world that is relevant to an actor's goals,"or a set of properties and variants that an actor is compelled to watch in order to make a decision. Since variants were only allowed by regular Goal Models, but not clear-cut defined, an extension of TROPOS, named Contextual Goal Model, was created to meet the notions of context and goals by specifying explicitly which changes in the environment can be adopted and when. This allows for systematic derivation of variants based on the several different contexts that may appear [1]. Figure 2.1 describes *C1* and *C2* as contexts of the G1 OR-refinement of the Filling Station Advisor Application, explained in Chapter 3, as an example of how contexts are depicted in Contextual Goal Models.

A natural question that comes to mind is how to build an architecture that upholds the presented goal modeling. This must be done without losing its benefits of adaptation in environments that are highly heterogeneous and susceptible to run-

time changes, as well as all its variants and different contexts. Lamsweerde [28] suggests that the derivation process should be systematic, incremental, meet functional and non-functional requirements and make room for different architectural approaches to be foregrounded.

Yu et al. [29] corroborates Lamsweerde's idea by proposing a methodology that systematically preserves the variability property into the architecture according to a goal model. This is achieved by a component-based design that utilizes interface-binding for accomplishing goals through the arrangement of components, the called Component-connector view. This view assigns to each goal a component and an interface type, a collection of message signatures that lays out the bindings between the components and provides them a way of interacting with the environment. Two types of interface are defined by him: a *provides* interface, which delineates how a functionality of the component can be accessed, and a *requires* interface, that specify to which *provides* interface the component must be bound to. AND-refinement goals are characterized for having several *requires* interfaces as subgoals, while OR-refined have only one, indicating that any of the subgoals' *provides* interfaces can be bound to the goal.

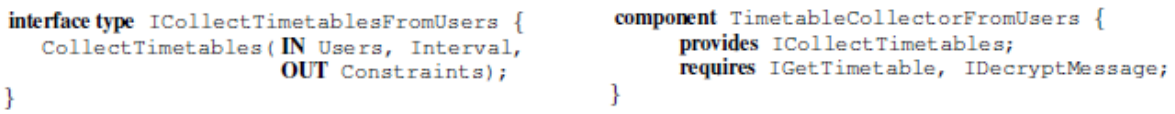


Figura 2.2: Interface Type and Component Example [29]

2.2 GoalD

Motivated by the heterogeneity of devices that have been emerging in the last few decades and the complexity needed for new environments to handle them, Rodrigues [24] suggested, in his master's degree final project, an approach named Goal-D. It focus on the deployment of systems in highly heterogeneous environments based on a goal-oriented design of requirements. By deployment, he means the full stack of operations that go all the way from reasoning which artifacts should be deployed to determining the best configuration through an analysis of the context and finally to getting them up and running.

This methodology is divided in two main sets of activities, with one being offline and the other online. The offline activities are responsible for developing and publishing components and are held by software engineers. In this stage, there are three main tasks that are performed: the first consists on collecting requirements and modeling them as goals, the second relates to mapping the goals obtained into components, and the third performs the packaging of components into artifacts. Those artifacts are stored in a repository for later use in the online activities. The online activities relate to choosing which components, now artifacts, meet the environment's needs based the presented context and the application of these components. The online activities can be divided into two: deployment planning, that analyses the environment and reasons about which artifacts

will satisfy the goals portrayed in the offline stage; and designing the deployment of those artifacts by fetching them on the repository and binding them. Each activity will be further explained next.

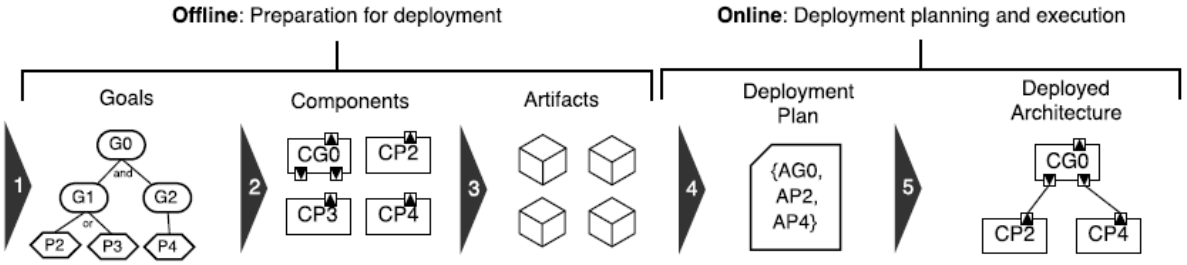


Figura 2.3: Overview of the Steps in Goal-D [24]

The first of the offline activities is Goal modeling, specifically, building contextual goal models as described in the previous subsection. The concept of a context in Goal-D still holds the definition given by Ali et al. [1] but it is described in a more concrete way as a set of resources, meaning that, when present in an environment, a resource points out that a correspondent computational capability is available to operate and is able to achieve the related goal. Contexts, thus, represent restrictions to the suitability of a plan and Goal-D utilizes them to configure variability at deployment time.

The second offline activity consists of obtaining components from goals. Goal-D also borrows the notion of components from literature as being units of composition that have "contractually specified interfaces and explicit context dependencies only."[13] It views the architecture as organized in terms of components and interfaces, but, differently from Yu et al. [29], this approach considers context conditions when associating goals to components.

The AND/OR-refinements are also related to the definitions from literature [28] [29] and play an important role in this activity since each refinement follows different patterns resulting in different kinds of components. AND-refinements follow a pattern in which there is an interface specification for each node and a component specification that requires every sub-node's interface and provides the interface for the node being specified. The satisfaction of this goal happens only when all its subgoals are fulfilled. OR-refinements observe a pattern that maintains an specification of the interface's node and a different component specification for each sub-node, allowing for variability. The goal is accomplished when at least one of its subgoals needs are met. Both refinements can be related to context conditions in context goal models. Goal refinement with contexts alongside AND/OR-patterns strongly support the variability present in heterogeneous environments since they enable the adaptation of systems in different scenarios.

The third offline activity is responsible for packaging components into artifacts. Artifacts, in this methodology, are components packaged into deployment units alongside with a set of metadata. This metadata includes the following informations: name, conditions, defines, implements and depends. The name metadata specifies a unique identification for the artifact; the conditions metadata relates to the artifact's context, or the

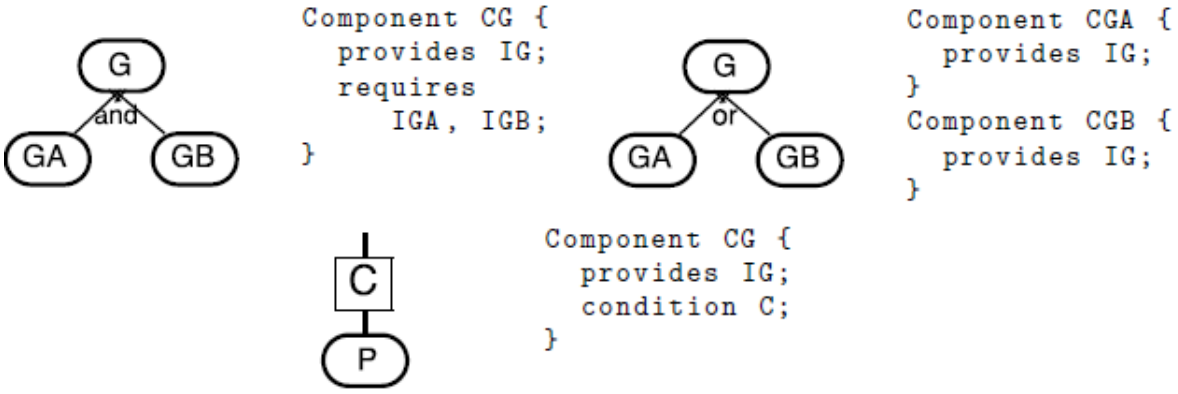


Figura 2.4: Refinement Patterns in Goal-D [24]

restrictions needed for the artifact to be deployed; the defines metadata indicates that an artifact defines the contract for a group of goals; the implements metadata points out that the artifact supplies the implementation for a set of goals; and the depends metadata, indicates a dependency on other artifact's provides and defines.

As a matter of variability, an interface should be packaged in different artifact than the component that implements it, so that components will not be fetched in an environment that they are not going to be used [29]. Artifact packages are then outlined in two different categories, namely, Definition and Implementation. While a Definition artifact encapsulates interfaces and declares the goals it defines, an Implementation artifact packages components with the implements metadata and the depends metadata, announcing the goals it provides implementation for and showing a list of artifacts it depends on, respectively. Finally, after being packaged, an artifact is stored and registered in a repository in order to be seen by prospective environments.

One of the main activities of the online stage, called Automated Deployment Planning, is characterized by the analysis of the computing resources described in the repository that are available in the environment. This is done after an explicit definition of a set of goals by the stakeholders. As a result, a set of artifacts is selected which are able to fulfill the goals considering the available computational resources laid out in the environment. This set is named Deployment Plan. Subsequently, the Deployment execution is responsible for fetching the contents of the Deployment Plan and binding the components from the acquired artifacts.

2.3 OSGi

As a means for the Goal-D methodology to be able to automatically deploy components, described previously, it is essential the usage of a platform that is capable of retrieving artifacts and binding them, based on a Deployment Plan developed beforehand. To this effect, the Java framework OSGi was analyzed for this project. The OSGi technology was developed in 1999 by the non-profit organization called OSGi Alliance [2], formerly known as Open Service Gateway Initiative. It aids in the process of turning

pieces of modular software into components besides guaranteeing the management and concurrence of applications and services held by highly heterogeneous devices.

The OSGi's Framework functionality is described in terms of the four layers that composes it [3]. Built on top of the Java 2, the Security Layer is an optional layer architecture that offers a secure packaging format and is intended to applications that operate in controlled environments. The Module Layer specifies a modularization model by providing support for packaging, deploying, and validating components and applications based on Java. The Life Cycle Layer defines an Application Programming Interface (API) for managing bundles life cycle; moreover, it presents an event API for a better control of the Framework's operations. The Service Layer eases bundle development and deployment by providing a programming pattern that detaches the the specification from the implementation, as suggested in [29].

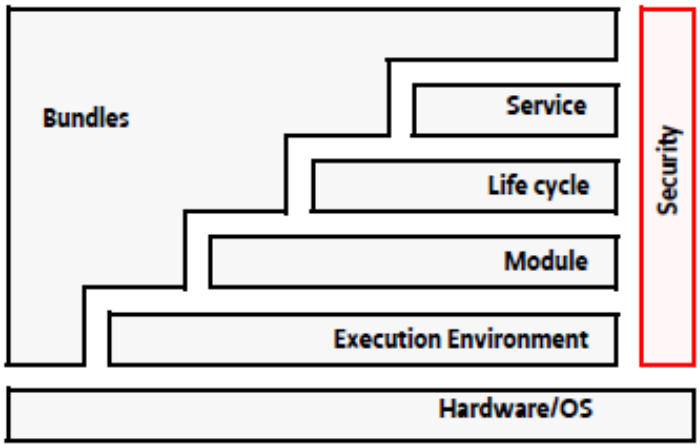


Figura 2.5: OSGi Framework Layers [3]

In the following subsections, the OSGi's chief concepts related to the solution will be laid out. Initially, the definition of bundle will be presented. Afterwards, the idea of services will be deepened in light of the model and applied in three of the most used methodologies. Next, a contract model used in bundle retrieval, titled Requirement-Capability, will be indicated. Lastly, it will be shown how OSGi defines and utilizes repositories for acquiring bundles for usage in deployment time.

2.3.1 Bundles

OSGi's Core Specification defines a *bundle* as "a unit of modularization"[3]. It is also defined as a JAR package consisting of a collection of configuration files, codes and resources [16]. A bundle contains basically two main informations. First, a set of resources, like Java classes, images, HTML files among others, that are required in order to provide some intended functionality. Second, a file called Manifest, which holds all the meta data for describing the bundle's attributes and other contents in the JAR file. A bundle can also contain two optional directories for storing extra information like the

bundle's source code or additional information, in the OSGI-OPT, or service registration information for the Framework, in the OSGI-INF.

The bundle's attributes, or headers, presented in the Manifest file are used to provide useful information for the Framework like its identity, a description of what it holds and instructions on how it should be used. Gédéon [16] divides the headers into three main sections: the mandatory headers, the functional headers and the informational headers. The mandatory headers are the ones needed by the Framework for properly registering a bundle and consist of the *Bundle-manifesVersion* and the *Bundle-symbolicName*. The functional headers relate to the requirements that the bundle needs from the service platform, as the *Bundle-requiredExecutionEnvironment*, *Import-package* and *Export-package*. Lastly, the informational headers provide extra informations for the bundle's clients, like *Bundle-name* and *Bundle-Description*. Furthermore, the Core Specification says that custom headers may be annexed to the Manifest as long as it does not clash with OSGi Alliance defined names or header names registered by other organizations. [3]

A bundle's life cycle observes the following pattern [3]: after the installation of the bundle in the Framework, some of its information is retained in a local bundle cache and the bundle is set to the INSTALLED state. Next, the Framework makes an effort to resolve all of the bundle's dependencies and, if it succeeds, the bundle goes to the RESOLVED state, else, it remains in the INSTALLED state. From the RESOLVED state, while initiating, the module unit is attributed with the STARTING state and becomes ACTIVE when successfully started. After its initialization, if the bundle stops, he goes to the STOPPING state during the stoppage process and goes back to the RESOLVED state if all of his dependencies are still met. Bundles remain stored in the Framework until explicitly uninstalled, when they transit to the UNINSTALLED state. In this state, a bundle cannot move to another state and, if it is still displayed, is because there might be a lost reference to it somewhere.

