Master’s Thesis

in the international graduate program

Service Design and Engineering (SDE)
offered by

Aalto University
School of Science and Technology

Semi-Automatic Service Integration of
Telecom and Internet Services in a
Service Delivery Platform

submitted by

Irina Todoran, BSc

Supervisor (Aalto University):
Prof. Dr. Heikki Saikkonen
Instructor (Technical University of Munich):
MSc. Christian Menkens
Place:
Espoo, Finland
Date of Submission:
July 25th, 2011
Master's Thesis Abstract

The purpose of this study was to identify the most appropriate way to (semi-)automatically integrate external Internet and telecom services into a Service Delivery Platform (SDP) for a Telecommunications operator, thus making them available to the community of developers. Another aim was to show how the concept can be implemented in a service-oriented manner.

Both the literature review and design science methods were applied in this thesis. The literature review was conducted to identify and assess the existing service description languages for Representational State Transfer (REST) architectures, and automatic code generation alternatives. For this, the concept-centric approach was used. The design science focused on implementing a prototype for the automatic code generation service, which shows how the concept developed can be materialized. The artifact constructed consists of a use case based on the Google Language Application Programming Interface (API).

The literature review indicated that an Extensible Markup Language (XML)-based description meets the requirements for the service specifications on an SDP. Furthermore, the study revealed that an engine which uses the description as the data model and a template as input, processes the data, and outputs a Java file is the most suitable solution for the automatic source code generation. The template engine chosen to develop this was the Apache Velocity open source software project, and the automatically generated source code was packaged within an Open Services Gateway initiative framework (OSGi) bundle, which can be deployed on the SDP.

The principal conclusion drawn was that semi-automatic code generation can be achieved on an SDP by using a template-driven approach. This solution meets the requirements regarding the generality of the project, and works for services with an indefinite number of compulsory and optional parameters. Therefore, the data model can be customized for any RESTful service which exposes its interface, and service-oriented architecture design principles such as loose coupling, composability and reusability are enabled.
Acknowledgments

This thesis would not have been possible without the guidance and help of several individuals who contributed and extended their valuable assistance in the preparation and completion of this research study.

First and foremost, I am deeply grateful to my supervisor, Prof. Dr. Heikki Saikkonen, Head of the Department of Computer Science and Engineering, Aalto University School of Science and Technology, for his support and the opportunity to do my Master’s Thesis abroad.

I would like to express my warm and sincere thanks to my thesis instructor, Christian Menkens, Center for Digital Technology and Management, Munich, whose encouragement, guidance and continuous feedback enabled me to develop a thorough understanding of the subject and accomplish the goals of the thesis.

I thank Mika Helenius, Director of the Service Design and Engineering Master’s Program, Aalto University, for the numerous constructive discussions and his valuable support in academic matters and beyond during my Master’s studies in Helsinki.

I am grateful to Prof. Dr. Marjo Kauppinen, Prof. Dr. Matti Hämäläinen and Kari Hiekkanen, who had a significant impact on my professional development throughout the last two years.

I would also like to acknowledge my colleagues in the Service Design and Engineering Master’s Program and my friends for their permanent support, notwithstanding the distance.

Lastly, I thank my parents and my boyfriend Ralph, who have always believed in me, and wholeheartedly supported me in all my pursuits. This thesis is dedicated to them.

Munich, Germany
July 2011

Irina Todoran
## Contents

1. Introduction ........................................ 4  
   1.1. Background ........................................ 4  
   1.2. Motivation .......................................... 5  
   1.3. Research Question ................................. 6  
   1.4. Objectives ......................................... 6  
   1.5. Scope ............................................... 7  
   1.6. Research Methods ................................... 7  
   1.7. Outline ............................................. 8  

2. Related Work .......................................... 10  
   2.1. Technology Concepts ............................... 10  
      2.1.1. Representational State Transfer (REST) Services ............................. 11  
      2.1.2. Service Oriented Architectures (SOAs) .................................... 12  
      2.1.3. The Open Services Gateway initiative Framework (OSGi) .................. 15  
   2.2. Service Delivery Platforms - State of the Art .............................. 19  

   3.1. The Concept of the Semi-Automatic Code Generation ...................... 24  
   3.2. Service Description Languages .............................................. 26  
      3.2.1. Details of Service Description Languages .................................. 26  
      3.2.2. Comparative Study on Service Description Languages .................... 33  
      3.2.3. The Description Language Chosen for the Project ....................... 37  
   3.3. Code Generation .......................................... 38  
      3.3.1. General Overview on Automatic Source Code Generation .................... 38  
      3.3.2. Comparative Study on Code Generation Alternatives .................... 41  
      3.3.3. The Code Generation Option Chosen for the Project .................... 43  

4. Semi-Automatic Code Generation: The Implementation ....... 45  
   4.1. REST Services Calls ....................................... 46  
   4.2. The Definition of the Service Description Language ....................... 49  
   4.3. The Template Engine ....................................... 52  
   4.4. OSGi Integration ......................................... 56  

5. Discussion and Achieved Objectives .......................... 62
List of Figures

2.2. A sample REST PUT call [1] ............................................ 13
2.3. OSGi layered architecture [2] ........................................... 17
2.4. A sample OSGi MANIFEST.MF file .................................... 18
2.5. OSGi bundle life-cycle [3] ................................................. 19
2.6. The converged application architecture concept [4] ................. 22
3.1. The concept of the semi-automatic code generation .................... 25
3.2. The generative domain model, adapted after [5] ......................... 39
3.3. The template-driven mechanism ......................................... 40
4.1. The technical design .................................................... 46
4.2. Calling a REST service with Simple HTTP Request .................. 48
4.3. Calling a REST service with GlassFish Jersey ....................... 49
4.4. The XML Schema document for the SDL defined ...................... 50
4.5. An example of a service description .................................... 51
4.6. The template engine - Code sample. Part I .......................... 53
4.7. The template engine - Code sample. Part II ......................... 54
4.8. The template for the translate service provider ....................... 55
4.9. The translate service interface ........................................ 57
4.10. The implementation for the translate service provider ............... 58
4.11. The metadata.xml file for the translate service provider .......... 58
4.12. The implementation for the translate service client ................ 60
4.13. The metadata.xml file for the translate service client ............. 61
A.1. POM file for the service - Template .................................. 65
A.2. Service interface - Template ........................................ 66
A.3. Metadata for the service provider - Template ....................... 66
A.4. POM file for the service provider - Template ....................... 67
A.5. Service client - Template ............................................. 68
A.6. Metadata for the service client - Template ......................... 69
A.7. POM file for the service client - Template ......................... 70
A.8. The pom.xml file for the translate service ......................... 71
A.9. The pom.xml file for the translate service provider ............... 72
A.10. The pom.xml file for the translate service client ................. 73
API: Application Programming Interface
BSD: Berkley Software Distribution
CRUD: Create, Read, Update, Delete
CSP: Communication Service Provider
DSL: Domain Specific Language
GSM: Global System for Mobile Communications
GUI: Graphical User Interface
HATEOS: Hypermedia As The Engine Of Application State
hRESTS: HTML for RESTful services
HTML: HyperText Markup Language
HTTP: Hypertext Transfer Protocol
IDL: Interface Description Language
IMS: IP Multimedia Subsystem
IP: Internet Protocol
iPOJO: injected POJO
IT: Information Technology
JAR: Java ARchive
Java VM: Java Virtual Machine
JPEG: Joint Photographic Experts Group
JSON: JavaScript Object Notation
MDA: Model-Driven Architecture
NGN: Next Generation Network
NSDL: Norm’s Service Description Language
OASIS: Organization for the Advancement of Structured Information Standards
OMA: Open Mobile Alliance
OPEX: OPerational EXpenditure
OSE: OMA Service Environment
OSGi: Open Services Gateway initiative
OWL: Web Ontology Language
POJO: Plain Old Java Object
RELAX NG: REgular LAnguage for XML Next Generation
RESEDEL: REstful SErvices DEscription Language
REST: REpresentational State Transfer
RIDDL: RESTful Interface Definition and Declaration Language
SaaS: Software as a Service
SAWSDL: Semantic Annotations for WSDL
SAX: Simple API for XML
SDE: Service Delivery Environment
SDF: Service Delivery Framework
SDL: Service Description Language
SDP: Service Delivery Platform
SIP: Session Initiation Protocol
SMEX: Simple Message Exchange
SMEX-D: Simple Message Exchange Descriptor
SOA: Service-Oriented Architecture
SOAP: Simple Object Access Protocol
UDDI: Universal Description, Discovery and Integration
UML: Unified Modeling Language
UMTS: Universal Mobile Telecommunications System
URI: Uniform Resource Identifier
URL: Uniform Resource Locator
USDL: Unified Service Description Language
VTL: Velocity Template Language
W3C: World Wide Web Consortium
WADL: Web Application Description Language
WDL: Web Description Language
WRDL: Web Resource Description Language
WS-*: Web Services
WSDL: Web Services Description Language
WWW: World Wide Web
XHTML: eXtensible HyperText Markup Language
XML: Extensible Markup Language
XSD: XML Schema Definition
1

Introduction

1.1. Background

Digital Services have become a significant part of humans’ daily lives throughout the recent years, and the demand for increasingly sophisticated solutions has reached levels unknown before. In this context, traditional business models evolve, technological advances bring new opportunities for executives to raise enterprise revenues, and users become more critical and exigent.

Furthermore, the economics of online delivery change as Cloud Computing in general, and Software as a Service (SaaS) as a particular software delivery model, gains wider acceptance [6]. This has an important impact on service providers, software developers, users, advertisers and intermediaries, and companies need to adjust their internal capabilities related to service delivery in order to retain their roles in the market [7].

An additional factor that leads to this change is the maturity level reached by the Application Programming Interfaces (APIs) for services on the World Wide Web (WWW). These enable the easy construction of mash-ups and therefore extremely rapid service design.

Among the industries greatly affected by the digital services revolution [8] is the telecommunications industry. In the competitive Telecom market [9], service providers need a fast and flexible way to provision and profit from the advantages of the new technologies and opportunities available, bringing together Internet services and high-margin, bandwidth-intensive broadband services [10]. Moreover, end-users become increasingly demanding in their requirements for customized, easy to use applications [9], which assist them in their everyday activities. In order to answer these needs, the main idea is that telecommunication operators put at developers’ disposal gateways providing high level Web Services or REpresentational State Transfer (REST)-based APIs. On these interfaces, software developers can combine existing services to build new applications. Therefore, the number of new available services will increase proportionally with the ease of creating them on the given platform, bringing significant benefits to the
telecommunications infrastructure provider[11], and further on to the end-users.

This concept referred to as a Service Delivery Platform (SDP) or sometimes Service Delivery Environment (SDE), is often regarded as an innovative paradigm which enables the facile composition of various services to create new ones and deliver new value[12]. Consequently, there have been numerous attempts of both academic researchers[4, 9, 13–15] and industry practitioners[1, 10, 16, 17] to describe the architecture and working mechanism of the ideal service delivery platform. Due to the quasi standard nature of external service interfaces and descriptions, there is no standard way to import these services easily and efficiently into an SDP infrastructure[4], and the existing attempts are rather divergent. However, their common goals are: rapid service creation, deployment and execution[11].

1.2. Motivation

Whereas the concept of a service delivery platform has already been implemented by companies such as Apple[18] or Google[19], providing developers worldwide with the necessary resources to build and deliver new services right on the platforms, the situation of the telecommunication operators is rather disappointing. The Telecom industry needs to rapidly change in order to remain competitive on the market, since it is currently challenged by exclusively Internet service providers[20, 21]. The systems which have been closed and proprietary for a long time need to shift towards a more open infrastructure in order to work together with new services and tools available on the Internet, such as Web (2.0) services and applications.

As it results from previous studies, a gap between application developer communities and the Telecom industry was identified[4, 9, 13–15]. The developers are willing to work with telecommunication services, but the problem is they do not have access to the specific resources. In order to address this issue, the aim of this project is to develop such a service delivery platform for a Telecom service provider, which will bring together both telecommunication and Internet services, and put the necessary resources at developers’ disposal.