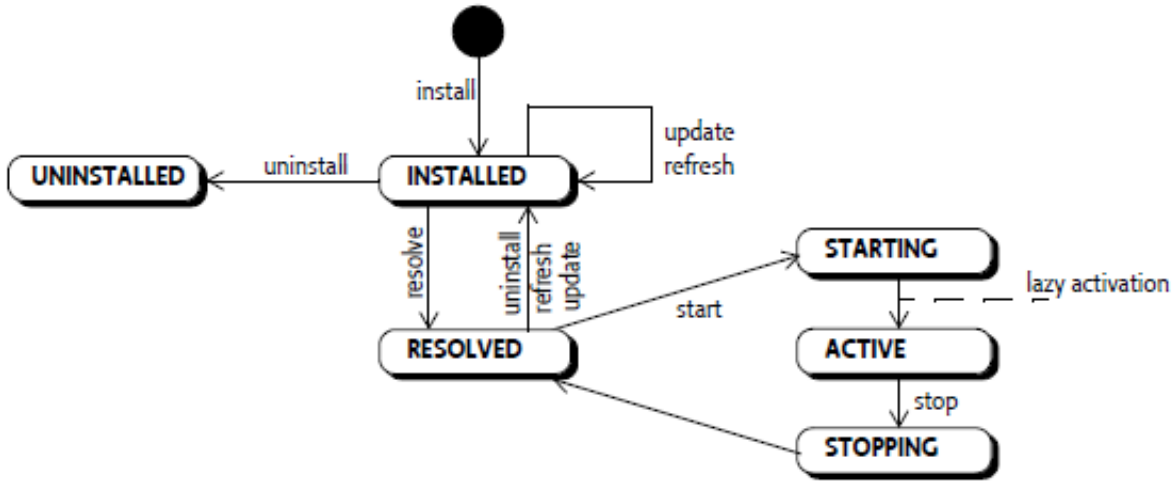


Figura 2.6: Bundle's Life Cycle State Diagram [3]

Hall et al. [17] explicitly defines how bundle's dependency management is performed by the OSGi Framework, also called bundle dependency resolution. They define

resolution as "The process of matching a given bundle's imported packages to exported packages from other bundles and doing so in a consistent way so any given bundle only has access to a single version of any type."The resolution of a bundle may transitively resolve other unsatisfied bundles. Each import package from a bundle is conceptually "wired", or given access, to a matching export one, resulting on a graph of satisfied bundles. In the case of any dependency getting unsatisfied, the resolve process fails and the bundle becomes incapable of starting, staying in the INSTALLED state. Otherwise, the resolved bundles get the RESOLVED state and are ready to initiate execution, transitioning to the ACTIVE state. Moreover, the Framework favors first the highest version, when it comes to multiple matching candidates, and later the installation order, when the matching bundles have the same version.

Although down to code, two other concepts related to bundles that are relevant to a better understanding of the proposal of this project are Bundle Contexts and Bundle Listeners, since they play an important role in the deployment of bundles and in the autonomous characteristic of the solution. Bundle Contexts are Java objects that are utilized for relating the Framework to its installed objects. Every bundle is associated with a Bundle Context object when installed, fact that grants it access to methods that allows the bundle to interact with the Framework. The bundle becomes able to be notified about the Framework's published events, to register a service object, to install new bundles, to get a list of the installed bundles, to get the bundle object for a bundle, among other benefits. Meanwhile, Bundle Listeners are listener interfaces that are associated with bundle events. They are called when a bundle event occurs, namely, when the bundle changes its life cycle state [3]. How these ideas impact the solution will be discussed in later sections.

2.3.2 Services

Even though the modularization of applications into bundles provides several benefits, as shown previously, it also has a drawback. By nature, bundles can not assume that the desired context will be available at runtime due do the volatility of the environment and, thereby, bundles need to be dissociated so that the functional pieces can be organized in different ways, allowing for variability. Nevertheless, decoupling individual modules isolates them, making intrinsically impossible for them to interact with each other. [23] While using the Bundle Dependency resolution may provide a solution, it ties the bundle with implementation details of the provider, not allowing for the different organizations of bundles that may occur. Besides, it handles the resolution on startup, which makes difficult to adapt to environments where bundles may come and go frequently. The notion of service comes in to fill this gap.

Generally speaking, a service is a kind of "work done by another"that implies a contract among the provider and the consumer. From this follows that, as long as the contract is met, the consumer should not be concerned about the implementation details of the service provided. Likewise, services in OSGi are defined by Hall et al. as interfaces in Java that depict contracts within service providers and prospective clients [17], and are defined by the OSGi Core Specification as "an object registered with the service registry

under one or more interfaces together with properties. The service can be discovered and used by bundles."[3] This means that OSGi makes use of the notion of interfaces to detach specification from implementation, thus providing a flexible way of applying several different implementations. This is done by referencing the interface rather than the implementation [23]. Therefore, "the selection of a specific implementation, optimized for a specific need or from a specific vendor, can thereby be deferred to run-time"[3], and, without the need for restarting an application, services can be dynamically swapped, making room for the variability found in heterogeneous environments.

Taking advantage of this handful concept, the OSGi Core Specification defines the Service Layer which delineates a programming model following an interface-based approach that can be portrayed as dynamic, concise and consistent [3] - critical attributes for the implementation proposed by this project due to the diversity of devices in which the Framework is intended to run, allowing for continually stable systems, even with a mix and match of distinct implementations. It is a publish, find and bind model that incorporates service-oriented computing (SOA) concepts in the fact that service providers publish their functionalities in a service registry, becoming visible for clients to search and consume.

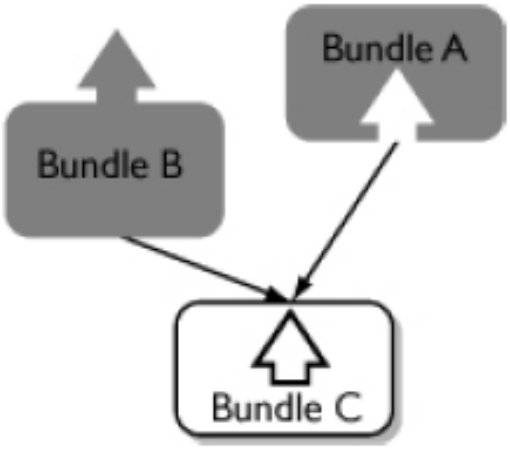


Figura 2.7: An OSGi Service [23]

Since context may vary during the lifetime of a system, several mechanisms were developed to handle the dynamic behavior of services and bundles [23]. Three of them are noteworthy and are described below: the OSGi's Service Tracker, the Service Activator Toolkit and the OSGi's Declarative Services.

First, the Service Tracker is a tool that listens to events published by the Framework that are related to specific services the bundle needs by using Java objects called *service listeners*. Once the tracker realizes that all the required services are registered, the bundle can acquire them and initiate its execution. The variability is handled in the Service Tracker mechanism with the help of *service tracker customizers*, wich are responsible for re-binding bundles when one of the bundles stops providing a service, allowing for an autonomous and uninterrupted execution of the application. Even though it provides a greater control over the bundle's life cycle, this approach is seen as a costly alternative

for having to deal with OSGi's dynamicity by code. It makes the process of balancing the simplicity of modular applications with system scalability highly difficult since adding more services increases the customizer's creation complexity.

Secondly, the third-party Service Activator Toolkit (SAT) is a utility bundle which offers an abstract activator class, thus allowing the re-use of complex but well-tested service listeners. It provides tools that aid in the development of bundle activators, bundle runtime and interdependency analysis and others. In this approach, code complexity clearly diminishes, when compared to Service Tracker since much is pre-built. Nevertheless, its usage generates a considerable overhead which increases start up time, since the activators are loaded even if the services required are not available. Therefore, in order to observe enhancements in comparison with Service Trackers, this utility bundle must be applied in scenarios with a high number of bundles, more than hundreds and below millions, each of them with great probabilities of being initialized.

Lastly, strictly related with the methodology proposed by Yu et al. [29], the Declarative Services is a component model that provides a light-weight methodology for dependency service management [17]. It defines a way for bundles to publish their provided and required services, through an XML file, besides handling the binding and unbinding of services [23]. Moreover, since it has a declarative nature, meaning that it uses a declarative model for registering the service and handling dependencies, there is no need for an activator or any sort of code. This approach is based on the concept of components, which are entities that make reference to, or provide, zero or more services, and consist of an XML file for service description, and a class for bind handling. The XML file is stored inside a folder called OSGI/INF within the bundle and the class binds the consumer with the provider through an initialization method conventionally called *startup*. Declarative Services is very scalable, scaling up to systems with thousands of bundles, having better performance in environments where bundles only run occasionally [23] and, by removing code, the development gets less complex and brings performance enhancements. However, once much is done automatically, maintenance becomes more difficult.

It is remarkable that, in spite of the many nuances in all three approaches, well-structured OSGi systems may be developed based on any of them, once the domain logic gets little or no modifications. Hence, one must take into consideration the environment in which the system is intended to run and the desired benefits in order to make a choice of which methodology to follow. In this project, the Declarative Services approach was chosen due to its proximity with the literature [29] [18] and for having a close relationship with the concepts of the solution proposed by the Goal-D [24].

2.3.3 The Requirement-Capability Model

The usage of a service also suggests a form of discovery or negotiation, indicating that each service implementation has a set of identifying features [17]. This happens because different implementations may vary in certain characteristics like quality, configuration settings or set of Java methods utilized for instance, and the client must explicitly

specify the needed, or even desired, attributes required in a service in order to properly run. This helps in strong contract design between the parties involved in the process.

Nevertheless, interfaces are not a reliable way of defining such strong contracts since they simply define the methods that make up a service, which are implemented by providers, leaving aside useful informations as of "what"the service actually does besides quality and configuration related issues. One form of capturing these knowledge is by attaching them as metadata, i.e. in the Manifest file. By doing so, the Framework becomes able to filter in only the desired services, needing not to load and access the service itself.

Consequently, the OSGi Alliance, in accordance with this pattern, developed a generic constraint model that describes dependency relationships named *Requirements and Capabilities* [3]. This model provides a contract definition through dedicated functional headers that starts by the specification of a distinct namespace in which a unique set of attributes and directives is presented. Attributes are responsible for the matching process, and directives give information about the namespace's semantics. Furthermore, Capabilities are tied to specific Namespaces and are responsible for describing features or functions provided by a resource when active in the environment, by supplying values for its attributes and directives. Requirements, subsequently, symbolize that a given resource is available in the environment when satisfied and are related to specific Capabilities, each having an LDAP filter that matches the related Capability's attributes. A Resource with requirement headers, thus, only function properly when all of the requirements are satisfied by a matching Capability. Accordingly, the declaration of a Capability is made by usign the *Provide-Capability* header in the service provider bundle's Manifest file, while the declaration of a Requirement is make through the *Require-Capability* header. The process of matching Requirements and Capabilities is called resolving.

Augé [10] describes a pet grooming service as an example to illustrate these concepts. Since there are numbers of kinds of pets and skills, tools and facilities required may differ, so clients may find difficulties when finding an appropriate agency to groom pets. He declares a namespace called *pet.grooming* with four attributes: type, that specifies which kind of pets the agency grooms; length, which limits the size of the pet; soap, that names the type of soap utilized; and rate, which points out the rate per hour charged. In each agency's OSGi Bundle Manifest, there can be found a Provide-Capability header presenting the namespace, and a value for each of the four attributes described. The clients requirements are set by specifying the proper namespace along with an LDAP filter for selecting matching agencies. An example of the headers is depicted by figure 2.3.3 below.

```
Provide-Capability: pet.grooming;type:List="cat,horse";length:Long=3000;soap="commercial";rate:Long="20"

Require-Capability: pet.grooming;filter:="(&(type=cat)(rate<=20))"
```

Figura 2.8: Requirement-Capability Model Example

2.3.4 OSGi Bundle Repositories (OBR)

Due to the modular aspect of OSGi-based applications, systems are likely to grow over time, requiring larger groups of bundles, thereby, making more difficult to manage their deployment in an ad-hoc manner. In large systems, an automated solution becomes a *sine qua non* given the impracticality of maintaining hundreds or even thousands of bundles manually. A possible solution is to create an agent that specifies the information related to the installation or update of the bundles externally from the managed bundles [17]. One of the implementations of such solution is the OSGi Bundle repository (OBR). It is a proposal for a specification of the OSGi Technology that intends to deal with bundle deployment in a two-fold approach: the discovery of the needed bundles amongst the ones that are available for deployment, and the deployment of the desired bundles and its dependencies.

For the discovery process, a place where the JAR files of bundles are stored, along with other resources, is defined as a repository. This storage is described by an XML file which specifies each resource through its metadata, allowing to search the repository for specific resources without needing to fetch them first. Resources are a generic representations of artifacts, such as bundles, certificates, configuration files and others, whose metadata may contain the requirements or capabilities which are inserted onto the XML file. This file can be created by hand, but there are several tools that provides automatic and more reliable ways of storing the artifacts, like Maven integration used in this project.

The image 2.9 shows an example of a repository descriptor file. The root element is a tag that contains as attributes the name of the repository and the date in which the repository was last modified, so the OBR can identify when changes occur. For each resource there is a block of tags, in which the bundle's Manifest headers are laid out as tags or tag attributes. Tags named *capability* represent the packages that a bundle or a resource exports, and can be tied to the Manifest header *Export-Package* for instance. The tag property *name*, which determines the name of the package exported, and the other properties are based on the header attributes. Likewise, the *require* tag can be related to the *Import-Package* header, and represents a resource's required package. The *name* property also defines the name of the needed package and the property *filter* has the LDAP filter that matches a specific package. With the information in the XML file, and using an API defined in the OSGi Enterprise Specification [4], the OBR is able to resolve a set of bundles to be deployed.

With the information in the XML file, and using an API defined in the OSGi Enterprise Specification [4], the OBR is able to resolve a set of bundles to be deployed.