In addition, in order to meet the goals commonly formulated by academia and industry (rapid service creation, deployment and execution[11]), it is apparent that the working mechanism of the Platform needs a certain degree of automation. Having formulated the problem, in the present Master’s Thesis document, we are presenting a solution for the semi-automatic service integration of telecom and Internet services in a service delivery platform. This approach will not only reduce the time and human resources needed for the generation of new services, but it will also reduce the OPerational EXpenditure (OPEX) by enabling process automation on the SDP[10].

1.3. Research Question

Our main research question is: “What is the most appropriate way to semi-automatically integrate external Internet and telecom services in a service delivery platform for a Telecom operator, and make them available to developers?”

This main question will be answered by addressing the following sub-questions:
1. What is the most suitable Service Description Language (SDL) for defining the external Internet and telecom services which are integrated into the SDP?
2. How can compilable (Java) code be automatically generated from the service descriptions?
3. How can the result of the automatic code generation service be integrated in the SDP with the Open Services Gateway initiative (OSGi) framework?
4. What are the factors that define the generality of the project? What is the degree of generality that can be achieved for describing services and the automatic code generation?

1.4. Objectives

Having set the research question which needs to be answered, the objective of the study is to clarify how external Internet and telecom services can be semi-automatically integrated into a service delivery platform for a Telecom operator.

In this respect, we are defining the sub-objectives which form the main research objective, based on the questions presented in the previous section.

Objectives of research question 1:

a) Conduct a comparative study on different service description languages
b) Based on the literature review, define the appropriate service description language to be used in this project

Objectives of research question 2:

a) Conduct a comparative study on different alternatives for automatic code generation
b) Based on the study, choose the most appropriate option for this project

Objectives of research question 3:

a) Investigate how the new service created can be integrated in the service delivery platform with the Open Services Gateway initiative framework
b) Create a bundle which encapsulates the new service; this is based on the automatically generated source code, metadata and XML instruction files

Objectives of research question 4:

a) Define the factors that describe the generality of the project
b) Assess the degree of generality for the solution found
c) Explain the constraints and limitations
1.5. Scope of the Research

Our research work starts from the premise that the service delivery platform is implemented; therefore, the way the infrastructure of the project is built is out of scope. The service designed is to be integrated within an existing structure, as a stand-alone module (bundle).

In addition, the scope of this thesis is to focus on RESTful services and architectures, and protocols such as the Simple Object Access Protocol (SOAP) or the Session Initiation Protocol (SIP) are not considered in the comparisons made.

Moreover, as far as the automatic code generation method is concerned, the template-driven approach is the scope of the present research. Other methods using, for instance, model-driven architectures (MDAs) or semantic ontologies are beyond our scope.

Finally, the reference programming language used is Java, due to its perfect compatibility with OSGi and other significant features, such as portability and architecture-neutrality.

1.6. Research Methods

In order to fulfill the objectives of the research, the methods which were chosen are the literature review [22, 23] and the design science [23–25].

Firstly, a literature review was conducted to build a foundation for the research carried out. This assisted in collecting and structuring the information which exists in the fields of service description languages and automatic code generation. For this, we took the concept-centric approach [22], as opposed to the author-centric approach, since the main aim was to find different alternatives for the goal of the project and compare these, in order to reach the optimal solution.

The references for this thesis range from books, journal articles and conference papers, online articles, PhD and Master’s Theses and white papers to technical specifications provided by the Open Mobile Alliance (OMA), Apache, the OSGi Alliance.

The technical specifications documents were chosen in order to ensure an accurate representation, according to the standards. The journal and conference papers were selected based on keywords search on the engines of the portals of the publishers and manual search in Journal Libraries and Conference Proceedings. The keywords used included: ”service delivery platform”, ”service description language”, ”OSGi”, ”automatic code generation”, ”REST services”, ”RESTful SOA”, ”generative programming”, ”descriptions for REST services”.

Secondly, the design science method was used [23–25], consisting of a generate/test cycle. Various design alternatives were generated, which were then tested against the project requirements and constraints. This process was iterated several times, until the
initially formulated requirements were met, and the final artefact was created. This is an artefact “as situated implementation” [25], which means it is an actual construct, an implementation of a need, which can be further abstracted to the level of a model. Therefore, it can also be considered new “Knowledge as Operational Principle” [25].

1.7. Outline

This section gives an overview of the structure of the Master’s Thesis document and its chapters.

1. Introduction
   The first chapter introduces the background of the topic, the motivation, research question, objectives of the research, scope and the research methods used for this Master’s Thesis.

2. Related Work
   The second chapter firstly discusses the technology concepts used in the Thesis work: REST services, service-oriented architectures (SOA) and OSGi. Then, the state of the art in the field of service delivery platforms is presented in the second section.

3. Semi-Automatic Code Generation: The Design
   Chapter 3 introduces the concept of the semi-automatic code generation and shows the results of two comparative studies: one on different service description languages, and one on code generation alternatives. In addition, it arguments the choices made for the present research work.

4. The Implementation
   The fourth chapter is dedicated to presenting the technical details of the implementation. After a description about how RESTful services can be called, the definition of the service description language is provided. Then, the following sections discuss the working mechanism of the template engine, the template utilized in this project and how the OSGi integration is made.

5. Discussion and Achieved Objectives
   In the Discussion and Achieved Objectives section, the results of the study are evaluated. Therefore, we assess how the research questions are answered and the goals of the thesis are accomplished. In addition, the degree of generality that can be achieved for the automatic code generation on the service delivery platform with the proposed
concept is explained.

6. Conclusion and Future Work

The last chapter summarizes the contributions of the Master’s Thesis. Furthermore, it discusses possible directions for the future in the field of semi-automatic service integration of telecom and Internet services in service delivery platforms.
Generally, the SDP research community has agreed that a service delivery platform should follow the service-oriented architecture style [15], due to necessary features such as reusability, discoverability, composability and loose coupling of services. In this chapter, we firstly give an overview on the technology concepts including the REpresentational State Transfer, service-oriented architectures and the Open Services Gateway initiative, and then discuss the state of the art in the field of service delivery platforms.

2.1. Technology Concepts

An increasing number of organizations expose their information and resources through Web services nowadays, thus opening opportunities for new value creation and new applications, often called service mashups [26], which do not necessarily have to be defined at design time. Nevertheless, there are no uniform interface standards available, and standards such as SOAP, Web Services Description Language (WSDL) and Universal Description, Discovery and Integration (UDDI) language do not seem to be enough [27]. In this context, the lightweight REST-based architectures [28] have gained popularity and have become the preferred option for Internet-scale applications during the recent years.

For instance, in 2004, Amazon.com had two APIs for service exposure: one SOAP and WSDL-based, and one Hypertext Transfer Protocol (HTTP) and Extensible Markup Language (XML)-based, thus REST [29]. Recently, they have decided to shift towards the REST approach. Another example is Google, which used SOAP for the Google Web Search service until 2008 and then turned to REST, offering a parametrized Uniform Resource Locator (URL) that returns a JavaScript Object Notation (JSON) data set. Therefore, they admitted that the Web Services stack is unnecessarily complex and neither scalable nor visible enough. For these reasons and also for advantages such as human-readability, lightweightness, compatibility with the SOA thinking and ease to build, REST is the approach we chose for this project.

As far as the application of the concepts is concerned, the framework chosen is the
Open Gateway Initiative, since this supports the implementation of component-based service-oriented applications, and also provides means to publish and search for services [30].

The following subsections provide a deeper insight into REST, SOA and OSGi.

### 2.1.1. REpresentational State Transfer (REST) Services

REST was first defined by Roy T. Fielding in his Doctoral Thesis [28], as a means of communication between the basic concepts underlying the Web. This is the reason why it is also known as “the architecture of the Web” [31]. Nevertheless, its popularity has continuously grown since then, and it has been used on a much more general scale, since REST implementations are seen as increasingly important in the modern services landscape [32].

It is largely accepted that REST is to be recommended for rather basic, ad-hoc integration scenarios, whereas Web Services (WS-*) and implicitly SOAP, are better suited for stricter requirements regarding flexibility and quality of service [33]. As an architecture, its style is client-server, with clients initiating the requests, and servers receiving the requests, processing them and returning the results. The resources that can be addressed are any meaningful concepts, and their states are captured in a representation [28]. What differentiates the REST style from other client-server software architectures are the guiding principles presented below. Although there is no commonly agreed upon list of principles, most authors consider the following:

1. **REST services are stateless**
   - This means they do not require their states to be stored on the server. Therefore, all requests from the client to the server will be self-sufficient and will not rely on any information stored remotely [32, 34]. The same principle is formulated differently by Pautasso et al.: REST services are stateful through hyperlinks [33]. The meaning is the same, although in this case the emphasis is put on the fact that all the necessary information is contained by the hyperlink.

2. **REST services have a uniform interface**
   - They highly rely on HTTP, but they are theoretically protocol-independent [32–34].

3. **Resources are manipulated through representations**
   - Most often, the representations are XML documents. Nonetheless, a resource can be given in a variety of formats: XML, JSON, eXtensible HyperText Markup Language (XHTML), or as a Joint Photographic Experts Group (JPEG) image [32]. This principle is called “addressability” by Burke [34].
4. Messages are self-describing

This principle is one which sometimes generated heated debate, since its meaning is often considered synonymous to the meaning of the first listed here. However, other experts of the field list it as a separate standalone feature [33].

In addition, the idea of using hyperlinks within the data format received from a service is sometimes referred to as HATEOS (Hypermedia As The Engine of Application State) [34].

As far as the technical implementation is concerned, REST uses some features of the HTTP protocol, such as the concept of methods, Internet media types and Uniform Resource Identifiers (URIs) [35]. Therefore, a RESTful web service needs to have these three aspects very well defined.

The URI is an identifier which is given by the provider of the web service.

The Internet media types indicate the types of data supported by the web resource, and can be any valid HTTP type. Nevertheless, the most common, in the context of RESTful web services, are JSON and XML.

The HTTP methods are the operations which are supported by the service. These include: GET, PUT, POST, DELETE. GET requests a representation of the specified resource, PUT uploads a representation of the specified resource, POST sends the data to be processed to the resource, and DELETE deletes the specified resource.

In order to exemplify the theoretical concepts above, we present two samples of REST calls in Figures 2.1 and 2.2.

Figure 2.1 depicts how the call history, including all the calls made by a specified User, can be retrieved by making a GET on the Call resource. The POST method is exemplified by the code in Figure 2.2, which shows how an active call can be terminated, by using the PUT on the RESTful telecom web service [1].

2.1.2. Service Oriented Architectures (SOAs)

Sometimes seen as the next compelling step after Web 2.0 [6], the SOA concept has started to be received with increasing interest and has become much more than a simple buzzword. Generally, the service orientation design paradigm usually refers to individually shaped logic units, which can be collectively and repeatedly utilized [36].

According to Erl [36], such logic units are called services. Other researchers go further and give more detailed definitions for services. For example, they are seen as mechanisms which enable access to one or more capabilities [37], or even defined as self-describing, open components that support rapid and low-cost composition of distributed applications [38].

Regardless, the SOA foundation is composed of such basic services, their descriptions, and basic operations, such as publication, discovery, selection, and binding, which produce or utilize such descriptions \[38, 39\].

Although it is not extremely challenging to define its components, it is rather problematic to give a definition for SOA \[40, 41\]. It is not an architecture - it is a paradigm, style, concept and philosophy, which lead to a concrete architecture.

Nevertheless, the ways it is addressed vary greatly, from architecture to model or
paradigm, or even only by enumerating its main characteristics. For instance, Davis defines it as a kind of architecture that uses services as building blocks to facilitate enterprise integration and component reuse through loose coupling [40]. According to other researchers, it is better described as an architectural model that aims at enhancing the agility and cost effectiveness of an enterprise while reducing the burden of Information Technology (IT) on the overall organization [36]. The most comprehensive definition, which encompasses both the basic operations and the service ownership issue, is the one given by the Organization for the Advancement of Structured Information Standards (OASIS) [42]:

"SOA is a paradigm for organizing and utilizing distributed capabilities that may be under control of different ownership domains. It provides a uniform means to offer, discover, interact with, and use capabilities to produce desired effects consistent with measurable preconditions and expectations".

In order to give a correct overview on SOA, one should not overlook the main characteristics [36]. These show how SOA is not only about the way technologies are used, but also greatly about the business motivation and requirements:

1. **business-driven**: the technical architecture has to be aligned with the business architecture; hence, careful consideration needs to be given to maintenance over time, due to the frequent changes that might occur on the business side;

2. **vendor-neutral**: no proprietary vendor platforms should be used, but an open approach;

3. **enterprise-centric**: a SOA always needs to have a strong meaning and bring value to the enterprise;

4. **composition-centric**: SOA supports repeated composition cycles, service aggregation and agility to deal with changes.

Despite being largely accepted in literature, these four characteristics are sometimes considered to be too business-oriented and not to have enough technical substance to thoroughly describe a SOA [40]. Therefore, Davis proposes considering the following, which give more importance to the technical aspects:

1. **service interface or contract**;

2. **service transparency**;

3. **service loose coupling and statelessness**;

4. **service composition**;

5. **service registry and publication**.

Interestingly, Erl also uses some of the previously mentioned characteristics when he builds the list of SOA design principles [43]. These are based on the separation of concerns, a software engineering theory according to which complex problems should be solved by decomposing them into simpler, smaller problems. This may sometimes...
lead to conflicts between different design principles, some competing against others, and a compromise will always need to be reached when a new architecture is designed. The main principles are:

1. **Standardized service contracts**: meant at enabling inter-operability between services.
2. **Loose coupling**: aiming at having fewer dependencies, on the one hand, between service consumers and service implementation and, on the other hand, between the service contract and service implementation.
3. **Abstraction**: details are limited to the essential and necessary information.
4. **Reusability**: the idea is to build services with functionalities which are required and easy to use also in other contexts or scenarios; this principle enables composability.
5. **Autonomy**: services should be as independent as possible from outer influences.
6. **Statelessness**: with very few exceptions (for instance, the single sign-on service), it is desirable to have services which are active only when required, for imposed tasks.
7. **Discoverability**: in relation to the reusability principle, services should be easy to discover, that is identify, understand and interact with them.
8. **Composability**: services should be compatible with one another in order to assist in creating new services with new functionalities.

One common point for REST and SOA is their desire to achieve loose coupling [44], by allowing different components of the distributed systems to evolve autonomously. Moreover, REST brings an important constraint related to the uniform interface, and this will allow for more highly scalable systems, which is also a goal for SOA. In addition, the way REST handles data formats by allowing services to have multiple accepted types also supports scalability. Furthermore, self-describing messages are promoted by REST, and this will make discovery and composability in the context of SOA possible. Therefore, REST is actually a manifestation of resource orientation [37], and the concepts of REST can be used to build distributed services and model service-oriented architectures [34].

### 2.1.3. The Open Services Gateway initiative Framework (OSGi)

The demand for complex software is constantly growing nowadays, and the possibility to integrate existing code into large software systems is receiving more and more attention. Although the idea of adapting existing functionality to perform in a new environment has been around for decades, the issue of integration has not been solved yet. Therefore, tools for standardizing these aspects are needed, to make the re-usability of components possible, reliable, robust and inexpensive [2]. One way to address this diversity and heterogeneity of existing services is to implement the solution provided by
the OSGi Alliance [45]. Moreover, this will also solve two main limitations which are to be identified when using the modular system employed in Java. On the one hand, packages and Java ARchive (JAR) files cannot benefit from individual restricted access for their classes, and this will easily lead to highly-coupled systems. On the other hand, the modular system of Java is rather static: modules can only be updated at deployment time, which will result in the need to stop and restart the system in case a change is made [3, 30].

The Open Services Gateway initiative framework is a dynamic module system and service platform for the Java programming language. Java is portable and platform-independent, and the OSGi technology completes it by providing the standardized primitives that allow applications to be built from different components, in a service-oriented manner. There is no standard pattern for SOA implementation, but OSGi and SOA have one essential feature in common: the service is the smallest integration unit, and the communication between different services is possible via resource sharing [46]. With OSGi, composition is done dynamically, since software systems are constructed from independent modules, called bundles - the smallest units of modularization in OSGi [47]. These can be remotely installed, uninstalled, updated, activated and deactivated, without requiring a system restart [2, 3, 48, 49].

Currently, there are three main open source implementations of the OSGi Specification Release 4 available: Apache Felix [50], Equinox [51] and Knopflerfish [52]. In spite of some minor differences, they are all based on the key specifications of OSGi: the architecture, bundles and life-cycle rules.

The Architecture

Figure 2.3 depicts the typical OSGi layered architecture. This includes the execution environment, modules, life-cycle, services, security and bundles [2]; the native operating system and the Java Virtual Machine (Java VM) are the foundation of the architecture.

**The Execution Environment** is used for defining what methods and classes are available on a specific platform.

**The modules** layer is meant for defining dependencies related to importing and exporting code.

**The life-cycle** provides the API for installing, starting, stopping, updating and uninstalling bundles.

**The services** layer has the aim to dynamically connect bundles.

**The security** handles the security aspects, imposing limitations where necessary.

**The bundles** are Java JAR files containing a manifest file, used to declare static information about the bundle, and class files, required APIs, images, icons.
The Bundles

For OSGi, modularity is one of the core features, and this is embodied in the bundle concept [2]. According to Bartlett [3], a module has to be:

**Self-contained.** A module is seen as a single unit, thus making it possible to install or uninstall it independently. Nevertheless, it does not have an atomic structure, since it is composed of different parts, but those constitutive parts are strongly inter-dependant.

**Highly cohesive.** The responsibilities and functionalities of a module need to be strongly related, and the logical purpose should be very clearly defined.

**Loosely coupled.** The internal implementation of the modules is neither important nor relevant when the composition between different modules takes place. This leads to meeting an essential design principle of service-oriented architectures, thus making it possible to change the implementation of a certain module, without having to restart the others it interacts with.

A bundle is a cohesive, self-contained unit [47, 53], consisting of a JAR file and additional meta information. The meta information is included in a manifest file called MANIFEST.MF, which is to be found in the META-INF folder. Such a file has a standard structure and uses the OSGi headers, as shown in Figure 2.4.

*Bundle-Name:* is an optional header, enabling the developer to give a human-readable name to the bundle.
Bundle-SymbolicName: is a compulsory header, specifying a unique identifier for the bundle.

Bundle-Description: is a human-readable description of the functionality of the bundle.

Bundle-ManifestVersion: shows which OSGi specification to use for reading the bundle.

Bundle-Version: gives the bundle a version number.

Bundle-Activator: indicates the class name to be invoked once a bundle is activated.

Export-Package: shows which included Java packages will be made available.

Import-Package: informs about the dependencies of the bundle and which Java packages from outside are needed.

The Life-cycle

As previously specified, the life-cycle layer adds bundles which can be dynamically installed, started, stopped, updated and uninstalled. Class loading takes place in the Modules layer and the API meant at managing modules at run-time is added in the life-cycle layer.

A bundle can be in one of these six states: INSTALLED, RESOLVED, STARTING, ACTIVE, STOPPING or UNINSTALLED [2, 49]. The cycle a bundle can go through is shown in Figure 2.5.

The meaning of the states is the following:

INSTALLED: the bundle was successfully installed.

RESOLVED: the bundle is either ready to be started (having all the dependencies available), or it was stopped.

STARTING: the bundle is being started by calling the BundleActivator.start method; it can be found in this state until the activation policy is resolved.

ACTIVE: the bundle is running, which means that the BundleActivator.start method returned.

STOPPING: the bundle is being stopped by calling the BundleActivator.stop method.

UNINSTALLED: the bundle was uninstalled.
2.2. Service Delivery Platforms - State of the Art

For about a decade, service delivery architectures have evolved greatly, aiming at integrating both IT and telecom capabilities to create services that cross network and technological boundaries. Thus, the term service delivery platform, referring to an architectural style applied to software design and provision [15], started to be used very often, and an increasing number of Communication Service Providers (CSPs) adopted this idea as a novel method for generating revenue. This is in complete opposition to the "stovepipe" approach [9], which promoted the idea of having one different platform for each service.

More recently, the popularity gained worldwide by Application Stores provided by significant players such as Apple [18] or Google [19] has drawn researchers’ and practitioners’ attention to SDPs even more. In this context, the delivery platforms are seen as the perfect means for the telecom operators to expose their infrastructure assets both internally and externally, to enable developers to create new applications, and thus generate revenue from the data already residing in their data centers.

There have been several approaches to unify various implementations of the SDP concept under one single standard. For instance, the Open Mobile Alliance [54], the leading standardization organization for mobile service-related technologies, focused on how simple services can be re-used and combined to create new services, by defining a reference architecture called the OMA Service Environment (OSE) [9]. Furthermore, TM Forum [55] emerged as a global industry association which tries to define standardized specifications in the field of SDPs, and published the Service Delivery Framework.
Nevertheless, there is no commonly agreed upon standard for service delivery platforms, since different providers usually choose their own ways depending on their business models and infrastructure resources.

Stakeholders’ Needs

As identified by Menkens and Wuertinger, the main stakeholders involved in the development of a service delivery platform are the communication service providers, application developers and end-users. All these form a triangle and their needs have to be carefully taken into account so that the SDP brings additional value to them all.

Firstly, the communication service providers are interested in being competitive on the market. Due to the increasing popularity of service mashups, telecom operators have started to face new competitors which, interestingly, exclusively operate with Internet services, such as Google. Therefore, they need to provide reusable services of high quality and also short development cycles of services and converged applications (Telecom/Internet convergence). Moreover, it is essential that they adhere to the quality parameters (and standards, when available) and their services are available on all end-user devices and platforms. In addition, CSPs ideally make best use of the telecom infrastructure and data they own, and turn these into differentiating points from the Internet service providers.

Secondly, the application developers require a development platform with state of the art technologies and functionalities, available APIs, and based on programming languages which are well-known and used on a large scale. The short development cycles of applications are also important, and so is the low barrier of entry. This means they should be able to develop new applications using existing services and resources on the SDP at low costs, having minimal hardware and additional tools requirements. Furthermore, developers need support from the CSPs for the distribution to end-users and marketing of applications, and a sustainable business model.

Thirdly, end-users want their needs to be addressed efficiently by easily searching and finding the most suitable services on the SDP. The applications have to be rich, user-friendly, customizable and easy to install. In addition, they should be configurable on all the devices end-users might have (computer, smartphone, IPTV, and others), and the interfaces should change to adapt to the characteristics of the devices. The data ought to be easy to transfer from one device to another, and synchronization should be possible between different devices running the same application.
Requirements for the SDP

According to stakeholders’ needs formulated above, and also specifically concentrating on the business needs, the main requirements for a service delivery platform for a telecom operator were defined [14]. To begin with, shorter times for developing applications are required [14], and therefore faster time-to-market [9, 12]. This will address the needs expressed by all the three groups of stakeholders identified. As the demand from end-users grows, the dynamics and flexibility of the development process must be improved to allow services to merge with others in a service-oriented manner.

Strongly connected to this is the second requirement, according to which it is important for the SDP to enable the easy creation of applications [12, 14]. This covers the need related to the available APIs and the programming languages used to develop the new applications.

Ohnishi et al. [14] also suggest the necessity of using multivendor services and interworking with technologies from other domains. This is related to the SOA paradigm and promotes exposing interfaces. Although in order to achieve this, standards are needed and the creation of mashups between telecom and Internet services is still at an early stage, significant progress has been made recently.

Furthermore, the SDP needs to enable customization [14] and offer compelling user experience [9]. This requirement includes the idea that one of the differentiating points for the CSP on the competitive market actually resides in its own network resources and data, which can be used to its advantage, by creating new services in a converged fashion with Internet services.

Lastly but not least, ways to achieve cost efficiency need to be investigated [9, 12]. This is one of the requirements with the highest degree of complexity, since it relies on the other requirements already mentioned, and it also greatly depends on the specificity of the business model implemented by each telecom operator. Nevertheless, there are conditions like those presented above which, when fulfilled, will surely also contribute to achieving cost efficiency.

The SDP Concept Used

Based on the state of the art in the field of convergence of telecom and Internet services and SDPs, the present project aims at filling the gap between the communities of developers and the telecom industry, by creating a service delivery platform which takes into account the stakeholder requirements and brings together both Internet and telecom services.