The benefits of using XML-based repositories and the OBR approach come specially from the simplicity of the approach. Once a repository is only a place where bundles are stored, any developer can build his own bundle repository by describing in an XML file the set of bundles he wants to publish. Besides, even though possible, XML-based repositories do not require a server-side process, since the OBR API handles all the job. Repositories can also refer to other repositories, creating a federation structure and allowing for the discovery of a greater number of resources, thus, reducing the risk of failures in the fetching process.

```
<repository lastmodified='20171111081235.518'>
    <resource id='bundle1_01/1.0.0' symbolicname='bundle1' version='1.0.0'>
    <description>Bundle 01</description>
    <size>3939</size>
    <capability name='bundle'>
      <p n='symbolicname' v='bundle1'/>
      <p n='presentationname' v='Bundle 01'/>
      <p n='version' t='version' v='1.0.0'/>
      <p n='manifestversion' v='2'/>
    </capability>
    <capability name='package'>
      <p n='package' v='Bundle'/>
      <p n='version' t='version' v='0.0.0'/>
      <p n='uses:' v='Bundle01_api'/>
    </capability>
    <require name='package' filter='(&amp;(package=Bundle01_api)(version&gt;=1.0.0))'>
        Import package g1_get_position_api;version=1.0
    </require>
  </resource>
</repository>
```

Figura 2.9: Repository XML File Format

2.4 Apache Maven Development Tool

Maven is a "software project management and comprehension tool"which manages the build, reporting and documentation of a given project based on the concept of a project object model (POM) [22]. It aims to provide the developer with a greater comprehension of the complete state of a development effort in the minimal amount of time possible.

It revolves around the concept of a build life cycle, which is divided into seven distinct phases, namely, validate, compile, test, package, verify, install and deploy. First, in the validate phase, the project is validated by checking its correctness and the availability of the needed information. Second, the compile phase involves the compilation of the project. Third, in the test phase, the compiled code is tested with the help of a unit testing framework. Fourth, in the package phase, the compiled code is packaged in a distributed format. Next, the verify state relates to checks on the results of the tests to ensure quality. Later, in the install phase, the package is installed in the local repository. Lastly, the deploy stage is responsible for copying the final package to a remote repository [20].

The Project Object File, also known as POM, is an XML file that is considered the basic unit of work in Maven, which holds information related to the project and build configuration details. Among the set of possible tags that go inside this files, some are noteworthy and are described next. The *project* is the root tag for defining a project; the *modelVersion* tag relates to the model of the POM file, which should be set to '4.0.0', as for today; the *groupID* tag uniquely identifies the project's group, while the *artifactId* tag identifies the project; *version* specifies the version of the artifact in the specified group; *name* is related to the actual name of the project and *description* brings a short description of it; *package* specifies which type of artifact will be built; *dependencies* brings

a list of the projects that the project depends on in order to function properly; the *build* tag lays out the information needed for the build of the project; as a child of the *build* tag, the *plugin* tag are artifacts that provide goals to Maven, which are bound to the phases of the build's life cycle; lastly, the *distribuitionManagement* tag holds the information of the repository to which the artifact will be registered [21].

In this chapter, chief concepts related to the solution proposed by this project where laid out. First, general concepts of Goal-Oriented Requirements Engineering were presented as Goal modeling, Goal-Oriented design and Contextual Goal Models. Afterwards, the Goal-D approach was thoroughly described in order to properly contextualize this project. The online and the offline phases were described and each activity was delineated, along with important concepts like context, artifacts and others. Next, the OSGi technology, specially the OSGi Framework was presented. Bundles were presented with the Manifest file and the bundle's life cycle. Later, it was shown how the OSGi uses the notion of services to provide an approach that favors the variability intended by the Goal-D and the definition of strong contracts for these services with the usage of the Requirement-Capability model. Lastly, the Apache Maven was detailed once it was used in some stages of the development of this project.

Capítulo 3

Filling Station Advisor

The motivating example that will be used throughout this work is the application described by Rodrigues in [24] called Filling Station Advisor. As the name suggests, this application revolves around providing the driver, the human or the artificial intelligence behind the wheel, places where the car can be refueled or recharged, taking into account proximity and convenience. A station can be conveniently reached if it meets certain prerequisites, like the type of fuel that the car runs on or reachability considering the amount of fuel left, as well as some user preferences, like route deviation, or the quality of service provided.

One of the main characteristics of the advisor is that it is intended to run in environments where the set of devices may vary. It may run on a smart phone or a system navigation, calculate the fuel autonomy by reading fuel levels or by using data from onboard computer, search for stations using Internet access or relying on user input, amongst other possibilities. The advisor, thereby, must autonomously indicate the best station taking into account the heterogeneity of the environment without losing track of the user preferences.

In an effort to develop an application that matches the attributes described, Rodrigues modeled the application requirements as goals, following the methodology proposed by literature [1] [12] [28] [29]. The requisite *Assist vehicle refueling* was modeled as the root goal, and refined in five subgoals, each considering a few of the different contexts that may happen. Initially, the subgoal named *Get Position* is responsible for acquiring the vehicle position, either with an antenna or with a GPS system. Secondly, the *Assess distance to empty* subgoal intends to calculate how long the amount of fuel is going to last, through the data obtained by an on-board computer, accessing data about fuel level, or by getting user input. Thirdly, the subgoal called *Recover information about nearby filling stations* gathers information of the filling stations available within reach online or based on a cached result. Next, *Decide on the most convenient filling station* subgoal uses the information obtained by the other subgoals and the user preferences to choose the most convenient filling station. Lastly, the subgoal *Notify driver* is responsible for deciding how and when to notify the driver, by using the navigation system, a synthesized or prerecorded voice, or an on-screen notification. Figure 3.1 better illustrates this idea.

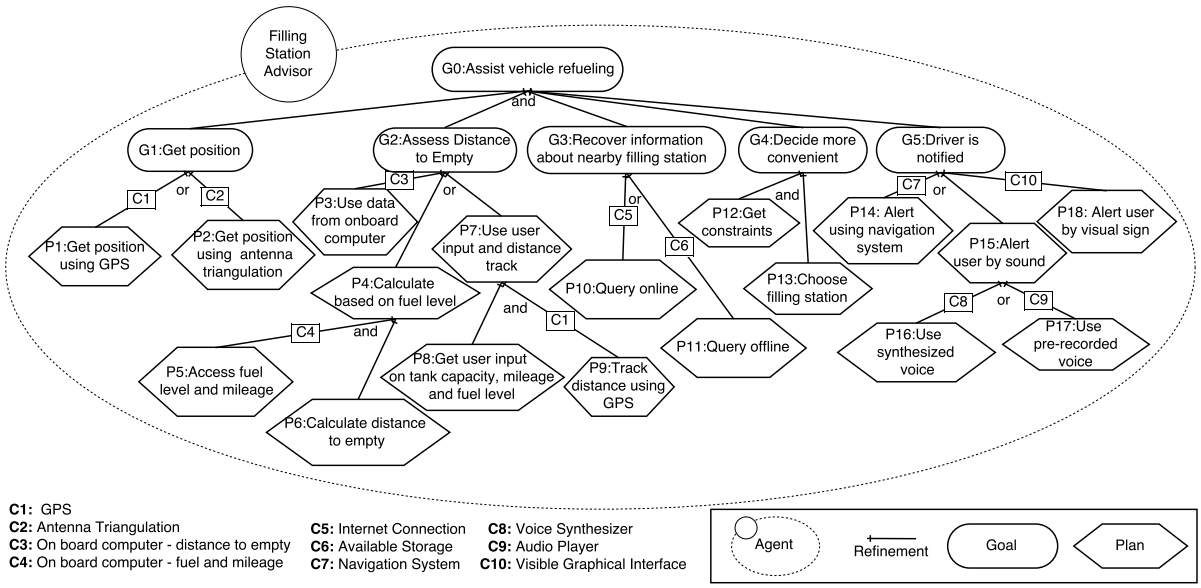


Figura 3.1: CGM of the Filling Station Advisor [24]

Continuing on this track, a contextual goal model was designed to depict the goals and subgoals in a tree structure. Each goal described was assigned to a label: G0, G1, G2, G3, G4 and G5, respectively. *G0: Assist vehicle refueling* was AND-refined into the other five subgoals since it needs all of them to be satisfied in order to be fulfilled. The goals G1, G2, G3 and G5 were OR-refined, meaning that, for each, if any of its subtrees have their requirements met, the goal is achieved, thus making variability feasible, seeing that any of the different settings can be used. In addition, contexts are portrayed in the associations as required context [1] denoting that the plan is executed only if the context holds. Figure 3.2 shows the variability context expected for the subgoals, defining the context space of the environment in which the system is intended to run.

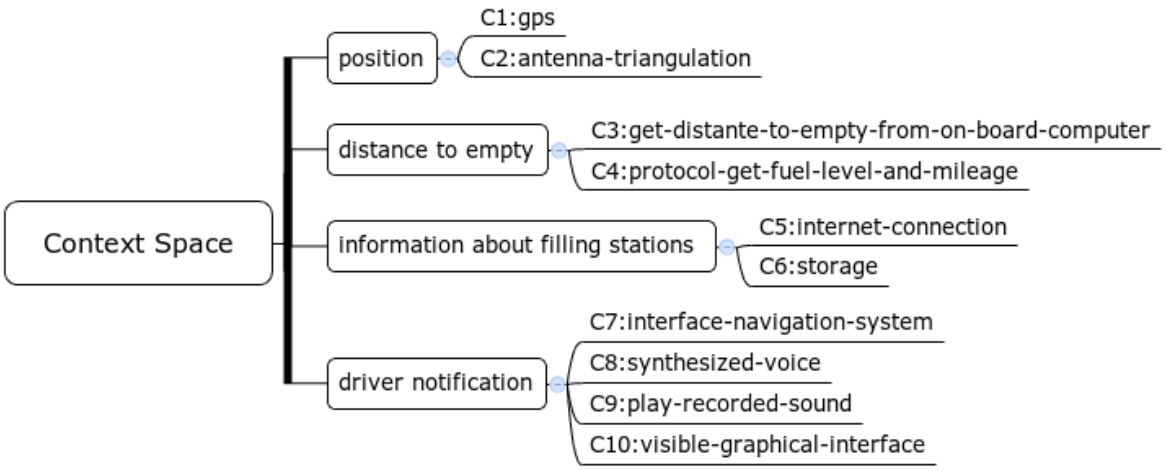


Figura 3.2: Context Space of the Filling Station Advisor [24]

Capítulo 4

Goal-Oriented OSGi Environment

The GoalD methodology [24], described in the background chapter, provides an autonomous way of deploying systems based on a Goal-Oriented approach. Rodrigues defines a set of offline and online activities that pervades all of the stages required for the deployment of an application, that goes from goal modeling to fetching and binding units of a modular system in a heterogeneous environment. Nevertheless, after a careful reading, it is possible to observe that his analysis offers just a proof of concept of GoalD, meaning that his project revolves around verifying that his methodology has potential for real-world application. By using an evaluation methodology called Goal-Question-Metric (GQM), the feasibility of GoalD was tested on the Filling Station Advisor application by checking the reliability, efficiency and scalability of his planning algorithm, leaving the assessment with an actual implementation of a modular system, integrated with a framework for fetching and binding artifacts, for a future stage of his work. Therefore, even though his proposal was met, his description lacks a way of certifying in a concrete manner the feasibility of the method.

For the purpose of providing a fully functional implementation of the GoalD approach, this work will explore the OSGi Framework and technology as a potential solution. The so described as "the best model to modularize Java"[2] was chosen to compose this project since it suits GoalD's needs in at least three ways.

First, it provides a very flexible, open and common architecture for a diverse group of developers and service providers allowing for the development of diverse smart devices, offering the variability needed by GoalD's intended background [24]. The Core Specification states that this technology "targets set-top boxes, service gateways, cable modems, consumer electronics, PCs, industrial computers, cars, mobile phones, and more"[3]. In other words, the OSGi Framework aims to work on a variety of devices with differing hardware attributes. This broad range of intended computing environments is due to the general notion given to resources, responsible for providing the features and the functions packaged in bundles, which can basically be anything from only standard Java classes to displays and secure USB key stores. Rodrigues, on the same track, characterizes these computing environments as highly heterogeneous and states that the first challenge of his project is to tackle the non-uniformity of such environments. Consequently, as the inten-

ded set up for the GoalD methodology is the one that OSGi focuses on, this technology was opted for composing such analysis.

Second, it allows for a run-time selection of available implementations. The programming model brought by the OSGi Framework's service layer decouples the specification of a service from its implementation, letting applications bind to services only through the specification. Hence, the selection of the specific implementation of a service can be postponed to run-time. The Core Specification describes this model as collaborative, meaning that "the service layer must provide a mechanism for bundles to publish, find, and bind to each other's services without having a priori knowledge of those bundles [3]". In the meanwhile, Rodrigues' project defines as his second challenge the consideration of the uncertainty at design time, once the set up of the computing environment of the end user cannot be previously asserted by the architect or the developer of the system. Therefore, the service layer's intentions match the challenge proposed, making OSGi a viable candidate for implementing the GoalD approach.

Thirdly, It offers the desired automatic way of binding services. The OSGi Framework's Service Layer makes use of a service registry which enables bundles to add new services, check for available services or be informed of the state of services through an specific API. This empowers systems to install new features or update and modify existing ones in a programmatic way, without needing human intervention in the process, allowing for dynamic environments where services may come and go and bundles can be updated on the fly. The Core Specification points out that "the service mechanism must be able to handle changes in the outside world and underlying structures directly"[3], and then describes the API for handling the mentioned changes by code. Tied to this characteristic, the third challenge laid out in the GoalD's proposal relates to the automatic deployment of the components, seeing that a deployment specialist may not be available at deployment time given a certain environment. Thus, the OSGi offers an automatic framework that meets GoalD's intentions, attesting the OSGi as a sound option for a GoalD implementation.