In this respect, the SDP definition used as reference is the one provided by OMA. We are therefore following the OSE specification, which is also considered on a larger scale.
than SDF in the Telecom domain \[11, 14, 58\]. In addition, one of the premises of the research project is that distributed capabilities which can be found under the control of different ownership domains can be organized and utilized for common goals by using the SOA paradigm. This way, in the Telecom field, SOA facilitates opening up the network resources of the CSPs by means of reusable telecom service enablers \[12\].

Figure 2.6 depicts the service delivery platform concept used in this project.

**The Telecommunication Layer** includes the specific telecom services which are focused on communication features. For example, these can be voice, video, text messaging or IP Multimedia Subsystem (IMS) service enablers.

**The Telecom Protocol** ensures the connection between the Telecommunication Layer and the Application Services Layer. Most often, in the case of Next Generations Networks (NGNs), the SIP protocol is used.

**The Internet/Web Services Layer** includes Internet or web services, which are easily accessible via open interfaces offered by service providers.

**The Service Protocol** is used for connecting the application logic of a converged application with the services in the Service Layer which were used to build it. In a REST architecture, this protocol will most often be HTTP.

**The Shared Service Layer** contains simple, re-usable services, which are ready to be utilized by the developers on the platform. They can be loosely-coupled among themselves, and make use of the resources which are to be found in the Telecommunication
and Internet/Web Service Layers.

The **Application Services Layer** contains the bundles or packages which are converged applications of Internet and telecom services. Therefore, these represent the essential part of the service delivery platform.

The **Application Protocol** is an application-specific protocol (based on the Internet Protocol), which connects the converged application logic with the Graphical User Interface (GUI) Layer.

The **Graphical User Interface Layer** includes various interfaces for different devices on which the applications can run.

The **Network Layer** represents the telecom infrastructure, including the IP Network, Global System for Mobile Communications (GSM) Network and Universal Mobile Telecommunications System (UMTS) Network.
Semi-Automatic Code Generation: The Design

This chapter introduces the concept of the semi-automatic code generation implemented in this project. Firstly, a comparative study on existing service description languages is conducted, and the arguments for the language chosen for the project are provided. Secondly, different source code generation alternatives are evaluated, and the code generation option selected is presented in more detail.

3.1. The Concept of the Semi-Automatic Code Generation

As discussed in Section 2.2, the main goal of the service delivery platform is to enable the communities of developers to easily implement new applications, by using the resources available on the platform. This way, they bring value to the CSPs, which make the best use of their existing infrastructures and data, thus meeting the needs of the end-users. In order to accomplish this, the SDP has to provide the developer communities with the basic services for the functionalities they require.

The problem identified in this context is the lack of standardization for the APIs for both Internet and telecom services. Numerous providers (such as Google and Yahoo) describe their services in an utterly non-standardized and non-structured manner, predominantly textually, in the form of HyperText Markup Language (HTML) pages. Therefore, a method to fetch the information from those descriptions and make it available in a structured form on the SDP needs to be investigated. Moreover, the services need to be described in a uniform way, and then included in the Shared Services Layer, as depicted in Figure 2.6, which presents the architecture of the SDP.

Consequently, we are proposing a concept which addresses the problem determined by semi-automatically generating source code from structured descriptions, and bundling this as standalone services, which can then be re-used by developers. Figure 3.1 depicts...
the concept of the semi-automatic code generation developed in this research work. The working mechanism is the following.

1. An employee of the communication service provider which owns the SDP looks at the interface descriptions of the Internet and telecom services, which are found in the Internet/Web Services Layer, and the Telecommunication Layer, respectively, in the SDP architecture. These will most often be textual descriptions providing information about the service resource location, required and optional parameters, their names, types and value, output, and other features.

2. He or she fills in a description file, which is a textual file, with a standard predefined structure. This will include the information fetched from the Web/telecom service interface description.

3. The description file is computed by a code generator, which exports source code (a JAR file) as output.

4. The source code is bundled as an OSGi service and added to the Shared Services Layer. Therefore, the Internet or telecom service, which is described only textually by its provider, is turned into compilable code on the service delivery platform, which can be straightly used by the communities of developers.
The aim is to have the semi-automatically generated code written in a language which is well-known and understood by developers, who will see it as a “black-box” - they will only have to give the appropriate input to the functions, and the result of the executed code will be the expected output. Therefore, they will be able to easily integrate it in complex applications and deliver new value for the end-users in a short time and using quality, verified and reliable resources.

The only reason why the concept is called semi-automatic, and it is not considered entirely automatic, is that it requires the human intervention in the incipient phase: the employee of the CSP who processes the information read about the service interfaces and fills in the description file. The code generation performed afterwards, based on the structured description file, is completely automatic.

Whereas the first and last step of the code generation process are rather straightforward and do not present any possibilities to use different alternatives, the language in which the service is described and the way the code generator is implemented can vary greatly and can depend on numerous factors. Moreover, these two choices are particularly significant for the end-result of the research project, and we therefore assess the different options in more detail in the following two sections.

### 3.2. Service Description Languages

When the idea of WWW appeared, only one type of interaction was considered: that between the web application and the end-user [59]. Therefore, the application did not need a machine-processable formal description; however, we are now confronted with a new situation. The emergence of web services has brought new requirements in the field of service descriptions and it has been proved that natural language-based descriptions are not enough any more [59, 60], and a mechanism for machine-to-machine interaction needs to be defined. In this context, new specifications have appeared to support the automation of new software development, based on machine-processable descriptions.

#### 3.2.1. Details of Service Description Languages

In spite of common requirements and motivation, there is no holistic framework which covers the lifecycle of a REST service and there is no agreed standard for describing a RESTful Web service. Numerous approaches have been taken depending on particular problems which needed to be addressed in various contexts, and the consequence of this is the large variety of languages available nowadays. This study comprises information about nine such specifications and, after briefly introducing each of them, a comparison is made, based on nine different criteria. The languages discussed are: Web Application Description Language (WADL) [59, 61], WSDL [60, 62], RESTful Interface
Definition and Declaration Language (RIDDL) \cite{63,64}, Web Resource Description Language (WRDL) \cite{65}, Web Description Language (WDL) \cite{66}, Simple Message Exchange Descriptor (SMEX-D) \cite{67}, Norm’s Service Description Language (NSDL) \cite{68}, REstfuL SErvices DEscription Language (RESEDEL) \cite{69} and Unified Service Description Language (USDL) \cite{70-73}.

**WADL**

WADL is a language which aims at providing a machine processable description of HTTP-based Web applications \cite{61}, such as those following the REST architectural style. Consequently, it is a resource-centric description language and WADL documents mainly consist of resource descriptions. Therefore, the applications are described as basic operations (GET, POST, PUT, DELETE) on the resources that comprise the state of the applications.

The main premise of WADL is that most web applications today use textual documentation and, whereas this is adequate for human consumption, it is not satisfactory enough for three main use cases which require a machine-friendly description format. These are: application modelling and visualization, automated code generation and configuration.

Unlike other languages which are designed to be as general as possible, WADL has the goal of being simple, providing only the absolutely necessary features. An example in this respect is represented by the protocols supported. WADL is meant for web applications which use the HTTP protocol, and it is not suitable for others. This characteristic makes it appropriate in the context of RESTful services, since HTTP is the only protocol available, which is also remotely RESTful \cite{59}. In addition, although the WADL specification does not truly require that a web application be stateless, it does not provide for any of the stateful features of HTTP, such as cookies for instance.

As far as the technical details are concerned, WADL documents have the XML namespace: `http://wadl.dev.java.net/2009/02` and the following components: application, documentation, grammars, resources, method, request, response, representation and parameter \cite{61}.

The application represents the root of a WADL document, and the documentation (or doc element) is used to document the element whose child it is - one element can have one or more doc elements.

The grammars include all external data types, and no specific schema language is mandated, but the REgular LAnguage for XML Next Generation (RELAX NG) or XML Schema can be used.

The resource references the base path for accessing a tree of resources, and it can hold multiple resource elements. Consequently, a resource element describes a set of resources,
and each of these is identified by a URI that follows a common pattern. Such an element can have the attributes: id (an identifier), path (a relative URI template for the identifier of the resource), type (a resource type) and queryType (defining the media type for the query component of the resource).

The method element describes the input to and output from an HTTP protocol method that may be applied to a resource, the request element describes the input which needs to be included when applying an HTTP method to a resource, and a response element describes the output that results from performing an HTTP method on a resource.

The representation element shows the representation of a state of a resource, and a parameter (param element) describes a parametrized component of its parent element.

Currently, the WADL specification is not a World Wide Web Consortium (W3C) member, but has a pending application. Nevertheless, an increasing number of service providers have started to use it on a larger scale, and there are concrete examples of such descriptions from significant players such as Yahoo and Amazon (for example, Amazon uses exclusively WADL for describing the search service).

**WSDL**

WSDL emerged as a result of the same need of providing a machine-processable means of formally describing Web applications. In addition, its model has generality as a goal, thus being protocol-independent [59]. The scope of our study is only the WSDL 2.0 specification, which also has good support for HTTP bindings and the REST way of thinking. A WSDL description contains all the details of a web service, including: the URL of the service, the communication mechanisms it understands, the operations it can perform, and the structure of the messages [60, 62].

As far as the technical details are concerned, WSDL documents use the core namespace: http://www.w3.org/ns/wSDL. The root element is the description element, and this has four child elements: types, interface, binding and service.

The types element contains the XML schema element and type definitions which describe the messages of the Web service. Most often, WSDL is used with W3C XML Schema, although it theoretically is compatible with any schema language. The interface defines the operations of the Web service, and also includes the specific input, output and fault messages which are passed, and the order in which they are passed. The binding describes the way in which a client can communicate with the service, therefore how the binding is established. The service element associates an address for the Web service with a specific interface and binding.

As far as the practical applicability and popularity of WSDL is concerned, it has been the de-facto standard for describing Web services since June 2007, when it was declared a W3C recommendation. It is currently still used for both REST and SOAP services al-
though, unlike WADL, its specific orientation is definitely towards interfaces.

RIDDL

Starting from the fact that there is no standard way of describing REST interfaces [64], Mangler et al. appreciate both initiatives of using the WADL and WSDL specifications [64]. However, they identified some drawbacks of these languages and propose defining a new means of describing services, which is more flexible, extensible, and as simple as possible [63].

The main issues about WADL they observed are related to the service evolution and service composition. On the one hand, WADL documents include the endpoints of services, and this characteristic hinders the descriptions from being highly reusable when the services are moved between different locations. On the other hand, service composition is not always possible because existing services cannot always be modified to add new functionalities, since this would only break the applications which already use them. Therefore, they suggest providing a language which can describe RESTful services and also support design principles such as composition and evolution. In their view, the evolution allows to transparently modify existing functionalities without affecting the services and the composition enables adding new functionalities to existing services without modifying them, different service descriptions being merged to form new composite descriptions [64].

Similarly, there are several features they observed, which could be used as starting points to improve the WSDL specification. Firstly, the new language should not hold vocabulary to define elements which are no longer used. Secondly, descriptions should be reused for multiple instances of services, running on different machines. Thirdly, an enhanced description should be protocol-independent and extensible, thus allowing new types and messages. Lastly but not least, XML should be used as the main standard for data streaming [63].

Based on all the weak points of the two main languages considered and the improvements suggested, a new description language was defined: the RESTful Interface Definition and Declaration Language, which is meant to address the issues of both WADL and WSDL.

WRDL

Previously known as the Web Service Operations Language and the Simple Web Service Behaviour Language, the Web Resource Description Language defines its goal as describing the runtime behaviour of services, or the transitions from one document to another, starting from the premise that services consist of XML documents and XML
schemas provide only partial descriptions of the services \[65\].

According to its authors, WRDL is intended to be the Interface Description Language (IDL)/WSDL for HTTP Web services. Therefore, adopting the REST strategy, they suggest that resources should be represented by XML document types and, whereas each document has an associated XML Schema, the resource behind the document should have an associated operational description.

In order to describe services, elements such as \texttt{resourceType}, \texttt{method}, \texttt{input}, \texttt{output}, \texttt{query} and \texttt{header} are used. The \texttt{resourceType} element declares a resource type and can have one or more methods. A \texttt{method} is an element of type \texttt{GET}, \texttt{POST}, \texttt{DELETE}, \texttt{PUT} or \texttt{otherMethod}. The \texttt{POST} methods and \texttt{otherMethods} can have \texttt{creates} attributes, which declare the \texttt{resourceType} of the sub-resource that will be created. If the \texttt{input} and \texttt{output} elements require no child elements or attributes, then they can be omitted. The \texttt{query} and \texttt{header} are sub-elements of \texttt{input} and \texttt{output}, and extra headers will always be allowed, thus making extensibility possible.

\textbf{WDL}

The Web Description Language is based on some of the design choices of WSDL 2.0, such as the re-use of Schema, definitions of operations, URI construction syntax and configuration of HTTP parameters. Nevertheless, ”it targets a different design center: operations can be defined and then associated with resources” \[66\].

WDL is a language oriented on resources and the documents will essentially be lists of resources, which are defined for locations. The URIs are used in a manner which is similar to that of WSDL 2.0, the path being defined by the instance content of a schema type. Moreover, the resources elements contain operation elements, and multiple inputs, outputs and faults can be specified for each operation. Another important design decision for WDL is that different media types can be specified for both inputs and outputs, and the faults are similar to WSDL 2.0: they are described with type, code and header. Therefore, WDL is similar in numerous respects to WSDL and also keeps the interface orientation.

\textbf{SMEX-D}

The aim of SMEX-D is to provide simple descriptions of a wide range of web service message exchanges, both REST-based and SOAP-based \[67\]. A Simple Message Exchange (SMEX) is defined as a pair of messages (request and response) sent between two systems referred to as client and server.

The namespace used by SMEX-D is \texttt{http://smex-d/ns/}, and all elements in a SMEX instance are in this \texttt{namespace}, whereas the attributes are in no namespace, unless other-
wise specified. The typical SMEX-D document has the root element *smex-d*, which has two child elements: *request* and *response*. The compulsory attribute of *smex-d* is *href*, which gives the value of the URI of the service resource. The request and response elements also have a compulsory attribute, which is *form*, and this can take as value *pairs*, *soap* or *nsx* (non-SOAP XML), or *soap*, *nsx* or *none*, respectively. Another technical feature is that the response message is sent to the client as an HTTP response to the request message. Moreover, another element used is the *language* element. This informs about the language which appears in the non-SOAP XML, or in the header or body of a message, and must either have a *namespace* or contain one or more *schema* elements. The values which can be taken by these are *relaxng* (for RELAX NG), *xsd* (for XML Schema Definition - XSD) or *schematron* for Schematron.

Therefore, based on the definition and features of SMEX-D, it can be easily concluded that it truly provides one of the simplest ways to declare the way XML messages are exchanged.

**NSDL**

NSDL is an initiative by Norman Walsh, who decided that there is an obvious need for a service description language, but the largely-used WSDL is not enough to meet his needs [68]. Therefore, he proposed NSDL, a new language which can enable programmers to use web services transparently, and achieve some level of interoperability with standardized interfaces, in a simple fashion.

From the technical perspective, a typical NSDL document firstly describes the *service*, which has three attributes: a *name* (which defines a method), an *action* (which is one of the HTTP action verbs, GET for example) and a *uri* (the resource identifier). Then, the *request* contains the parameters that the service can have; various values can be defined for these, such as *type*, *default* and others. The positional parameters in the method invocation get mapped to the list of parameters in the request block. The response shows what is returned, and can also be augmented to look for errors.

The existing documentation for NSDL [68] provides examples of how both GET and POST operations can be used, and also gives direct links to various useful relevant resources available, among which a complete RELAX NG Grammar for NSDL can be very useful.

**RESEDEL**

Although the REstful SErvice DEscription Language only benefits from a rather poor documentation [69], it is still considered a good trial of designing the ideal description language for services. It is wanted to be a hybrid of SMEX-D and NSDL, and to com-
prise the best features of both languages.

As far as the technical details are concerned, RESEDEL uses the namespace available at: http://www.ccil.org/cowan/resedel/ns, and the default schema language RELAX NG. Services have an id, URI, and belong to one of the four types corresponding to the Create, Read, Update, Delete (CRUD) database operations. The documentation provides examples for all these types of services, a wrap request-response in SOAP, different types of responses, possible URI definitions and others.

USDL

One observation which needs to be made in the context of this section is that our focus was mostly on syntactic languages, and not on semantics or other tangential approaches, such as Semantic Annotations for WSDL (SAWSDL) [74, 75] or HTML for RESTful services (hRESTS) [76, 77]. However, we included USDL, since it is the major release in the field of semantic web service description languages and completes the overview given in our study.

USDL has been developed mainly by SAP Research [72] as a specification language to describe services from a business, operational and technical perspective, aiming at complementing the current Web services stack [73]. Thus, a universal description includes four main elements: a name, a textual description, keywords and a set of ontological concepts.

The motivation for defining this new language rests in the purely syntactic nature of almost all the other existing approaches, among which WSDL is the most important one, according to the creators of USDL [71]. Therefore, they initially wanted to provide semantics to WSDL statements, and they designed the new markup language in a way that it relies on two other distinct formal languages: WSDL and the Web Ontology Language (OWL). This way, the scope of WSDL is to describe message formats, types and method prototypes, whereas a specialized OWL ontology is utilized for formally describing what the messages and methods mean on a conceptual level [70, 71].

Although originally thought of as a language which complements the existing syntactic ways of describing services, USDL has evolved as a separate, standalone solution for describing the semantics of web services and has gained more and more popularity during the recent years. Nevertheless, the field of semantics is not considered to be a mature one yet, and efforts are still made to achieve a minimum degree of standardization. In this respect, USDL is part of the W3C Incubator Group until September 2011.
3.2.2. Comparative Study on Service Description Languages

In order to give a structured overview of the different service description languages presented, we built a comparison based on nine distinct criteria: the officially formulated purpose of the language, orientation, fundament, W3C membership, stateless feature requirement, HTTP authentication, Schema language, description type and specific operations. These were chosen to enable the assessment of suitability of these languages for the purpose of the present research study. Therefore, the main goal was to evaluate which language (if any) is appropriate for the requirements of the concept presented in Section 3.1. Tables 3.1 and 3.2 show the comparison of the nine service description languages introduced above.

1. The officially formulated purpose

The first criterion used is the officially formulated purpose. Here, several strategies can be observed. Whereas some languages aim at achieving generality (WSDL, USDL), others try to provide a simple way of describing services (RIDDL, SMEX-D), or even combine other approaches to satisfy specific requirements (RESEDEL).

2. The orientation

As far as the orientation of the SDL is concerned, there are two main directions. Languages such as WADL, RIDDL and WRDL are resource-centric, thus following the REST style, while WSDL and WDL put emphasis on describing interfaces. The only language which supports the two orientation types is SMEX-D, which can be both resource- and interface-centric. For instance, WADL documents are composed as sets of resource descriptions and WSDL documents are composed as sets of interface definitions, which are each comprised of operation definitions.

3. The fundament

Thirdly, the languages are compared based on the fundament. In this respect, most languages use XML, the only exception being USDL. According to its specification, it is based on WSDL and the WordNet Ontology. However, if strictly considering the syntactical formal description, it can be concluded that USDL is also based on XML, since WSDL uses the XML structure.

4. The W3C Membership

The fourth criterion is the W3C Membership. As it was expected, the degree of standardization in the field of service description languages is extremely low. Among the nine languages studied, only one is a W3C Recommendation: WSDL, since June 2009. USDL is part of the W3C Incubator Group until September 2011, when it is decided
<table>
<thead>
<tr>
<th>Officially formulated purpose</th>
<th>WADL</th>
<th>WSDL 2.0</th>
<th>RIDDL</th>
<th>WRDL</th>
<th>WDL</th>
<th>SMEX-D</th>
<th>NSDL</th>
<th>RESEDEL</th>
<th>USDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offer a machine-readable description of HTTP-based web applications</td>
<td>Provide a general model for describing web services</td>
<td>Be a flexible, extendible, and as simple as possible description language</td>
<td>Describe the runtime behavior of the services</td>
<td>Reuse some of the WSDL features, but first define operations, and only then associate them with resources</td>
<td>Provide simple descriptions of a wide range of Web service message exchanges, both REST- and SOAP-based</td>
<td>Enable programmers to use Web services transparently, achieve some level of interoperability with standardized interfaces, in a simple fashion</td>
<td>Design a hybrid of SMEX-D and NSDL</td>
<td>Aim at aligning business services by unifying them using a common description format, complementing existing standards</td>
<td></td>
</tr>
</tbody>
</table>

|-------------|-----------------|------------------|-----------------|-----------------|-----------------|-------------------------------|---------------|---------------|---------------|

<table>
<thead>
<tr>
<th>Fundament</th>
<th>XML</th>
<th>XML</th>
<th>XML</th>
<th>XML</th>
<th>XML</th>
<th>XML</th>
<th>XML</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>W3C Membership</th>
<th>W3C Member submission - pending</th>
<th>W3C Recommendation (June 2007)</th>
<th>No</th>
<th>No</th>
<th>No</th>
<th>No</th>
<th>No</th>
</tr>
</thead>
</table>

<p>| Table 3.1.: Comparison of Service Description Languages. Part I. |</p>
<table>
<thead>
<tr>
<th>Master's Thesis</th>
<th>Service Description Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Table 3.2.</strong> Comparison of Service Description Languages. Part II.</td>
<td></td>
</tr>
</tbody>
</table>
whether it becomes a recommendation or not, and WADL has a pending submission for the W3C Membership. None of the other languages is recognized as a standard.

5. The stateless feature requirement

As far as the stateless feature requirement is concerned, many languages (RIDDL, WRDL, NSDL, RESEDEL, USDL) do not specify their orientation at all. Being resource-centric and REST-oriented, WADL does not provide for any of the stateful features of HTTP, such as cookies, but it does not specifically require that a Web application be stateless. By contrast, WSDL 2.0 and WDL provide support for cookies and SMEX-D also provides some other stateful features.

6. The HTTP authentication

The next criterion used is the HTTP authentication. WADL and RIDDL do not support HTTP authentication at all, while interface-centric languages like WSDL 2.0 and WDL provide support in this respect. This feature is not clearly specified for the other languages analysed here: SMEX-D, NSDL, RESEDEL and USDL.

7. The Schema language

Among the Schema languages specified for the different description languages, RELAX NG is seemingly the most popular, being the default for most (WADL, RIDDL, SMEX-D, NSDL, RESEDEL). In addition, WADL also uses the W3C XML Schema or XSD, and SMEX-D the XML Schema and Schematron. WSDL only uses the W3C XML Schema and USDL utilizes an OWL Ontology-based Schema.

8. The description type

The description type is exclusively syntactic for almost all languages. The only exception is USDL, which is based on semantics.

9. The specific operations

The ninth and last criterion is represented by the specific operations. Whereas WADL, WSDL, WDL, NSDL and RESEDEL support the well-known four operations: GET, PUT, POST and DELETE, RIDDL and WRDL can also support other arbitrary operations, which can be defined by the users. USDL supports four operations too, which are similar in meaning, but have different names: find, create, delete, update.
3.2.3. The Description Language Chosen for the Project

Hadley [61] suggests that the ideal service interface description language should provide support for the development of modelling and visualization tools, support the generation of stub and skeleton code and provide a common piece of configuration for client and server. Starting from this recommendation and taking into account the results of the study presented in Section 3.2.2, we introduce here the theoretical fundamentals of the service description language used in this project. It will be defined in more detail in Section 4.2.

To begin with, according to our requirements, the language does not need to be utterly general, but it should provide a model for describing REST services, in a both human- and machine-readable and interpretable way. Therefore, we firstly aim for simplicity and not necessarily for generality.

As far as the orientation is concerned, since the goal of the project is to deal with RESTful services, the language needs to be resource-centric, and the interface-centric approaches should be avoided, in order to support the REST specifications.

Moreover, the language chosen to describe the services fetched to the SDP should be XML-based, since XML is a standard, simple, widely-used, robust and portable language. In addition, it is structured and also easily human-readable, which is an important requirement since the description file in our concept has to be manually filled in by a human user.

Next, it is ideally a standard recommendation, because this would ensure continuous and reliable support.