Besides the rationale presented, the OSGi technology brings other noteworthy benefits that add to it as an option to underlie the GoalD implementation, like enabling a low-cost development and maintenance of components, due to the capability of incorporating pre-built and pre-tested modules into an application [2]; reduction of the complexity of the code, for it makes use of a modular architecture that is convenient for both large-scale and small, even embedded, systems; dynamically handling of bundles, which allows bundles to be started, stopped, updated and uninstalled without requiring the system to restart; and others. A more complete list of the benefits of using OSGi can be found in [5].

Hence, owing to the previously described features, the OSGi technology was found to be a remarkable match for Rodrigues' proposed methodology and was used in this project to implement the GoalD's approach.

As to certify that the OSGi technology is a strong match for the GoalD methodology, an OSGi implementation for an autonomous Goal-Oriented deployment was developed using the approach defined by Rodrigues in his project. Such implementation

was built in a twofold approach, following the two sets of activities outlined by GoalD: the development of bundles as GoalD artifacts, which comprehends the offline stages, and an OSGi framework implementation, for the automated deployment of the bundles of the online activities.

This solution will provide GoalD with a fully functional implementation of its approach, by integrating all of the offline and online activities with the OSGi Technology. In this chapter, the proposed solution will be laid out along with specific details. For a better understanding, it was divided into the two set of activities used by GoalD, the offline and the online stages. In each, the concepts from Goal-Oriented Requirements Engineering and from the GoalD approach will be mapped to the OSGi technology through a description of the implementation of each GoalD activity, initially the offline activities, followed by the online ones.

4.1 Offline Activities

This subsection will bring an in-depth description of the GoalD's offline activities in light of the OSGi implementation proposed as a solution. As described in Section 2.2, the offline activities consist of: requirement modeling as goals, mapping goals into components and packaging components into artifacts.

4.1.1 Goal Modeling

Some of the activities of the offline stage require partial or no integration with the OSGi technology to function properly, as it is the case of the first activity, namely, goal modeling. In this activity, a Deployment Goal Model, which basically is a Contextual Goal Model that takes into account the notion of resources, is developed in order to specify the restrictions faced by the deployment environment. Hence, the requirements engineer holds responsibility on eliciting the requirements and designing them as goals, depicting them as a Contextual Goal Model. Rodrigues states that a requirements engineer along with the participation of a domain specialist, possibly the user, are responsible for coordinating this phase. In the Filling Station Advisor, for example, the goal model showed in Figure 3.1 portrays the work done by the specialist after modeling requirements as goals and applying the notion of context.

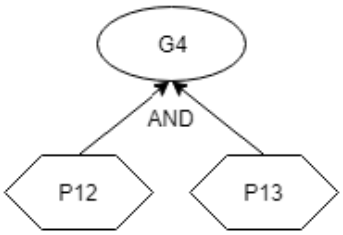
Goals can be seen in terms of services provided by OSGi bundles. Each goal, in order to be satisfied, relies on the satisfaction of all of its sub-goals, or at least one of them, depending on the kind of refinement applied. Analogously, a service can only be provided if all of the services required by him are being provided. In the case of tasks, the leaf-nodes of the tree, they work as actual implementation of the services offered by means of the OSGi bundles, with no dependencies. In case of the root goal, it can be seen as a regular goal that depends on other services, but provides the application as a general. This parallelism enables the variability in the computing environment since

goals, as services, can be fulfilled by more than one subgoal, or service implementation due to all the interface binding described in Section 2.3.2

4.1.2 Mapping Components

Mapping goals into components is an example of an activity that partially needs the integration with a framework. It is a twofold assignment composed of component analysis and component development. On the one hand, the component analysis task does not require the usage of the OSGi Framework to be fulfilled, only the expertise of the architect in charge. He receives the Deployment Goal model obtained from the earlier stage and identifies the components and interfaces that should be developed in order to reflect the model. On the other hand, the component development task requires the use of the OSGi technology for the creation of the components that will be used in future stages of the approach.

In this project, components are developed as Java classes and are built using the same pattern designed by *Yu et al.* [29], but enhanced by Rodrigues for supporting context restrictions. For each AND-refinement, there is a Java Interface for every sub-node, that serves as a service specification for the OSGi Service Layer, and a class, or a set of classes, that relies on each of the sub-nodes Interfaces as dependencies, and implements the node interface. An example to illustrate the usage of AND-refinements is the G4 of the Filling Station, which is responsible for choosing the most convenient filling station and is AND-refined in *P12: Get Constraints* and *P13: Choose Filling Station*. This goal can be mapped into two Java classes: a Java Interface that defines the API for the *G4: Decide more convenient*, and other that references the P12 and P13 interfaces by using their methods in its domain logic, and that implements the API for G4. Figure 4.1 depicts the resulting specification.




```java
package g4_decide_more_convenient_api;

public interface DecideConvenient {
    String makeDecision ();
}
```

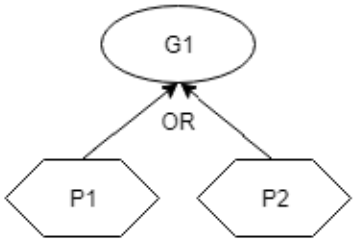
```java
package g4_decide_more_convenient;

import g4_decide_more_convenient_api.DecideConvenient;
import p12_get_constraints_api.GetConstraints;
import p13_choose_station_api.ChooseStation;

public class DecideConvenientImpl implements DecideConvenient {

    private GetConstraints constraints;
    private ChooseStation choice;

}
```

Figura 4.1: AND-Refinement - G4 Component Specification

OR-refinements, differently, result in one Java interface specification for the goal node being described and an implementation class for each sub-node, in order to allow the

sub-goals to, when satisfied, provide the service depicted as the goal node. An example that can be taken from the Filling Station Advisor application is the G1, which relates to getting the vehicle's position in some sort of way, thus, being OR-refined in two possibilities *P1: Get position using GPS* and *P2: get position using antenna triangulation.* As a result, the specification for this goal will be developed by two Java classes, an interface, that defines the API contract of the service provided by G1, and two classes that implement this API using different hardware definitions - a GPS system and an antenna triangulation. The result can be seen in Figure 4.2.

Subsequently, the main difference between AND- and OR-refinements is that each sub-goal is a different implementation of the service API of the goal, while every subgoal in an AND-refinement has its own API and implementation and the goal is satisfied only when all services provided by its sub-nodes are available. Nevertheless, depicting goals as services also brings variability for AND-refinements. This happens since, by performing an AND-refinement pattern, Java interfaces are developed for every sub-goal, which allows different service implementations for each of these interfaces, a similar result obtained when OR-refining a goal.




```java
package g1_get_position_api;
public interface GetPosition {
    public String getPosition();
}
package p1_gps_tracker_01;
import g1_get_position_api.GetPosition;
public class GPSTracker01 implements GetPosition {
    public String getPosition() {
        return "P1: Using GPS signal";
    }
}
package p2_antenna_triangulation_01;
import g1_get_position_api.GetPosition;
public class AntennaTriangulation01 implements GetPosition {
    public String getPosition() {
        return "P2: Using Antenna tringulation";
    }
}
```

Figura 4.2: OR-Refinement - G1 Component Specification

However, this project brings a different means for defining contexts. While Rodrigues, with his extended nomenclature for depicting contexts in goal models, propagates context conditions inside of the body of the component as explicit attributes in form of *condition* elements, the Framework proposed by this project portrays context conditions as metadata headers in the manifest file of a bundle. This choice was made for two main reasons.

Firstly, this information is not related to the code itself and probably would not be used by the internal structures to provide a service since, if the context condition is not met, that branch of the goal model tree would never be chosen and, thereby, the very bundle itself would not be fetched. As a consequence, if the context condition were to be stored inside the Java class, the information would be lost and, thus, of no use. Secondly, if stored as headers, context conditions would be easily accessed by the

planner. The headers of the Manifest file become tags in the XML file which are used by the OBR to search for bundles, meaning that, if this information is stored as a header, the Goal-D's planner algorithm would have easy access to it when generating a feasible deployment plan, for it relies on the repository file to create a plan. Hence, by searching for a more cohesive implementation and due to the purpose of the context condition as to limit alternative strategies in deployment time, this project chose to allocate this piece of information inside the Manifest header file of the component bundle. Where and how the headers are stored will be further explained next.

4.1.3 Packaging Artifacts

The last offline activity of GoalD is the packaging of components into artifacts. As outlined previously, components are seen as Java classes by the OSGi technology, which can be either interfaces or regular classes. Since classes are packaged as bundles, there can be observed a one-to-one correspondence between artifacts and bundles. Further explained in Section 2.3.1, bundles can be defined as JAR packages containing a set of Java classes, a metadata file that describes the bundle's attributes, and other resources. In a similar manner, Rodrigues declares that "GoalD's artifacts include packaged components and interfaces as well as metadata that describe the packaged elements". Therefore, bundles and artifacts have a similar definition and are used to store the same kind of data, making an easy conceptual match among the OSGi technology and the GoalD approach. This subsection will first present the information that is packaged inside a bundle, while specifying the relation between the artifacts attributes used by GoalD and the headers that are used by the OSGi in the Manifest file. Then, we describe how artifacts are added to repositories.

Package Metadata

Two pieces of metadata information are packaged inside a bundle. The first is the manifest file, which relates to the general metadata stored in order to properly identify the bundle and its attributes. The second is the declarative services metadata, which defines the services provided or required by the bundle.

Initially, the manifest file can be correlated with the artifacts identification metadata. The artifacts metadata are responsible for holding information related to goals, dependencies and context conditions in five main attributes: *name*, *conditions*, *defines*, *implements* and *depends*. Similarly, the OSGi Core Specification states that a bundle "can carry descriptive information about itself in the manifest file that is contained in its JAR file [3]". This manifest file contains headers that may be described as reserved headers [6], or custom headers. The *Bundle-SymbolicName* reserved header is said to, along with the *version* header, uniquely identify a bundle, being used in the proposed implementation of this project to represent the *name* artifact attribute. The bundle header called *Bundle-Condition*, dissimilarly, is a custom header developed particularly for this project to store information about the context conditions of the environment and relates to the *depends* artifact attribute. Figure 4.3 shows part of the Manifest files for the bundle package of

the tasks *P1: Get position using GPS* and *P2: Get position using antenna triangulation*, with their respective context conditions.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: GPS Tracker 01
Bundle-SymbolicName: p1_gps_tracker_01
Bundle-Version: 1.0.0.qualifier
Import-Package: g1_get_position_api
Bundle-Context: gps_capability
```

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Antenna Triangulation Service 01
Bundle-SymbolicName: p2_antenna_triangulation_01
Bundle-Version: 1.0.0.qualifier
Import-Package: g1_get_position_api
Bundle-Context: antenna_triangulation
```

Figura 4.3: P1 and P2 Manifest Files

Moreover, other OSGi headers can be used to identify the type of artifact. As defined in the GoalD methodology, different kinds of artifacts may hold different attributes, since interfaces and components are packaged into separate artifacts to favor variability and low-coupling. Definition artifacts are packaged also with the *defines* metadata, which stores a list of goals that such artifacts provide a definition for, while implementation artifacts declare the *implements* metadata and hold a list of the goals, or service, provided by the packaged components. In OSGi, we see the same happening. Java classes and interfaces are packaged in separated bundles, allowing for different implementation of services to come and go in runtime. Next, we see how both kinds of artifacts are implemented with this technology.

Definition Artifacts are associated with bundles that provide the API contract of services, defined in this project as API bundles, which declare a reserved header in the Manifest file called *Export-Package*. This header lists the contracts the bundle defines. Service providers and clients hold in their header file the *Import-Package* reserved header, specifying which APIs they make use of. In the Filling Station Advisor, an example of an API bundle, or a *definition* artifact, is the contract definition of the service related to *G1: Get position*. It will be packaged into a bundle, only with its interface definition alongside its manifest header file, depicted in Figure 4.4

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Get Position API
Bundle-SymbolicName: g1_get_position_api
Bundle-Version: 1.0.0.qualifier
Export-Package: g1_get_position_api
```

Figura 4.4: G1 Manifest File

Similarly, implementation artifacts are liable of holding the *implements* and the *depends* metadata, both related to a list of goals that the component either provides or relies on, depending on the AND/OR-patterns described earlier. In OSGi terms, we define goal as a service, which provides the intended functionality only if all of its dependencies are met. As a means for variability, each goal implements a service with a unique set of identifying features, suggesting that, depending on the available context, services with different features will be more desirable. Therefore, in order to describe accordingly this idea, the Requirement-Capability model, explained in length in Section 2.3.3, was used.

Both *implements* and *depends* metadata are packaged into bundles' manifest files as follows: on the one hand, the *implements* artifact is stored inside the bundle manifest file as a *Provide-Capability* functional header, specifying the list of provided services based on the namespace and the service's attributes. On the other hand, the *depends* artifact metadata is placed in the bundle's Manifest file as a functional header *Require-Capability*, which delineates a list of services that the bundle requires for a proper execution, based on a Namespace and an LDAP query that matches with the attributes of a service provider in the same Namespace. As an example, the Manifest file of of the *P7: User input and distance track* of the goal model is depicted in Figure 4.5.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: P7 Use User Input on Distance Track
Bundle-SymbolicName: p7_use_user_input
Bundle-Version: 1.0.0.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Import-Package: g2_distance_to_empty_api,
                p8_get_user_input_api,
                p9_track_distance_api
Provide-Capability: unb.tg.osgi.distance;symbolicname="g2_distance_to_empty_api";version:Version=1.0
Require-Capability: unb.tg.osgi.input;filter:="(symbolicname=p8_get_user_input_api)",
                    unb.tg.osgi.position;filter:="(&(symbolicname=p9_track_distance_api)(precision=100))"
Bundle-ActivationPolicy: lazy
Service-Component: OSGI-INF/component.xml
```

Figura 4.5: P7 Manifest File

Lastly, declarative services metadata are also packaged into bundles for service definition. OSGi bundles not only are responsible for storing classes and metadata, but they may also contain optional directories for holding extra information, as described in section 2.3.1. As stated in Section 2.3.2, this project uses concepts from the Declarative Services approach to publish and bind the services in the OSGi implementation, for it is a lightweight methodology and close to concepts already described. Declarative Services add to the bundle package a folder called OSGI/INF, which contains an XML file used to describe the service provided, along with the required service dependencies.