The stateless feature requirement, the HTTP authentication support and schema language used represent a set of criteria which are truly significant when the descriptions are involved in the client-server communication. Nevertheless, in the case of this particular research study, the description files are not essentially used for message sending, but rather for encapsulating information fetched from static and unstructured descriptions of Internet or telecom services, and for generating compilable reusable source code based on them. Therefore, the importance of these criteria is lower in our case, but we still choose to focus on those languages which are REST-oriented. In conclusion, the emphasis is put on those which do not provide stateful features; the HTTP authentication does not need to be supported and the Schema language can be any of the standardized options.

As previously mentioned, the language utilized in the present project should be syntactic from the description type perspective.

Summing up all the information which results from this evaluation, it is apparent that none of the nine languages compared meets all the requirements of the project. Languages such as WADL, WRDL or RIDDl are all syntactic and resource-centric, but
none of them is standardized. In addition, RIDDL and WRDL do not specify any stateless feature requirements, and by allowing user-defined arbitrary operations, they only add to the complexity of the languages. Similarly, the WADL specification contains numerous elements which are not relevant for this project and lead to building description documents which are difficult to understand and read. Therefore, we decided to define a language which is both simple enough to be easily comprehended by humans and complex enough to comprise all the important features explained.

In conclusion, it is a syntactic, XML-based language, which is resource-centric and suitable for describing RESTful services. The definition of the language is provided in Section 4.2.

3.3. Code Generation

In section 3.1, the concept of semi-automatic code generation was introduced, and the importance of having a means of code generation was explained. Therefore, it is apparent that a way to automatically generate source code from descriptions is needed to accomplish the goal of this research project. In the Computer Science literature, there are numerous terms that refer to this: automatic programming, generative programming, source code generation and others. Nevertheless, the key rests in their main purpose: maximizing the automation of application development [5].

3.3.1. General Overview on Automatic Source Code Generation

It is rather difficult to give a definition of automatic programming, since it usually is exclusively described by its features. It is a type of computer programming which has a mechanism for generating computer source code as its core. In addition, it uses generic templates, prototypes, frames or classes, and processors, generators or engines which produce the code.

According to Czarnecki et al., the key to generative programming is a generative domain model, which consists of a problem space, solution space and configuration knowledge [5], as depicted in Figure 3.2.

The problem space contains domain-specific concepts and features, and it is implemented as a Domain Specific Language (DSL). This can be materialized using various options, ranging from extensible languages (which allow for domain-specific syntax extensions such as OpenC++ or OpenJava, keyword-based programming, Jasper, graphical languages, and others) to interactive GUI and wizards.

The solution space includes the implementation components and defines the legal combinations of the implementation components.
The configuration knowledge defines construction rules, illegal feature combinations, optimizations, default settings, construction rules and default dependencies, and it is implemented in the form of generators. There are different options for building such generators and they can range from template-based approaches (for example, Jostraca) or built-in metaprogramming of a language (for example, template metaprogramming in C++) to extensible programming systems (such as OpenC++, OpenJava, keyword-based programming, Refill and Jasper). In conclusion, the configuration knowledge is the key element which translates the concepts in the problem space to components of the solution space.

In order to answer the second research question of this study, a way to implement the configuration knowledge needs to be investigated. In other words, a means for automatically generating code from the service descriptions needs to be found.

One of the main requirements of the service delivery platform project is that the code generation solution is embedded in the SDP. Therefore, available tools or applications for automatic source code generation are not to be used, since they cannot be included as loosely coupled services in the service oriented architecture designed for the platform. Consequently, such solutions are completely excluded from our research.

This decision led to choosing template engines, which are regarded as resource-sufficient for the aim of this work. A template engine is a component that takes fixed text and data as input, integrates these following certain processing rules, and outputs a text document containing the data. In this respect, the variable metamodel generation approach is used, as opposed to fixed metamodel solutions, such as those relying on the Unified Modeling Language (UML), for instance [78]. This ensures the fact that the generator is compatible with the XML-based description language introduced in Section 3.2.3.

In this chapter, when discussing about code generators, we are referring to third-generation code generators [79], which have the following characteristics. Firstly, they can generate code for an unlimited number of problems, and not only for specific cases. Secondly, the template file is an external file, and not hard-coded within the code generator as print functions, as it happens in the case of first generation code generators.
Moreover, the template starts to look like a program or script. Thirdly, the instance data becomes an explicit part of the code generator API.

Therefore, the template is the file which includes the problem-specific aspects, and the code generator is reduced to a general-purpose engine, which mixes the instance data and the template data. Figure 3.3 shows how the template-driven code generation mechanism works on a theoretical level.

The main components of the mechanism are the data model, the template, the template engine and the target.

The data model (or the instance data) contains information about the system entities which are generated. The problem-specific data of each use case is defined here, and it is organized according to a specific structure.

The template shows the syntax of the target programming language, formats the data model and contains references to entities in the data model.

The template engine takes the data model and the template as inputs and, based on these, automatically generates source code. It essentially consists of two components: one which fetches the instance data from the data model, and one which manipulates...
this data and generates the code using the template.

**The target** is the result of the code generation process, that is the source code.

Having decided that a template-driven approach is suitable for the needs of the present research, additional requirements can be defined. It is very important that the solution provided is open-source and also cross-platform. These are essential features which assist in delivering a service which works on any platform and is built on open-source software that is published and freely available to the public.

Additionally, the aim is to generate source code in a language which is complex enough, widely-used and known by developers. In order to address this need, Java language seems to be the most appropriate choice, being a general-purpose, concurrent, class-based, object-oriented language, that is specifically designed to have as few implementation dependencies as possible. In addition, according to various indexes which classify the popularity of existing programming languages [80, 81], Java is at the very top of the lists (either the first, or the second after the C language).

Furthermore, the engine also has to be written in Java. This is a constraint of the service delivery platform, since the entire project is implemented in Java and, in order to have this particular service module compatible with the OSGi framework, it needs to be implemented in Java.

### 3.3.2. Comparative Study on Code Generation Alternatives

Based on all the requirements mentioned, the possible options are: Jamon [82], Web-Macro [83], the Apache Velocity Project [84] and FreeMarker [85]. They are all general-purpose template engines, that is Java packages, class libraries for Java developers. They are not applications for end-users in themselves, but something that programmers can embed into their applications.

**Jamon**

Jamon is a text template engine for Java, which can be used to generate HTML, XML, or any text-based content. According to the specification [82], it is ideal for refactoring template-based user interface applications, and the templates are translated into Java classes with public methods, whose signatures reflect the arguments declared in the templates. This provides compile-time type-checking, as well as excellent performance. Jamon is licensed under a Mozilla Public License, and the latest processor version was released in March 2011, whereas the latest product release (version 2.3.0) took place in July 2007.
WebMacro

WebMacro is a stable template engine written in Java, now managed by a team at SourceForge. The current version of WebMacro is licensed under the GNU General Public License, and constitutes the starting point for the development of the Apache Velocity Project. Nevertheless, it is not as popular as its successor.

Apache Velocity

The Apache Velocity Project (formerly known as Jakarta Velocity) is an open-source project directed by the Apache Software Foundation. It provides a proprietary template language, the Velocity Template Language (VTL), which references objects defined in Java code. In fact, it uses Java classes, associated to a specific context, as the data model, or the description file, and the transformation process produces output in the format specified by the template. The documentation website provides a wide range of resources for developers and it also benefits from a very active community. The latest engine version (1.7) was released in November 2010.

FreeMarker

FreeMarker is a Java-based template engine which was initially developed for generating HTML Web pages and is still most often used for Servlet-based Web Application development. However, it has evolved as a general purpose engine, thus gaining more popularity. FreeMarker is licensed under a Berkley Software Distribution (BSD) License, and the latest stable release was launched in May 2011.

Comparing the four alternatives introduced above, the last two, Apache Velocity and FreeMarker, are the most popular. Due to the fact that they are all open-source and therefore developed by communities, the popularity also means more available resources and documentation necessary for implementation, more rapid bug fixing and better support. For these reasons, we chose to concentrate on these two. Whereas Velocity has been the leading template engine for a long time, FreeMarker has begun to overtake it recently, providing enhanced speed and sophistication. In order to perform this comparison, we chose three main criteria: power, ease of use and performance.

The power is a criterion which refers to the ability of the template engine to manipulate variable data while merging what is fetched from the data model and the template information. The ease of use discussed in this context consists of three main factors: ease of installation, consistency of the API and the quality of the documentation provided. The engine performance is the speed of execution, and is measured in the time which is
needed to generate a source code file based on a given template.

1. The Power

As far as the power is concerned, there are numerous features both engines have. For instance, they allow iteration and conditional processing, provide support for macros and for including other templates. In addition, they can both parse included templates, but they will not execute embedded Java code. Nevertheless, Velocity offers the opportunity to execute embedded Java code, if the objects are available in the context supplied to Velocity. In general, Velocity is less strict than FreeMarker, which can sometimes be seen as an advantage, but can also turn into a disadvantage, depending on the context. For example, FreeMarker does not allow references to non-existing properties, and will generate a null pointer exception, whereas Velocity can easily accept that.

2. The Ease of Use

The second criterion is the ease of use. Both FreeMarker and Velocity are easy to install; Velocity also depends on a number of additional libraries, but these are included in the distribution and should not cause any problems. Moreover, their APIs are easy to use and provide support for resource loaders. These classes enable the loading of templates and other resources from a number of locations like the file system or HTTP servers. One thing which differentiates the two template engines is the conversion tool offered by FreeMarker, which assists in converting Velocity templates to FreeMarker templates. Needless to say, this can save a great deal of time when the need to migrate projects from one to another arises.

3. The Performance

Thirdly, the performance was evaluated by van Bergen, who attempted to cover most possible usage scenarios for a template engine [86], including tests which output thousands of source code documents. The conclusions drawn were that both template engines evaluated are rather fast, and their performance is different only when extremely extensive templates are used. In those cases, FreeMarker seemingly performs better.

3.3.3. The Code Generation Option Chosen for the Project

Considering the comparison made in Section 3.3.2, the immediate conclusion drawn is that the Apache Velocity Project is the most suitable solution for automatically generating source code to be used on the service delivery platform.

To begin with, it meets the initial requirements of the project: it is both open-source and cross-platform. Furthermore, it is written in Java and is therefore easy to integrate with the rest of the existing code for the SDP prototype. Being a general-purpose tem-
plate language, the output consisting in the source code generated can be in any lan-
guage. In this regard, our option is Java due to its popularity and perfect compatibility
with OSGi.

As far as the power is concerned, Apache Velocity provides all the required features
for this implementation: it allows iteration and conditional clauses in the template, and
these are essential features for our scope. Moreover, unlike FreeMarker, it allows much
better template code flexibility, which is one of the major aspects which differentiate the
two and recommend Velocity.

Additionally, it is easy to use and, considering its popularity among the developer
communities, there are vast resources and documentation available. Also, the support
offered by the Apache community is the most notable among the different alternatives
studied.

Regarding the performance, although FreeMarker is faster at generating source code
than Apache Velocity when extensive templates are utilized, Apache Velocity provides
a much simpler and straightforward language. Similar to the case of the SDL, we seek
simplicity as long as the solution provides all the necessary features and, in this case,
Velocity appears to be the best choice.

Finally, Apache Velocity is orthogonal to any given DSL and, moreover, this template
generator attempts to maintain independence from any domain model.
Chapter 3 introduced the theoretical background of the project, including the presentation of the new concept, the results of the evaluation studies on description languages for the RESTful Internet or telecom services, and on the automatic code generation alternatives. The main decisions made were to use an XML-based language for service descriptions, and the Apache Velocity Project for source code generation. In this chapter, we demonstrate how the practical implementation was made. Firstly, a general overview of the technical implementation is provided. Secondly, REST services calls are presented, and the definition for the SDL is given. Then, the third section describes the details of the template engine and the template used in the project, and the fourth section shows how the integration with the OSGi framework is done.

Having established that a template-driven approach meets the needs of this research work, the technical design of the project is highly based on the template-driven mechanism depicted in Figure 3.3. Therefore, a data model and a template act as input for an engine, which generates the source code. Figure 4.1 illustrates the design of the technical implementation.