Service provisions are laid out in the XML file through the *implementation* tag, with the class attribute that expresses the fully qualified name of the class that implements that service, and the *service* tag, which holds the *provide* tag with the interface attribute, meaning that the described component provides the interface attribute as a service. Conversely, service requirements declare dependency on other services by using the *reference* tag, which has the following attributes: Name, which is the name of the reference; Interface, that specifies the fully qualified name of the class that the component uses to access the service; Cardinality, which states whether the reference is mandatory and if the component implementation supports single bound or multiple bound services; Policy, which relates to the dynamicity of the component; Bind, which tells the name of the binding method of the component implementation class with the service required; amongst others. Declarative Services also define the *Service-Component* header in the manifest file, indicating the name of the XML file that will be used.

Figure 4.6 shows a snapshot of the XML file packaged in the *P15: Alert user by sound* bundle implemented. This bundle provides the service related to G5, while

referencing service implemented by one of the contexts c8 or c9, which are the Voice Synthesizer and the Audio Player.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
               activate="activate" deactivate="deactivate" name="p15_alert_by_sound.AlertBySoundImpl">
   <implementation class="p15_alert_by_sound.AlertBySoundImpl"/>
   <service>
      <provide interface="g5_driver_is_notified_api.NotifyDriver"/>
   </service>
   <reference bind="setAlert" cardinality="1..1" interface="p15_alert_by_sound_api.AlertBySound"
               name="AlertBySound" policy="static"/>
</scr:component>
```

Figura 4.6: P15 XML File

Repository Storage

After packaging the components, with their respective metadata and service registry information, inside of bundles, the Goal-D methodology specifies that the resulting bundles should be registered in a repository in order to be distributed to the target environment. In this project, the M2Eclipse project, that provides an Apache Maven Project integration for the Eclipse IDE, was used for registering the bundles obtained from the earlier activities into a local repository.

In order to register a bundle, each bundle project was given a Maven nature, meaning that the internal structures as folder organizations were slightly modified as to conform to a Maven project. In this process, besides some Maven Dependencies that were added, a POM XML file was created to cope with the deployment cycle of the project by Maven. In this file, some standard configurations needed to be set so that the bundle could be correctly enlisted to the repository, in other words, some XML tags must have the following default values in order to properly register the bundle to the repository. First, to the *packaging* tag should be given the value 'bundle', for it will tell Maven to package the artifact with the specific attributes of a bundle, not only as a simple JAR file. Next, the *artifactId* and *version* tags were set with the same values as the *Bundle-SymbolicName* and *Bundle-Version* functional headers, since they have the same function of providing a unique identification. The artifact *org.osgi.core* must be added as a dependency in the *dependencies* tag, for it is used throughout the deployment and execution of the bundle, so it must be declared. Any other bundles that it depends on must be added also as dependencies in the same way, for example, the bundles that provide the packages that are imported through the *Import-Package* header.

Inside the *build* tags, there are two main configurations. The first is related to additional resources that should be packaged inside the bundle, the *resources* tag, in which the name of the Declarative Services XML file folder ought to be specified. The second is the *plugin* set of tags, where the *maven-bundle-plugin* is to be stated. The *configuration* tag, child of the *plugins*, will hold the most important pieces of information in regarding to the bundle's metadata, such as the headers *Bundle-SymbolicName*, *Bundle-Version*, *Import-Package*, *Provide-Capability*, *Require-Capability*, *Service-Component*, and others. The *maven-compiler-plugin* is also another artifact that should be added inside the *plugins*

tags. *DistributionManagement* tags hold the identification and place of the repository to which the bundle ought to be registered. Lastly, and least important, the *name* and *description* tags should also be set with valid values, since they provide a human readable description of the bundle, which can be accessed through OBRs for manual fetching.

After the configuration of the POM file, Maven goes through all of its build life cycle phases in order to enlist the bundle into the remote repository. When it gets to the install phase, the bundle is registered in a repository placed locally, usually inside of the .m2 folder in the home folder, along with the created XML file for the description of the registered bundles. Finally, when it gets to the deploy phase, Maven submits the bundle to a remote repository.

In sum, this section has shown that goals can be seen as services provided by OSGi bundles, while components are Java classes obtained by applying AND/OR-refinement patterns in every node of the goal model. As components are packaged in artifacts, Java classes are packaged into bundles, along with their specific metadata in terms of headers in the manifest file and service registration information. These bundles may contain only Java interfaces, for definition artifacts, or implementations of these interfaces for implements artifacts and are registered in remote bundle repositories through the build life cycle of the Maven project. These repositories consist of basically an XML file with the description of the stored bundles for later search.

4.2 Online Activities

This section explains how the online activities of the Goal-D approach were implemented using the OSGi technology as a base ground. In order to attest to the feasibility of Goal-D, a concrete implementation of the online activities was built, through the development of a supporting framework that provides the characteristics desired by GoalD, detailed in Section 2.2, found here [8] The project metamodel is presented first. Then, the online activities, namely, the automated deployment planning and the deployment execution, are described in regard to their relation with the OSGi technology, along with the different processes performed by each.

4.2.1 Conceptual Model

The conceptual model of our framework is depicted in the class diagram of Figure 4.7. It consists of four major elements: the Environment Interface, the Launcher, the Planner and the OBR agent, which is specialized as a local and a remote repository agent.

First, the Environment Interface is a boundary stereotype which identifies an entity that interfaces the communication between the Framework and the outer world. It has a two-fold liability: listen to context changes and inform the Framework about new goals set by the user. As the GoalD approach is intended to run in highly dynamic environments where computational devices may come and go, this entity is responsible

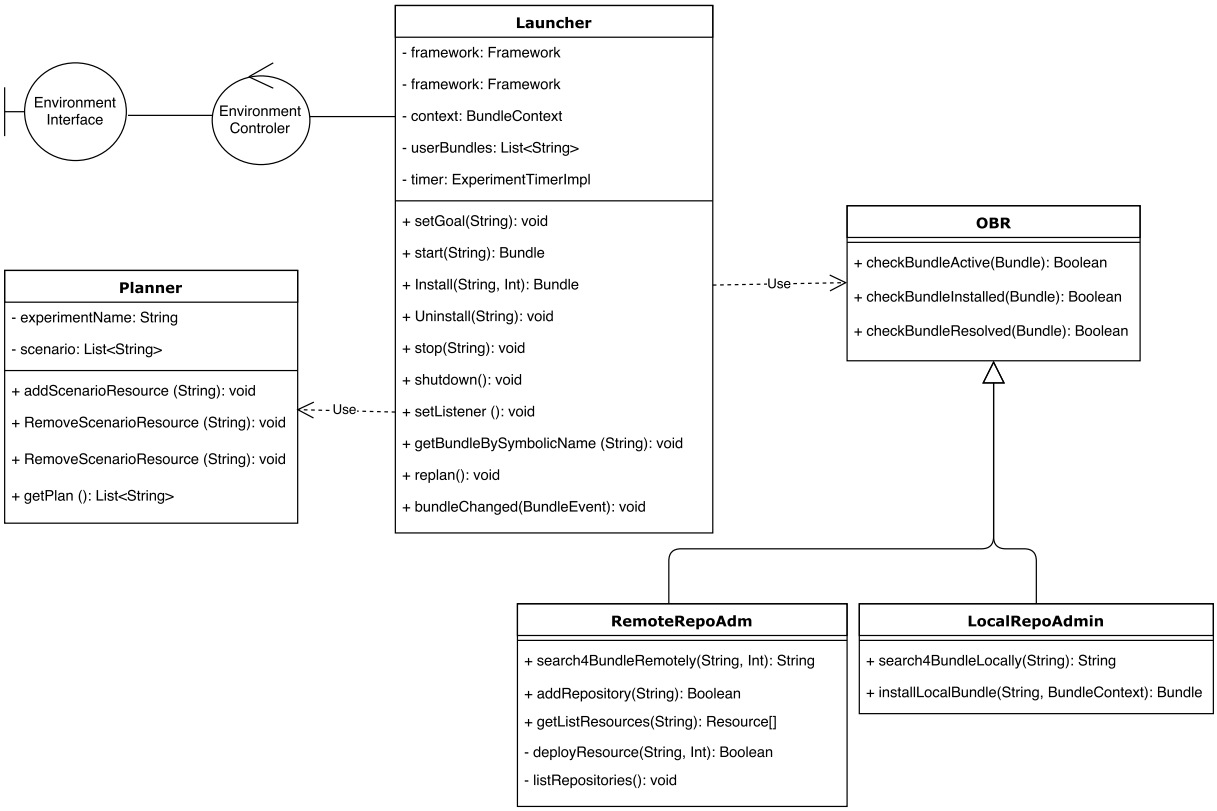


Figura 4.7: Framework's Class Diagram

for reading the context, listening for possible changes, so that the system may work autonomously. Besides, different goals may be desired by the end user, which must be acknowledged by the Framework for a proper action. Moreover, in order to notify the Framework about the changes that may occur unexpectedly, a control stereotype was depicted in the class diagram, named Environment Controller, which interprets these events and notifies the Framework. As for this project's scope, the Environment Interface and the Environment Controller are considered abstractions since their implementation have little impact on the research question asked.

Second, the Launcher is the central entity of the model, for it is responsible for controlling the main actions of the Framework and calling the other entities whenever needed, a somewhat similar approach to the proxy design pattern. It holds account on launching the implementation, by setting up important configurations for the execution of the OSGi Framework environment like installing essential bundles, adding remote repositories, and other tasks. It is also responsible for handling each bundle's life cycle, and listening for bundle state changes, once it uses methods such as *install*, *start*, *stop* and *uninstall* for the former and *setListener* for the latter. Besides, it manages the execution of the deployment plan with the methods *setGoal* and *replan*. Since it handles the main operations needed for the automated deployment of bundles, this entity is considered the core of the implementation.

Next, the planner entity is fundamentally based on the Deployment Plan Algorithm developed in Goal-D, which revolves around providing a deployment plan based on

a set of context conditions available in the environment at a given moment. Thus, to achieve this, this entity keeps a list of scenario constraints as an attribute that holds the environment context conditions mentioned, and makes use of a method called *getPlan* to provide Launcher with the desired plan.

Lastly, the OBR agent entity is accountable for interfacing the repository for fetching bundle artifacts and deploying them into the environment. It holds two specializations, the localRepoAdmin entity and the remoteRepoAdmin, which have the same job but act in different locations. On the one hand, the localRepoAdmin search for bundles in the local machine, by looking at the Eclipse plugin folder, where all of the default bundles are stored. This entity is primarily intended to install the configuration essential bundles, for they are more likely to be found there, and to save computational resources, once it does not need to perform the search in a remote location when the bundle may be found locally. On the other hand, the remoteRepoAdmin looks for bundles in remote locations, namely, remote OSGi Bundle repositories. Different repositories can be added through the specific location of the XML descriptor file by the *addRepository* method, and a bundle can be searched in any of these added repositories by the *search4BundleRemotely* method.

In conclusion, these four entities, the Environment Interface, the Launcher, the planner and the OBR agent, are responsible for the entire process of deployment planning, and acquiring and wiring the bundles, in other words, accomplishing the Goal-D's online activities.

4.2.2 Automated Deployment Planning

The automated deployment planning, said to be the focus of the online phase, relies on the repository created containing the bundles developed during the offline phase. A set of goals to be achieved are provided, or, in OSGi terms, a set of services are requested, and the target environment computes a deployment plan by using the set of resources available at a given moment as context conditions to pick the branches of the goal model tree that can fulfill the request.

As the deployment planning algorithm is already developed by the Goal-D's project, the framework presented here is only responsible for providing the information related to the set of goals and the context conditions of the environment, so that the plan can be generated. In order to stick to the scope of this project, the implementation developed abstracts away the Environment Interface entity that is responsible for gathering the information of the available computing resources, storing them in a list of the current context conditions, and notifying the framework. Moreover, the Environment Controller entity was also considered to be already existent, providing the implementation with a set of goals to be executed. The framework, thus, stores this information and calls the Deployment Planning algorithm with a plan request, giving the set of context conditions and the set of goals to be met.

The deployment plan algorithm was simply inserted into the OSGi Framework implemented as a solution for this project, as it has already been concretely developed

by Rodrigues in his work. It is used as a module that holds information about bundles registered in the repository and is in charge of generating a plan upon request for a given goal.

The deployment planning activity is performed by our framework through two different processes. The first, new goal registration process, provides the framework with the user desired goals to be achieved, while the second, re-planning process, is related with the autonomous characteristic of our implementation for it enables the adaptation of the system according to the environment context.

New Goal Registration Process

The first process is the New Goal Registration Process. This procedure aims to acknowledge that a new goal is defined and generates a deployment plan that fully and correctly satisfies the goal. It is portrayed by the Sequence diagram of Figure 4.8.

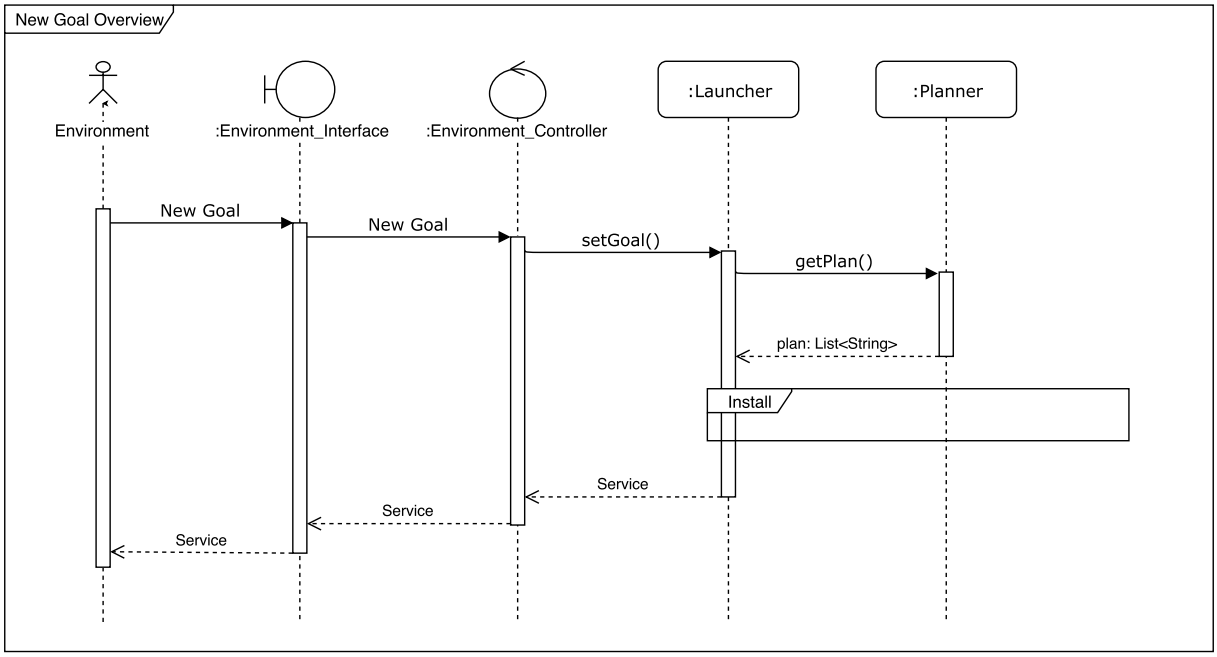


Figura 4.8: New Goal Registration Process' Sequence Diagram

The process starts when the Environment Interface entity perceives a new goal registration from the Environment, which could be an end user requesting for nearby Filling Stations or a sensor indicating low fuel. Such goal registration is interpreted by the Environment Controller and passed as parameter by the *setGoal* method to the Launcher. The Launcher receives the goal as a string, and asks the Planner entity for a new plan. Then, the *getPlan* method is called, which is responsible for generating a deployment plan given a set of environment constraints. This method returns a list of strings, which refers to the set of names of the bundles that must be deployed so that the Environment's goal can be achieved. At last, for each name on list, the Launcher performs the Installation procedure of the deployment execution activity. Upon success

on fetching and binding all of the bundles on the Deployment Plan, the service requested is provided for the Environment.

Re-planning Process

Next, the re-planning operation is key for providing the autonomous characteristic for the developed implementation. When the environment stops providing some computational feature, like the cellphone running out of battery, thus, interrupting the Internet connection, the set of bundles given by the Planner's Deployment plan at a certain moment are not able to provide the service anymore. Thereby, a new plan must be generated in order to the environment continue to be served. This functionality can be achieved in two different ways: the first is depicted in the sequence diagram in Figure 4.9, the other is through the Launcher listener.

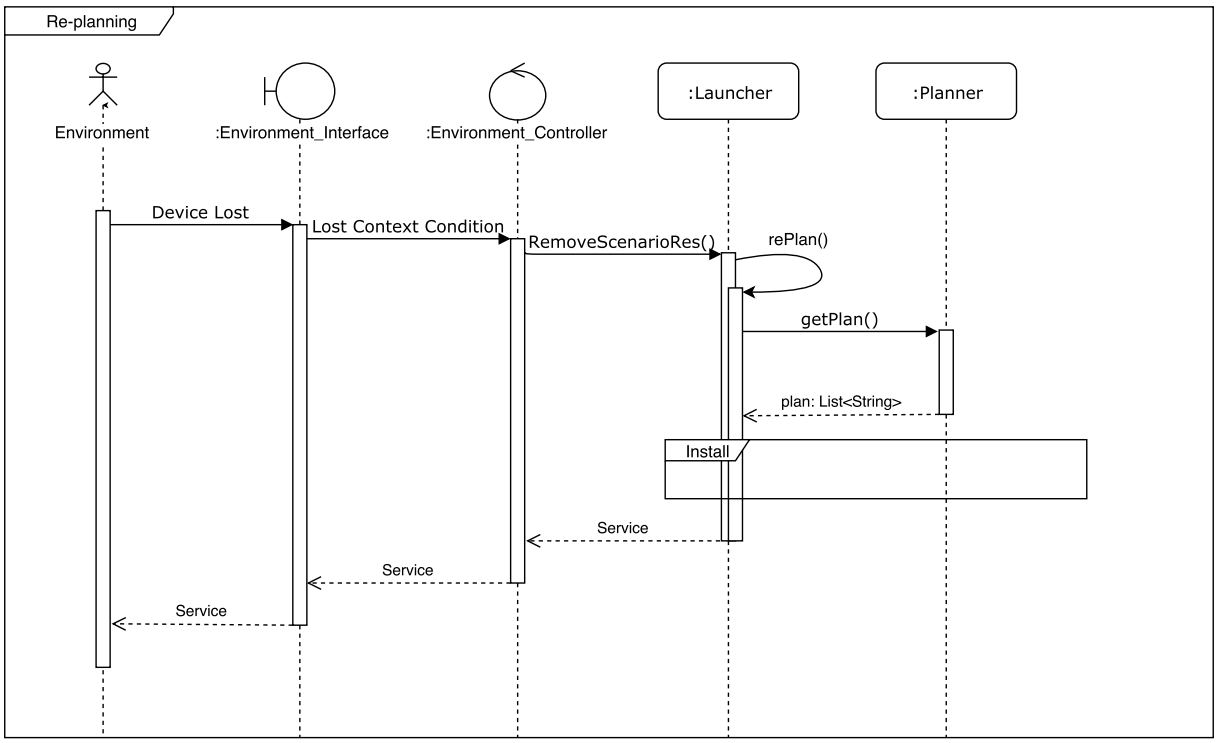


Figura 4.9: Re-planning Process' Sequence Diagram

The figure describes a process that starts when the Environment Interface identifies that a given resource is lost. The Environment Controller elucidates this fact and informs the Launcher, by calling the *removeScenarioRes*. The Launcher, in turn, removes that resource from the list of registered contexts. Afterwards, the Launcher asks the Planner to generate a new deployment plan through the *getPlan* method. The Planner, then, making use of the new set of context conditions, analyze the contents of the repository in order to provide a new Deployment Plan so that the Framework may continue providing the environment desired service. A new list of strings related to the names of the bundles is returned, and the Launcher proceeds by comparing the list of bundles that are already installed and the new list, installing only the bundles that are not on the new plan,

by performing the Installation process described in Figure 4.10. In case of successfully fetching and binding the new set of bundles, the service provision is guaranteed and our Framework returns to provide it.

Alternatively, the re-planning process can be carried out by using the listener attribute of the Launcher. This attribute is liable for listening for changes in the life cycle state of every bundle active in the environment. Once a bundle stops, transitioning from the ACTIVE state to any other state, the listener acknowledges it and calls the *replan* method, which works as described above.

This is one of the most remarkable features our proposed framework, since it enables the system to adapt to context changes, without requiring human intervention. It is also possible, depending on the decoupling level of the bundles being utilized, for the application to recover needing not to stop providing the service to the environment, as this is one of the OSGi technology features.

4.2.3 Deployment Execution

In the Deployment Execution activity, the plan provided by the Deployment Planning algorithm is executed in order to fetch the bundles from the remote repository and bind them, providing the requested service. The plan is composed of a list of bundle names, each matching the *Bundle-SymbolicName* header of unique bundle disposed in the repository.

After receiving from the Planner the deployment plan, the OSGi framework is responsible for looping through the list of bundle names and performing the installation process, depicted in Figure 4.10. For each bundle in the list, first, the Framework checks if it is already installed, then searches for a equivalent bundle in the local machine. Later, in case of package not found, the search is performed in remote repositories. Once the bundle is found, it is acquired and installed, and an attempt of resolving it is done, in other words, the Framework checks whether all of its dependencies are available. A dependency is said to be available if the bundle that it refers to is in the ACTIVE state.

The details of the installation process are as follows: this procedure starts when the method *install* is called. It performs a search in two basic locations: locally and remotely. The initial search is done locally aiming for performance and a better use of computational resources and is executed by the LocalRepoAdm entity when the *search4BundleLocally* method is called. The method receives as a parameter a string that refers to the bundle's name, which is checked, at first, to see if it ends with the *".jar"* extension, indicating that the name given is already the bundle name that should be searched for. The method then, conforms the string to the specific format used in the search which is *"file:"* followed by the path in the local machine and the bundle name, ending with the extension *".jar"*. Otherwise, if the name lacks the extension, it probably is not complete, missing the version or the timestamp, for instance. In this case, the LocalRepoAdm retrieves all of the files in the default bundle folder and tries to find a bundle that contains the parameter. Once found, the method returns a string in the format described. In case of receiving the bundle file name in the correct format, the

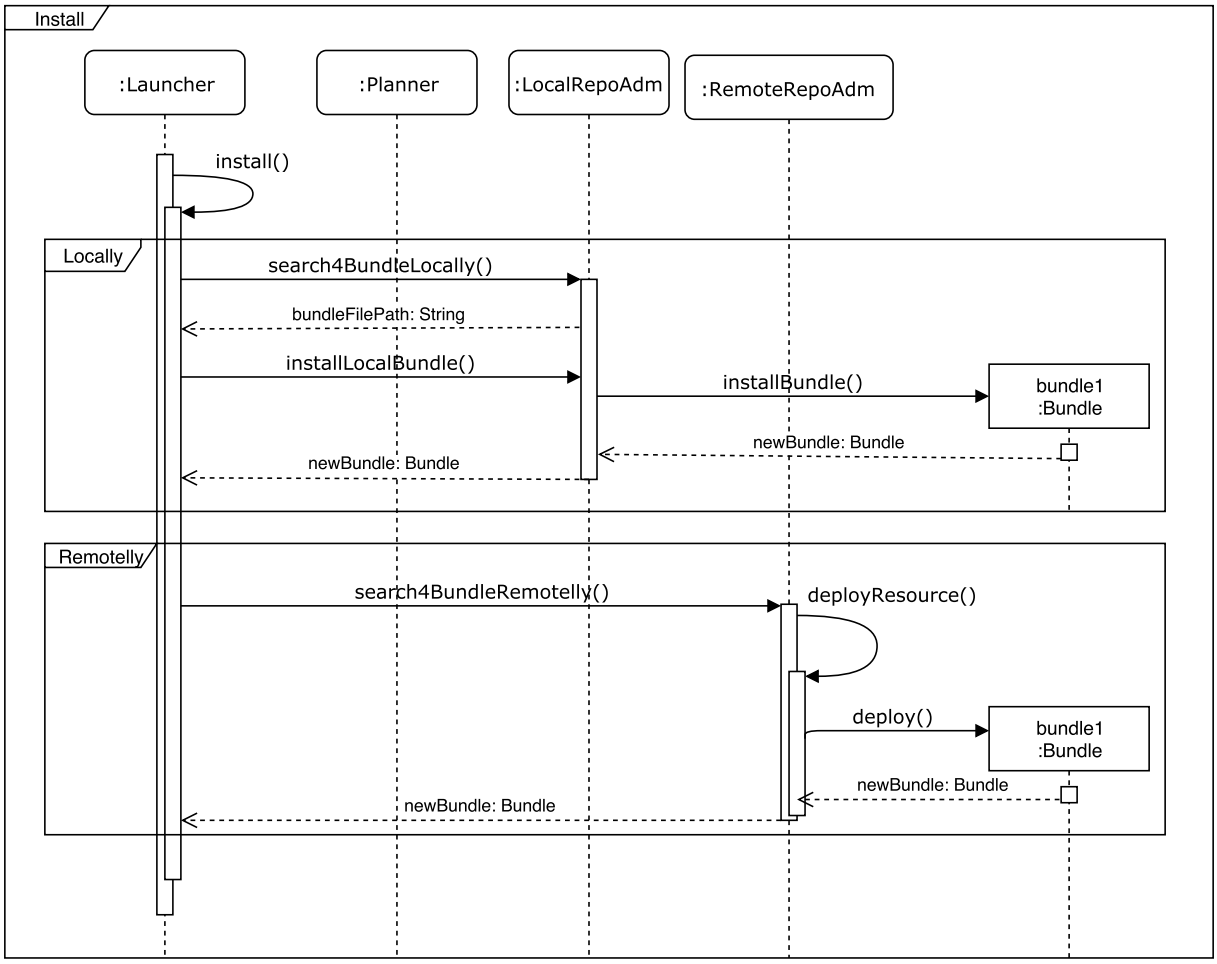


Figura 4.10: Installation Process' Sequence Diagram

Launcher calls another method from the LocalRepoAdm, the *installLocalBundle*, which is responsible for the installation of the given bundle. To begin with, it makes sure that the bundle is not already installed, and then uses the OSGi's Framework *Bundlecontext* object to install the bundle in the environment. In case of success, a bundle object that alludes to the newly installed bundle is returned.

However, if the local search fails, the Launcher tries a different approach by invoking the *search4BundleRemotelly* method, in which the RemoteRepoAdm entity searches performed in a remote repository. Parallel to the *search4BundleLocally* method, this method receives the bundle's name as a parameter and checks its format beforehand. If the format is correct, it conforms it to a requirement LDAP string, as the Requirement-Capability model indicates, for a correct search on the repository. Afterwards, the filter created is used to query the remote repository for matching bundles, and the most fit is chosen to be deployed based on pre-chosen quality criteria. The deployment is done by the resolving process, where dependencies are elicited and fetched along with the requested bundle. Upon success, the method returns and the Launcher acquires the Bundle object.

As for this stage of the project, it is noteworthy the fact that the "most fit"means the bundle with the highest version, being this the only quality criteria. In future stage, it

is intended to enhance the usage of the Requirement-Capability filter , in order to enable it build a more complex set of LDAP filters that takes into account quality aspects for a better search.

Bundles are fetched in remote search by using the OBR agent developed by the Apache Felix OSGi Bundle Repository, an implementation of the OBR format and R5 format [4] [19], explained in further detail in section 2.3.4. Upon request, this agent is responsible for reading the XML file, which contains the information about the stored bundles, and querying for a specific bundle through a LDAP filter expression. When it finds a match, the corresponding bundle is retrieved and installed in the environment.

Afterwards, an internal OSGi process called *resolve* begins, by checking for the availability of its dependencies and binding the requirements of the given bundle to the capabilities of its reliances, creating the so called "Wires", in OSGi terms. Then, if the resolving process succeeds, the bundle goes to the RESOLVED state and the Framework activates it, making the service it implements available for other bundles to use. Otherwise stated, if all of the subgoals of a refinement are accomplished, a given goal from the goal model tree is satisfied, allowing for goals in upper levels to have their dependencies fulfilled. However, when a lazy activation policy is set, a bundle may stay on the STARTING state until one of its methods is called, thereafter transiting to the ACTIVE state. After all, when all of the bundles on the list are fetched and binded, and their respective services are being provided, the root goal is satisfied, thus, delivering the desired functionality.

Moreover, the implementation also counts on a very important feature for adding the autonomous characteristic to the solution. A bundle listener was developed and inserted into the framework in order to catch bundle transition events, so, whenever a bundle changes from one state to another, the framework is notified and is capable of automatically handling the situation if needed, thus, without relying on human intervention. In a specific way, whenever some context condition changes in the environment, specially in case of sudden unavailability, causing the bundle that makes use of it to stop, the listener is acknowledged and seeks for a turnaround solution, most likely re-planning with the new set of context resources.