The data model consists of an XML file with a specific structure, which is defined in detail in Section 4.2. This includes the information about the RESTful service, such as the location of its resource and types of parameters.

The template is a file written in the proprietary language of the Apache Velocity project, the Velocity Template Language. The typical extension for such files is .vm, and the template written for this thesis is explained in Section 4.3.

The template engine is implemented using the Apache Velocity libraries, and consists of a Simple API for XML (SAX) parser, which extracts the relevant information from the data model, and the code generator itself.

The target is represented by Java files (Java classes) which are automatically generated by the template engine. These can be archived in the form of a .jar file.

In order to exemplify how the concept works in practice, we chose to use the Google
The essential aim of the implementation of the concept developed in this Master’s Thesis work is to prove that the theoretical fundament works in practice. For this purpose, we need a RESTful service description, based on which the target files are semi-automatically generated.

ProgrammableWeb [87] provides a vast variety of API resources for all types of services. In addition, the most important players on the market of services, such as Yahoo or Google, also expose the interfaces for developers to use them. The main criteria for choosing a specific REST service for this project were the need and utility of such a service on the SDP, the technical documentation and support available, and the popularity. Having considered all these, we decided to implement a use case by calling a RESTful Web service from Google, which belongs to the Google Language API Family.

The Google Language API Family aims at enabling users to communicate more easily across the globe, and consists of four main tools: Google Translate, Google Transliterate,
Google Virtual Keyboard and Google Diacritize. Google Translate is a tool which can be highly useful on the SDP, since it enables dynamic text translations between thousands of language pairs. In addition, it benefits from extensive documentation, and its popularity contributes to its being supported by large communities, including several which are not related to the official Google one. The latest available version of the Google Translate API is version 2, with the Google status "In Labs". Needless to say, a translation service can be mashed-up with numerous others in order to deliver new value in the form of various applications developed on the service delivery platform.

Being a REST service, Google Translate works by sending an HTTP GET request to its URI, which has the format: \texttt{https://www.googleapis.com/language/translate/v2?parameters}. The compulsory query parameters are: the API key, the target language and the source text string, and a list of optional parameters is also available.

There are two main possibilities to call a RESTful service: either using a hard-coded approach, or using one of the frameworks available.

**The Simple HTTP Request Approach**

When the Simple HTTP Request approach, depicted in Figure 4.2, is used, the entire process consisting of connection commands is hard-coded. Therefore, the location of the resource is saved in a private static string, the address of the server is provided and then the initial connection is set up, specifying that the request method is GET. Then, the result coming from the server is read in a BufferedReader object, and the connection is closed in the end.

It is important to observe one aspect here: the compulsory parameters of the RESTful service (the version, string to be translated and language pair) are included in the serverAddress. In this example, the string is “hello world”, and it has to be translated from English (en) to Romanian (ro).

**The Framework Approach**

A more elegant way, which also necessitates less programming effort, is to call a REST service by using one of the frameworks available for REST operations. In this thesis work, we chose the GlassFish Jersey framework (the Client part), which is more REST-oriented than the simple HTTP communication, and also provides support for typical formats used in RESTful interactions, such as JSON or XML. Nevertheless, in order to use the specific Jersey methods, several libraries need to be included in the project: \texttt{jersey-client.jar}, \texttt{jersey-core.jar} and \texttt{jsr311-api.jar}. All these can be found in the \texttt{jersey-archive-1.1.4.1.zip} file, downloadable from the website of the GlassFish project [89].

The same example for Google Translate, implemented using Jersey, is shown in Fig-
public class SimpleHttpRequest {
    private static String googleTranslationService =
        "http://ajax.googleapis.com/ajax/services/language/translate";
    /**
     * @param args
     */
    public static void main(String[] args)
    {
        HttpURLConnection connection = null;
        BufferedReader rd = null;
        StringBuilder sb = null;
        String line = null;
        URL serverAddress = null;
        try {
            serverAddress = new URL(googleTranslationService +
                "?v=1.0&q=hello%20world&&langpair=en%7Cro");
            connection = null;
            connection = (HttpURLConnection)serverAddress.openConnection();
            connection.setRequestMethod("GET");
            connection.setDoOutput(true);
            connection.setReadTimeout(10000);
            connection.connect();
            rd = new BufferedReader(new
                InputStreamReader(connection.getInputStream()));
            sb = new StringBuilder();
            while ((line = rd.readLine()) != null
                sb.append(line + 'n');
            System.out.println(sb.toString());
        } catch (MalformedURLException e)
            e.printStackTrace();
        catch (ProtocolException e)
            e.printStackTrace();
        catch (IOException e)
            e.printStackTrace();
        finally {
            connection.disconnect();
            rd = null;
            sb = null;
            connection = null;
        }
    }
}

Figure 4.2.: Calling a REST service with Simple HTTP Request

Figure 4.3: The resource URI is also stored in a String object, and the translateString method takes as parameters the string which needs to be translated (sourceString), the language of the string (sourceLanguage), and the language to which the translation is made (targetLanguage). It can be easily noticed that the implementation is more specialized in this example, where the Google Translation Service is manipulated as a web resource. Then the three compulsory parameters are allocated values and the response is returned. In order to verify the translations resulted, a simple call such as translateString("hello world", "en", "ro") can be made, and the “hello world” string will be translated from English to Romanian. The typical result of the REST call is given as a JSON response:
Consequently, the returned string “salut lume” represents the translation of the input provided. Moreover, the GlassFish Jersey framework, through its Client, provides support for parsing such JSON objects, thus turning them into proper Java objects. An alternative to using the Jersey libraries for parsing the JSON response is the JSON-SIMPLE library offered by Google.

The GlassFish Jersey framework provides support for parsing JSON objects, thus turning them into proper Java objects. An alternative to using the Jersey libraries for parsing the JSON response is the JSON-SIMPLE library offered by Google.

```
public class TranslationClient {

    private static String googleTranslationService =
    "http://ajax.googleapis.com/ajax/services/language";

    private static String translateString(String sourceString, String
    sourceLanguage, String targetLanguage) {

        ClientConfig config = new DefaultClientConfig();
        Client c = Client.create(config);
        WebResource r = c.resource(googleTranslationService);
        MultivaluedMap<String, String> params = new MultivaluedMapImpl();

        params.add("v", "1.0");
        params.add("q", sourceString); // string to translate
        params.add("langpair", sourceLanguage+"|"+targetLanguage);
        String response =
        r.path("translate").queryParams(params).get(String.class);
        return response;
    }
}
```

```
Figure 4.3.: Calling a REST service with GlassFish Jersey
```

Essentially, it is more REST-oriented and simpler to make the HTTP requests required for accessing RESTful services by using GlassFish Jersey, and this is the main reason why we also chose this approach in this project.

### 4.2. The Definition of the Service Description Language

To meet the requirements of the project, the XML-based service description language was designed to be as simple as possible in describing the REST services resources and features, and also self-sufficient to encapsulate all the details necessary for the template engine to automatically generate source code. It is a syntactic language, resource-centric, and it contains the elements described by the XML Schema (or XSD) document in Figure 4.4.

The first line of the description consists of the typical header for XML files:

```
<xml version="1.0" encoding="UTF-8"/>
```

The root of the instance file is the `<Content>` element, which does not require any attributes. The specification only allows one such element for the entire description.
The next type of element in the XML hierarchy is the <Class> element. A description file can have an indefinite number of such elements, and each of them will have a corresponding automatically generated Java class. Therefore, it is recommended that a <Class> element is defined for each different service included in the description file. The <Class> element must have a name attribute, whose value (a string) will also be included in the names of the generated files.

The <Class> element can include as many <Parameter> elements as necessary, according to the REST service specification. They must have three compulsory attributes: name, type and value. The name attribute provides an identifier for the <Parameter>, which has to coincide with the name recognized and imposed by the REST service. The type attribute shows the type of the <Parameter> element, and also has to respect the constraints of the REST service. The value indicates the fact that the <Parameter> element has a predefined value. Most often, this will be used for specifying the URI to which the REST request is sent. If the <Parameter> element does not have any predefined value, this is shown as void: "".

A simple example of a service description file is presented in Figure 4.5.
The `<Class>` element defines the service which is described. Therefore, this is a translation service, which has the name “Translate”, given by the `name` attribute. The REST service has four `<Parameter>` elements: "URI", "q", "v" and "langpair", and all of them are of the type "String". These names are the ones recognized by the service provider and mentioned in the service API.

The "URI" `<Parameter>` provides the location of the resource for the Google service, which is called in order to perform the translation. The "q" `<Parameter>` represents the string which needs to be translated, the "v" `<Parameter>` indicates the version of the service to be used, and the "langpair" `<Parameter>` specifies the source language of the text given in "q" and the target language to which the translation is made. The format of this language pair is established by the service provider (Google), and is the following: “sourceLanguage|targetLanguage”. This XML file can be extended to include also optional parameters for the "Translate" service, or additional services with complete parameters definitions. The language specification does not impose any limitations in this respect.

Therefore, the employee of the CSP, who analyses various APIs of Internet and telecom RESTful services, extracts the necessary information from the textual descriptions and fills in such a file with the appropriate information. The names of the service parameters (given by the `name` attribute of the `<Parameter>` element) always need to be consistent with those provided in the original service API.
4.3. The Template Engine

According to the results of the comparative study on code generation alternatives conducted in Section 3.3.2, the most appropriate way to automatically generate Java code from description files is using the Apache Velocity Project. In this section, we show how the code generation concept was implemented in practice.

The Engine Implementation

To begin with, it is important to explain how the template engine was constructed using libraries provided by the Apache Velocity Project. A relevant method written in this respect is illustrated in Figures 4.6 and 4.7.

The public static void start(String modelFile, String templateFileService, String templateFileInterface, String templateFileClient, String metaProvider, String metaClient, String pomService, String pomProvider, String pomClient)

Java method takes as input nine parameters: modelFile is the data model, that is the XML-based service description file described in the previous section, and the others are template files used in the project. After loading and parsing the data model, an array of classes is built. For example, based on the description depicted in Figure 4.5, this will only contain the “TranslateImpl” class but, in other cases, it can contain an indefinite number of different classes representing various RESTful services.

The next step is the essential one for the code generation process. An Apache-specific GeneratorUtility object is created and then, for each class, a VelocityContext object. For each of these, the parameters are maintained in the ArrayList attrs, and the total number of parameters is counted in the nr_atr variable.

Velocity provides a dedicated object for manipulating templates (the Template object), and also has special methods for loading such files (Velocity.getTemplate(templateFileService), for instance). Based on the data fetched from the data model and templates, the corresponding Java files for the classes in the XML document are generated: BufferedWriter writer = new BufferedWriter(new FileWriter(cl.getName() + "Impl.java")), for example, for the service provider class. As it can be observed, the automatically generated files have names according to the information in the service description, and also considering the constraints imposed by OSGi for the configuration files.