As an example, suppose a scenario in the Filling Station Advisor Application where the GPS and the Antenna Triangulation service are available in the environment, both able to fulfill the *G1: Get position*, but with the GPS active at a given moment. If the GPS signal is lost, and the bundle that implements P1 stops, the listener gets a notification and makes a new plan, now using the Antenna Triangulation service as a means to meet G1's dependencies. Since this situation was overcame without requiring any actions from the user, it was said to be autonomously handled.

In this section, it was presented how the online activities can be implemented with the usage of the OSGi technology. Initially, the automated deployment planning, which is responsible for providing our framework with a deployment plan upon user request or sudden context changes. Lastly, the deployment execution, which outlines the installation process on each bundle of the deployment plan, by fetching them from a repository and binding them in order to fulfill the desired goal.

In conclusion, in this chapter we explained our implementation of GoalD in OSGi.

This was accomplished through the description of each GoalD activity in light of the OSGi technology and our implemented framework, the offline and online activities. Initially, the offline activities consists of (1) goal modeling, which is performed in an early stage of the software development and outputs the Goal model that outlines the requirements of the system; (2) mapping components, which relates goals to OSGi services and components to Java classes considering the goal refinements; and (3) packaging artifacts, which is responsible for packaging components and metadata into OSGi bundles and registering them into repositories. Later, the online activities are (1) automated deployment planning, which performs the registration of user goals and the re-planning, in case of context changes; and (2) deployment execution, which is responsible for executing the installation process, which fetches bundles from repositories and bind them together, providing the requested service.

Capítulo 5

Evaluation

In order to evaluate the efficiency of the Framework OSGi implemented for the online activity, a series of tests were performed considering the Filling Station Advisor application, the motivating example of this project. Thereby, This section is intended to present the results of these experiments and provide an explanation about the outcome. Two tests were made: one that correlates the quantity of bundles in the repository over the execution time, and another that measures the time given the size of the bundles in the repository.

The tests were performed in an Asus computer, running a 64-bit Windows 10 Home Edition operational system with 4GB of installed RAM, along with an Intel(R) Core(TM) i3-3217U CPU processor at 1.80 GHz. They were accomplished by using the bundles created in the Eclipse Neon IDE, in version 4.6.3, through the implementation process described throughout this project, starting from the Goal Model from figure 3.1, going through the mapping of goals to Java classes and service declarations, until their package into bundles and storage in a remote repository. This process resulted in the creation of 31 bundles of 3 kilobytes, approximately, each containing only plain text as a means of providing a functional example for testing.

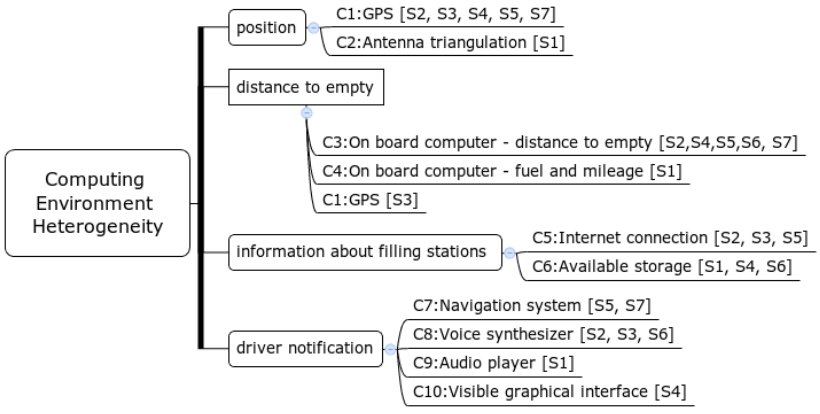


Figura 5.1: Computing Environment Evaluation Scenarios [24]

Each of the tests were divided into two experiments targeting different functionalities of the system. The first evaluates the time spent in order to set a new goal, and finish its execution, while the second computes the time taken to re-plan when the scenario resources vary. They were executed 100 times and the average was taken for analysis. For the first, the same scenarios used in the evaluation of the GoalD methodology were applied here, and are displayed in Table 5.1. Figure 5.1 depicts contexts *c1* to *c10* as they were defined by Rodrigues in his project and is placed here for reference.

Scenario	Context Condition
S1	c2, c4, c6, c9
S2	c1, c3, c5, c8
S3	c1, c5, c8
S4	c1, c3, c6, c10
S5	c1, c3, c5, c7
S6	c3, c6, c8
S7	c1, c3, c7

Tabela 5.1: Test Case Scenarios for Executing a New Plan

For the second, we created five scenarios that happen one after the other, starting from S1.1 until S1.5, in order to force the system to re-plan. They represent small steps taken as if we were transitioning from scenario S1 to S2. Table 5.2 below depicts this transition and the test cases for the re-planning process.

Scenario	Context Condition
S1.1	c2, c4, c6, c9
S1.2	c1, c3, c5, c8
S1.3	c1, c5, c8
S1.4	c1, c3, c6, c10
S1.5	c1, c3, c5, c7

Tabela 5.2: Test Case Scenarios for Re-planning

The first part of the initial test relates to the time spent in order to set a new plan, obtain the deployment plan and install all of the bundles from the plan, namely, the New Goal Registration process. Initially, the test was run with the 31 bundles created by the GoalD approach. Afterwards, each bundle was replicated as of showing the variability that is intended for the target environment, starting with the duplication of the initial number, followed by the multiplication by five in the initial number. It is noteworthy that the root goal was not multiplied, since it cannot vary for it the bundle name correspondent has to be specified by the environment when registering a new goal. So, the tests were performed with 31, 91 and 151 bundles inside the repository.

As it is depicted in Figure 5.2, little changed in regard to time, when new bundles were added to the repository. The test cases *S6* and *S7* successfully pointed errors when

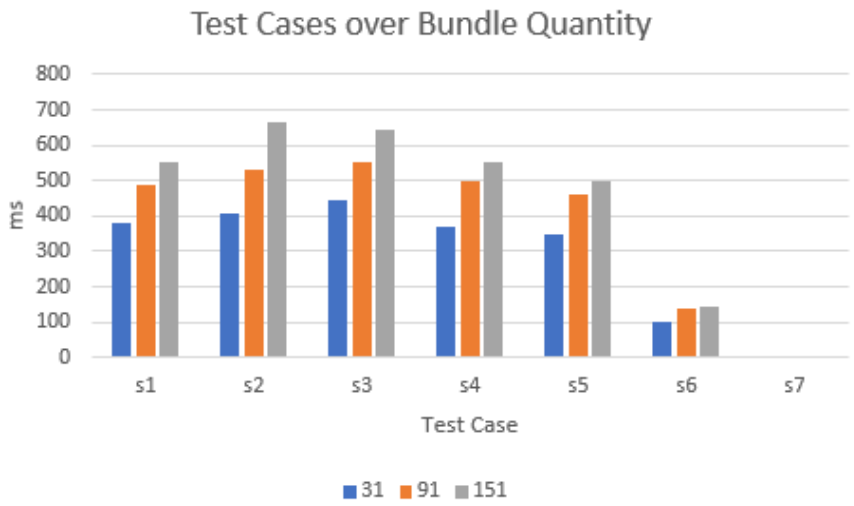


Figura 5.2: Quantity of Bundles over Time for New Goals

generating the Deployment Plan for not enough resources were provided, resulting in a small amount of time to finish the execution of the process. In the other cases, the Filling Station Advisor was executed correctly, and the quantity of bundles in the repository seems to have little influence on the time elapsed, since even with five times more bundles the total time increased by a small amount. This probably happens because all of the processing is client-side, done through the reading of a simple XML file for finding the matching bundle, which does not require much computation.

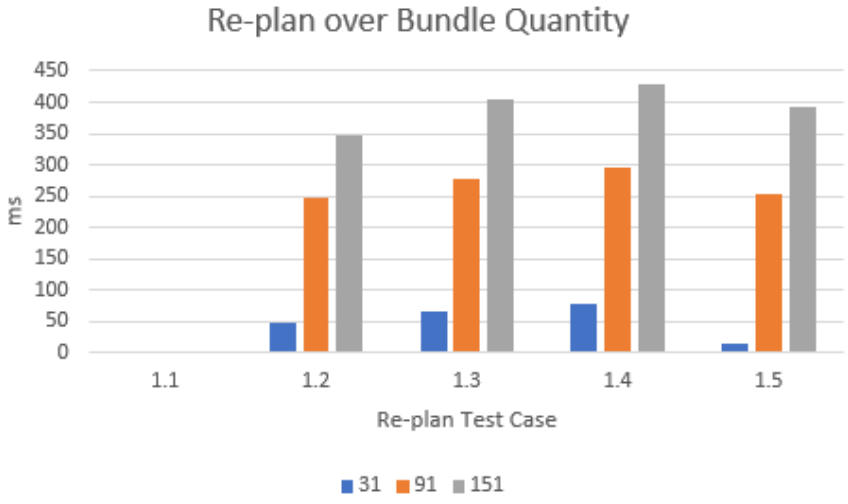


Figura 5.3: Quantity of Bundles over Time when Re-planning

Differently, when looking at the results of the second experiment of the first test in Figure 5.3, it is possible to infer that the time taken to re-plan has grown steadily given a larger set of bundles. Test case worked successfully in *S1.1*, for not enough resources were presented for reactivating the Filling Station Advisor application. But it is possible to see an exponential growth in time when more bundles were added, which can relate to the

time taken to uninstall unused bundles by the new plan, that may have to disable entire branches of the goal planning and install others completely different, adding a significant amount of time to the final result.

The second test performed intends to measure the efficiency of the same processes but now with varying bundle's sizes. As mentioned earlier, the bundles were originally created with a size near to 3K, since their packages only consisted of the metadata, the Java classes and the service declarations. Next, an image was packaged inside each bundle, increasing their size to 2 Mb. Next, a bigger file was packaged, adding up to 10 Mb in each bundle. Later, 20Mb and, lastly, the bundles got to the size of 200 Mb in order to have a better understanding of the curve of efficiency.

Figura 5.4: Size of bundles over Time for Test Cases

The first experiment of the second test evaluated the time taken by the New Goal Registration process, considering an increased bundle size. In order to better display the results, a logarithmic scale was adopted in Chart **??**. It depicts, in general, that as the bundles grow bigger, more time is spent in the referred process, since the test cases *S1* to *S5* present a very similar curve that escalates quickly. The other two cases, *S6* and *S7* represent the two cases in which there is no generated plan.

The second experiment of the second test, portrayed in Figure **??**, measures the time elapsed when performing the re-planing, while taking into account the increasing size of the bundles inside the repository. Just like in the earlier experiment, the 31 initial bundles had their sizes enlarged in order to correctly execute this test, and the chart shows a logarithmic scale to better display the outcome. The results were very similar to the previous experiment: there can be seen a quick growth in time due to the size difference, by looking at the curves of the cases when the planning was successful. The difference between them might be due to the magnitude of the variation between the test

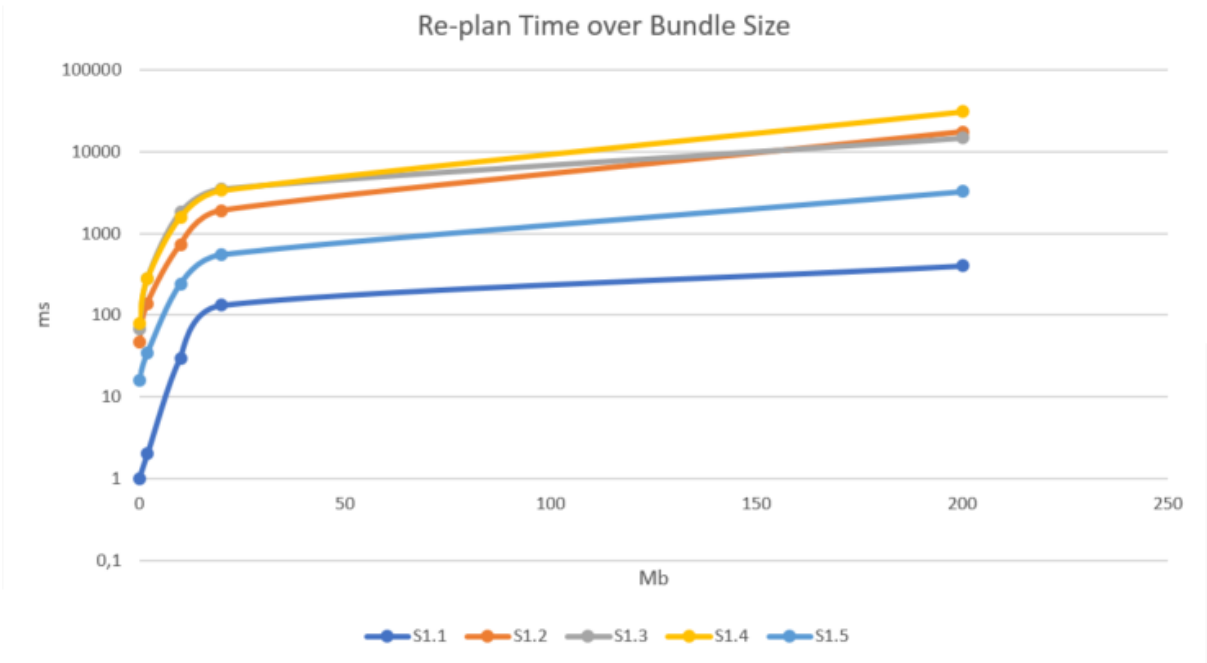


Figura 5.5: Size of bundles over Time for Re-plan Cases

cases. Moreover, the test case that differs from the others relates to failing plans when not enough resources are available.

After a careful analysis of the results presented in this Chapter, it is possible to conclude that the first test presents this implementation as a very scalable solution, since, even with the quantity of bundles multiplied by five, just a small variation of time was detected. This is important once the Framework may search in large of repositories in order to find a satisfiable bundle, even in federations of repositories, where thousands of bundles may be available for search. So, if the solution presented here supports large quantities of bundles, it will most likely suit the intended settings of a real world application. A further analysis of the Re-planning process should be done, though, in order to correctly understand the reasons that made it take so long.

Nevertheless, by looking at the outcome of the second test, the solution here presented is shown not to be as efficient as it should. Even though it suits the scope of the project of checking if the OSGi is a viable solution to implement GoalD, the time spent in order to process the bundles, by fetching and binding them, is not ideal. This happens because the environment in which the Framework developed is intended to run may contain resources with the most varied sizes, so a better performance is needed in order to applying it in a real-world scenario.

Capítulo 6

Conclusion

In this project, the OSGi technology was proposed as a means to concretely assert the feasibility of the GoalD methodology. In this perspective, all of the activities delineated by GoalD were mapped into concrete development patterns, by observing a correspondence between GoalD concepts and OSGi main ideas, and through the implementation of a framework that allows the process of fetching and binding of artifacts in run-time.

It was shown that OSGi fulfills the three challenges outlined in GoalD dissertation: heterogeneity, uncertainty at design-time and autonomous deployment. For the first challenge, heterogeneity, it was explained that OSGi targets environments with highly diverse kinds of devices due to the notion of resources, which are responsible for providing any set of computational features packaged into small units of modularity. For the second challenge, uncertainty at design-time, OSGi proved to be a match for GoalD, since it allows for a run-time selection of available implementations. This is achieved through the notion of a service, in which providers are binded with clients through very specific contracts, created beforehand at design-time. The last challenge, autonomous deployment, is achieved by OSGi through the same notion of services, which may come and go, and are binded in a programmatic manner, thus, not requiring human intervention. Because of these three facts, OSGi was found to be a match to the integration with GoalD.

In order to provide GoalD with a fully functional implementation in terms of the OSGi technology, we used GoalD's division of online and offline activities for describing the reification of each GoalD offline activity into steps used in the development of OSGi bundles along with the development of a framework which accommodates the fetching and binding processes of the online activities.

Initially, the offline activities were translated into steps of OSGi bundle development. In the first offline activity, modeling requirements as goals, little was added by OSGi since it is a process done by the software architect when eliciting the system requirements. In the second offline activity, mapping components as goals, components were defined as Java classes and, according with the type of refinement, different patterns were used to create them. Goals were mapped into the OSGi services for they may rely on other services in order to be provided, just as goals in the refinement process. In the

third offline activity, packaging artifacts, this project implements the package of artifacts into OSGi bundles, since they store the same kind of information, namely classes, metadata and other resources. This step also considers a repository to which the bundles are registered. Such repositories are tightly related to OBRs, in which bundles are fetched by searching an XML file and retrieving them to the environment.

Later, for the online phase, a functional Framework for the autonomous generation and execution of deployment plans was developed. Our framework was developed in order to concretely implement the online GoalD activities in an autonomous manner, by relying on four main agents: an environment interface, for context detection; the Launcher, responsible for requesting plans and executing them by fetching remote repositories and binding bundles; the OBR, which stores bundles for later fetching; and the Planner, which relates to the generation of deployment plans when requested.

The two steps of the online phase were implemented as follows: first, the deployment planning was achieved by integrating the Deployment Planning algorithm developed by Rodrigues with our framework, for the generation of a deployment plan upon request. This step performs two main process, the new goal registration process, which is accountable for understanding new user's goals in order to provide a specific plan, and execute it; and the re-planning process, that listens to the environment and acts when a given context condition ceases to exist, thus, this process enables the autonomous generation of a new plan in order to reestablish the provision of a given goal, or service. Second, we further contribute to the deployment execution step of Goald by using our framework for searching OBRs to retrieve and bind bundles of a given deployment plan, through the installation process performed over each bundle on the plan.

When evaluating the Framework used in the online activities, we saw that it is a very scalable solution, for the New Goal Registration process have little impact when performing the search in larger sets of bundles, even though, the Re-planning process has a quick increase in time due to the increased computational complexity. Conversely, efficiency is not a plus for this solution, since the results show that for the bigger the bundles are in the set, the greater is the time spent on both new goal and re-plan processes, since more time is spent on acquiring the bundles from the repository.

After an exhaustive examination of the OSGi technology, it was possible to conclude that each of the activities of the GoalD along with its chief ideas can be mapped onto OSGi concepts. Besides, the Framework developed enables an autonomous deployment in highly heterogeneous environments, which are the intended background for the GoalD methodology. Therefore, as a conclusion, OSGi has shown to be an adequate solution for a concrete implementation of the GoalD's approach.

6.1 Future Work

As for an advance of the work described in this project, three possible enhancements can be proposed: the integration of the Planner's repository with the OSGi Bundle

Repository (OBR), the creation of a remote repository for a better performance evaluation and the development of a context listener.

The first, relates to the integration of the remote repository created with the help of Maven to the repository that the Planner entity reads for generating the deployment plan. Rodrigues' work, as a means to evaluate his solution, implemented the planner algorithm, which was broadly used in this project, specially on the online activities. And, to support it, he built his own version of an ideal repository that stores the component attributes required by the GoalD approach in a convenient way, accessing them through methods very tied to his specific implementation. Simply put, he does not make use of the Maven created XML file for consulting the repository data when planning the deployment of bundles. So, in order to keep the integrity of his work, little was altered from his implementation by the solution proposed here. This fact resulted in a small shortcoming: two different repositories were created and had to be managed separately. One of them keeps the information of the registered bundles in an XML file, deployed using Maven, and is used by the Implementation of this project for performing the fetching process, further explained in Subsection 4.2, while the other repository is held inside the GoalD's project and mirrors the remote repository, being updated manually. Although this fact may bring inconsistencies in large projects with a high number of bundles, this is not the case when it comes to the motivating example used here, the Filling Station Advisor application, which was built using a small number of bundles, making easy to overcome this situation. However, it is planned for a future stage of this work to integrate both repositories for a better consistency.

The second, refers to the creation of a remote repository for enhancing performance evaluation. In order to meet the objectives outlined by this project, it was enough to simply create another local repository in a different folder and see it as if it was placed in a remote location, once all of the experiments were only performed locally, even though this new repository is still referred as remote. However, in a future stage of this project, a remote repository will be created for a further analysis of the efficiency of the methodology. This can be done with the usage of a tool named Nexus Repository [27], which is built on top of the OSGi container Apache Karaf [7], and is able to proxy and cache remote OBRs, apply CRUD privileges for access control, and other features. Thus, providing opportunities to enhance efficiency [26].

The third and last enhancement that is planned for a future stage is the development of a context listener. As for the current phase, the Environment Interface and Environment Controller entities were abstracted away in order to shorten the scope of the project. They are responsible for listening to the environment and gathering information about the context conditions available, as well as notifying the Launcher entity about the come and go of devices in dynamic places. This choice was made since the manner in which our framework receives information about context is irrelevant to attest to the feasibility of the OSGi as a possible implementation of GoalD. But, in later phases, it is planned to develop this layer in order to perform more accurate tests on our framework's efficiency.

Referências

[1] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. A goal-based framework for contextual requirements modeling and analysis. *Requirements Engineering*, 15(4):439–458, 2010. 4, 5, 7, 18, 19

[2] OSGi Alliance. About us. `https://www.osgi.org/about-us/`, 1999. 8, 20, 21

[3] OSGi Alliance. OSGi Core Release 6. `osgi.org`, 2014. viii, 9, 10, 11, 12, 14, 20, 21, 25

[4] OSGi Alliance. OSGi Enterprise Release 6. `osgi.org`, 2015. 15, 36

[5] OSGi Alliance. Benefits of Using OSGi. `https://www.osgi.org/developer/benefits-of-using-osgi/`, 2017. 21

[6] OSGi Alliance. Bundle Headers Reference. `https://www.osgi.org/bundle-headers-reference/`, 2017. 25

[7] Apache. Apache Karaf. `http://karaf.apache.org/`, 2017. 45

[8] Joao Paulo Araujo. An OSGI Implementation for Autonomous Goal-Oriented Deployment: The project's code. `https://github.com/jcosta9/OSGi`, 2017. 29

[9] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, October 2010. 1

[10] Ray Augé. Using Requirements and Capabilities. `http://blog.osgi.org/2015/12/using-requirements-and-capabilities.html`, 2015. 14

[11] Genevieve Bell and Paul Dourish. Yesterday's tomorrows: notes on ubiquitous computing's dominant vision. *Personal and Ubiquitous Computing*, 11(2):133–143, February 2007. 1

[12] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004. 5, 18

[13] Ivica Crnkovic and Magnus Larsson. Component-based software engineering-new paradigm of software development. *Invited talk and report, MIPRO*, pages 523–524, 2001. 7

[14] Anne Dardenne, Axel Van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of computer programming*, 20(1-2):3–50, 1993. 4

[15] Scott A. DeLoach, Walamitien H. Oyenan, and Eric T. Matson. A capabilities-based model for adaptive organizations. *Autonomous Agents and Multi-Agent Systems*, 16(1):13–56, February 2008. 4

[16] Walid Gédéon. *OSGi and Apache Felix 3.0: beginner's guide ; build your very own OSGi applications using the flexible and powerful Felix Framework*. Learn by doing: less theory, more results. Packt Publ, Birmingham, 2010. OCLC: 753206534. 9, 10

[17] Richard S. Hall, Karl Pauls, Stuart McCullock, and David Savage. *OSGi in Action: Creating Modular Applications in Java*. Manning Publications, 2010. 10, 11, 13, 15

[18] Van Lamsweerde. Goal-oriented requirements engineering: a guided tour. in requirements engineering. *Fifth IEEE International Symposium*, pages 249–262, 2001. 4, 13

[19] Apache Maven. Apache Felix OSGi Bundle Repository (OBR). `http://felix.apache.org/documentation/subprojects/apache-felix-osgi-bundle-repository.html`, 2017. 36

[20] Apache Maven. Introduction to the Build Lifecycle. `https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html`, 2017. 16

[21] Apache Maven. Introduction to the POM. `https://maven.apache.org/guides/introduction/introduction-to-the-pom.html`, 2017. 17

[22] Apache Maven. Welcome to Apache Maven. `https://maven.apache.org/index.html`, 2017. 16

[23] Jeff McAffer, Paul VanderLei, and Simon Archer. *OSGi and Equinox: Creating highly modular Java systems*. Addison-Wesley Professional, 2010. viii, 11, 12, 13

[24] Gabriel Rodrigues. Autonomic goal-driven deployment in heterogeneous computing environments. 2012. viii, 1, 5, 6, 7, 8, 13, 18, 19, 20, 38

[25] Stephen D. Smaldone. *Improving the Performance, Availability, and Security of Data Access for Opportunistic Mobile Computing*. PhD thesis, Rutgers University, New Brunswick, NJ, USA, 2011. 1

[26] Sonatype. Nexus Pro: Support for OSGi Bundle Repositories (OBRs). `http://blog.sonatype.com/2009/07/nexus-pro-support-for-osgi-bundle-repositories/`, 2017. 45

[27] Sonatype. Nexus Repository. `https://www.sonatype.com/nexus-repository-sonatype`, 2017. 45

[28] Axel Van Lamsweerde. From system goals to software architecture. *Formal Methods for Software Architectures*, pages 25–43, 2003. 4, 5, 6, 7, 18

[29] Yijun Yu, Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Julio Leite. From goals to high-variability software design. *Foundations of Intelligent Systems*, pages 1–16, 2008. viii, 6, 7, 8, 9, 13, 18, 23