The Templates

In Figure 4.6, it can be observed that a total of eight Velocity templates are utilized in this thesis work. TemplateFileInterface (Figure A.2) represents the template used for generating the main interface implemented; based on the TemplateFileService (Figure 4.8),
public static void start(String modelFile, String templateFileService, String templateFileInterface, String templateFileClient, String metaProvider, String metaClient, String pomService, String pomProvider, String pomClient) throws Exception {

    // Load the model
    FileInputStream input = new FileInputStream(modelFile);
    xmlReader.parse(new InputSource(input));
    input.close();
    classes = cdImporter.getClasses();

    // Generate classes
    GeneratorUtility utility = new GeneratorUtility();
    for (int i = 0; i < classes.size(); i++) {
        VelocityContext context = new VelocityContext();
        ClassDescriptor cl = (ClassDescriptor) classes.get(i);
        context.put("class", cl);
        context.put("utility", utility);
        ArrayList attrs = cl.getAttributes();
        for (int j = 0; j < attrs.size(); j++) {
            AttributeDescriptor at = (AttributeDescriptor) attrs.get(j);
            context.put("nr_atr", attrs.size());
        }
        Template template1 = Velocity.getTemplate(templateFileService);
        Template template2 = Velocity.getTemplate(templateFileInterface);
        Template template3 = Velocity.getTemplate(templateFileClient);
        Template template4 = Velocity.getTemplate(metaProvider);
        Template template5 = Velocity.getTemplate(metaClient);
        Template template6 = Velocity.getTemplate(pomService);
        Template template7 = Velocity.getTemplate(pomProvider);
        Template template8 = Velocity.getTemplate(pomClient);

        BufferedWriter writer1 = new BufferedWriter(new FileWriter(cl.getName() + "Impl.java"));
        BufferedWriter writer2 = new BufferedWriter(new FileWriter(cl.getName() + ".java"));
        BufferedWriter writer3 = new BufferedWriter(new FileWriter(cl.getName() + "Client.java"));
        BufferedWriter writer4 = new BufferedWriter(new FileWriter(cl.getName() + "MetadataProvider.xml"));
        BufferedWriter writer5 = new BufferedWriter(new FileWriter(cl.getName() + "MetadataClient.xml"));
        BufferedWriter writer6 = new BufferedWriter(new FileWriter(cl.getName() + "POMService.xml"));
        BufferedWriter writer7 = new BufferedWriter(new FileWriter(cl.getName() + "POMServiceProvider.xml"));
        BufferedWriter writer8 = new BufferedWriter(new FileWriter(cl.getName() + "POMServiceClient.xml"));

        template1.merge(context, writer1);
        writer1.flush(); writer1.close();
        template2.merge(context, writer2);
        writer2.flush(); writer2.close();
        template3.merge(context, writer3);
        writer3.flush(); writer3.close();
    }
}

Figure 4.6.: The template engine - Code sample. Part I.
the Java class for the service provider is generated, and based on the TemplateFileClient (Figure A.5), the translate service client is generated. Moreover, two template files are needed for the OSGi metadata files, and these are metaProvider (Figure A.3), for the service provider, and metaClient (Figure A.6), for the service client, respectively. The last three templates included in the method definition, pomService (Figure A.1), pomProvider (Figure A.4) and pomClient (Figure A.7), are used for automatically generating the pom.xml OSGi configuration files, including the instructions for building the bundles. These resulting specific OSGi files are described in detail in Section 4.4.

As an example, in Figure 4.8, the template for the translate service provider included in the project implementation is presented. It is written in the Velocity Template Language, the proprietary Velocity language. The first aspect which can be recognized is the list of Glassfish Jersey libraries imported: Client, WebResource, ClientConfig, DefaultClientConfig and MultivaluedMapImpl; as discussed in Section 4.1, these are needed for calling the remote REST service resource. The name of the automatically generated class is the same as the name of the class specified in the XML file: $class.Name, and the “$” symbol signals a Velocity variable in the template.

Three constants are also defined (“URI”, “Action” and “Impl”), and they serve the purpose of forming class, variable and method names, by concatenation. Subsequently, the attributes are treated one by one and the appropriate methods are generated for each, depending on their specifics. One of the most interesting parts of the template is that referring to the generation of the translation method:

\[
\text{private static String } \$\text{class.Name}\$\text{action (String... arguments). The result, after it is interpreted by the engine, is: private static String TranslateAction (String... arguments).}
\]
import java.util.*;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.config.ClientConfig;
import com.sun.jersey.api.client.config.DefaultClientConfig;
import com.sun.jersey.core.util.MultivaluedMapImpl;
import javax.ws.rs.core.MultivaluedMap;
import org.json.simple.*;

public class $class.Name$implementation implements $class.Name {

    private static String $class.Name$uri = "$att.Value";

    private $att.Type $att.Name;
    public $att.Type get$utility.firstToUpperCase($att.Name)() {
        return this.$att.Name;
    }

    public void set$utility.firstToUpperCase($att.Name)($att.Type $att.Name) {
        this.$att.Name = $att.Name;
    }

    private static String $class.Name$action ( String... arguments) {
        ClientConfig config = new DefaultClientConfig();
        Client c = Client.create(config);
        WebResource r = c.resource($class.Name$uri);
        MultivaluedMap<String, String> params = new MultivaluedMapImpl();
        if (arguments.length == $nr_atr - 1) {
            for (int i = 0; i < arguments.length; i++) {
                params.add($att.Name,arguments[i++]);
            }
        } else {
            System.out.println("The number of arguments for the method must be equal to the number of attributes in the data model - 1 (the URI)");
            String class_name = "$class.Name";
            String response = r.path(class_name.toLowerCase()).queryParams(params).get(String.class);
            return response;
        }
    }
}

Figure 4.8: The template for the translate service provider
This method takes an indeterminate number of arguments, since a service can provide various ways in which it can be called: using a set of compulsory parameters, but maybe also additional optional ones. Moreover, this approach ensures the fact that the solution works for a wide variety of RESTful services, independent of the number of parameters they require. Moreover, the generated TranslateAction method forms the REST request: `params.add("$att.Name",arguments[i+1])`, and manages the REST calls explained in Section 4.1. Therefore, in the example used in this chapter, the "q", "v" and "lang-pair" parameters are added as arguments and sent to the resource over the Internet. The response received from the REST service which was called is a string.

### 4.4. OSGi Integration

The result of the automatic code generation service needs to be integrated as a bundle in the service delivery platform, in order to become valuable for the developers who use the Internet and telecom resources to build new applications. To achieve this goal, as explained in Section 2.1.3, the code and configuration files automatically generated by the template engine are encapsulated in an OSGi bundle.

This can be done in various ways, and for this project the approach utilized is based on Apache Felix injected Plain Old Java Object (iPOJO) [90]. The main aim of iPOJO is to simplify service-oriented programming on OSGi frameworks, by transparently managing the dynamics of the environment, as well as other non-functional requirements. A significant advantage of iPOJO is that it enables developers to separate the functional code, represented by the service itself, from the non-functional code, consisting in the dependency management mechanisms, service provision, configuration and others. In this context, a service component is seen as an object that implements a given service interface embodied as a Java interface.

According to the Apache project specification, in the iPOJO model, a component describes service dependencies, provided services and callbacks, and this information is recorded in the metadata of the component. The second important concept in iPOJO is the component instance, which is a special version of the component. By merging component metadata and instance configuration, the iPOJO runtime is able to manage the component, that is manage its life-cycle, inject required services, publish provided services and discover needed services.

As far as the technical details are concerned, iPOJO relies on the OSGi R4.1 framework and the J2ME Foundation 1.1 Java Virtual Machine. Therefore, it can be used on Apache Felix, Eclipse Equinox, or any OSGi implementation compliant with the OSGi R4.1 specification. In this thesis work, we use it with Apache Felix, and the ways in which the translation OSGi service is published, required and different components are
activated or deactivated using life-cycle callbacks are presented below.

Essentially, the implementation of this OSGi integration comprises two components: one providing a translation service, and one requiring translation services; these are packaged into three bundles using Apache Maven [91], a software tool for build automation. The translate.service bundle contains the service interface, the translate.impl bundle contains a component implementing the service, and the translate.client bundle contains the consumer component.

The first step in the bundling process is to set up the environment by installing Maven (mvn clean install) - the version used in this project is Maven 3.0.3.

The Translate Service

This project only contains the Translate interface, which was automatically generated, as illustrated in Figure 4.9.

```java
public interface Translate {
    String TranslateAction(String... arguments);
}
```

Figure 4.9.: The translate service interface

By using the maven-bundle-plugin and the instructions contained by the automatically generated pom.xml file located in the project directory and presented in Appendix A, Figure A.8, the project can be built issuing the Maven command inside the project directory: mvn clean install. Upon success, the translate service component JAR file is installed in the local Maven repository, and a copy of the bundle JAR file will also be present in the target directory inside the project directory.

The Translate Service Provider

The implementation of the translate service is the TranslateImpl Java class which was automatically generated based on the data model and one of the template files, and it implements the Translate interface. In the sample code presented in Figure 4.10, for relevance purposes, only one method is depicted, but the actual implementation of the project contains several methods which can be used by developers on the platform.

To manage the component, iPOJO needs some metadata to understand that this component provides the Translate service introduced above. The generated iPOJO metadata file is at the root of the translate.impl project (“metadata.xml”), and it contains the meta-
data shown in Figure 4.11

```java
public class TranslateImpl implements Translate {

    private static String TranslateURI = 
        https://ajax.googleapis.com/ajax/services/language;

    private static String TranslateAction (String... arguments) {
        ClientConfig config = new DefaultClientConfig();
        Client c = Client.create(config);
        WebResource r = c.resource(TranslateURI);
        MultivaluedMap<String, String> params = new MultivaluedMapImpl();

        if (arguments.length == 4 - 1) {
            for (int i = 0; i < arguments.length; i++) {
                params.add("q", arguments[i++]);
                params.add("v", arguments[i++]);
                params.add("langpair", arguments[i++]);
            }
        } else
            System.out.println("The number of arguments for the method must be 
                equal to the number of attributes in the data model - 1 (the URI)");

        String class_name = "Translate";
        String response =
            r.path(class_name.toLowerCase()).queryParams(params).get(String.class);
        return response;
    }
}
```

Figure 4.10.: The implementation for the translate service provider

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ipojo
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="org.apache.felix.ipojo
    http://felix.apache.org/ipojo/schemas/CURRENT/core.xsd"
    xmlns="org.apache.felix.ipojo">

    <component classname="ipojo.sdp.translate.impl.TranslateImpl"
        name="TranslateProvider">
        <provides/>
    </component>

    <instance component="TranslateProvider" name="TranslateService" />
</ipojo>
```

Figure 4.11.: The metadata.xml file for the translate service provider
The `classname` attribute informs iPOJO about the implementation class of the component; here, it is `ipojo.sdp.translate.impl.TranslateImpl`. In addition, the `provides` element signals that iPOJO must manage the publishing of a service. When it does not contain an interface attribute, as here, iPOJO will expose all implemented interfaces of the component as a service. The `instance` element tells iPOJO to create an instance of the component when the bundle is started. Similar to the case of the previous project, a “pom.xml” including the build instructions is needed. The file used for the translate service provider is presented in Appendix A, Figure A.9. The first part of the document contains important information for Maven, whereas the rest of it includes the bundle configuration. The service provider bundle exports the package of Translate interface, and then the project can be built with `mvn clean install`.

The Translate Service Client

The `translate.client` project represents the translate service consumer or client, and is reproduced in Figure 4.12.

The translate service client creates a thread that periodically invokes the available translate services. The thread starts when at least one service provider is present using iPOJO’s callback mechanism. It is interesting to observe that iPOJO manages service synchronization too. Therefore, the service invocations do not require synchronization blocks, since the synchronization is maintained on a per thread basis, where each method that accesses a service is instrumented to attach the given service instance to the thread, so that the thread will continue to see the same service instances even across nested method invocations. The thread will not see different service instances until it completely exits from the first method it entered which used a service. The component provides two callback methods for its activation and deactivation, `starting()` and `stopping()`, respectively.

The iPOJO metadata file describing the component is “metadata.xml”, and contains the metadata in Figure 4.13.

Lastly, the build instructions are included in the pom.xml file, generated automatically, illustrated in Appendix A, Figure A.10. After building the service client bundle JAR file, Maven installs it into a local repository; consequently, to resolve compilation dependencies, it always looks in the local repository to find required JAR files.
package ipojo.sdp.translate.client;
import ipojo.sdp.translate.Translate;

public class TranslateClient implements Runnable {
    private static final int DELAY = 10000;
    private Translate[] m_translate;
    private boolean m_end;

    public void run() {
        while (!m_end) {
            try {
                invokeTranslateServices();
                Thread.sleep(DELAY);
            } catch (InterruptedException ie) {} 
        }
    }

    public void invokeTranslateServices() {
        for (int i = 0; i < m_translate.length; i++) {
            System.out.println(m_translate[i].TranslateAction("I am in Helsinki", "1.0", "en|fi");
        }
    }

    public void starting() {
        Thread thread = new Thread(this);
        m_end = false;
        thread.start();
    }

    public void stopping() {
        m_end = true;
    }
}

Figure 4.12.: The implementation for the translate service client

Running the Project

In order to run the project, Apache Felix is used. After starting the Felix framework with `java -jar bin/felix.jar`, the translate service bundle, the translate service provider and the translate client that were created are installed:

```
start file:../translate.service/target/translate.service-1.0.0.jar
start file:../translate.impl/target/translate.impl-1.0.0.jar
start file:../translate.client/target/translate.client-1.0.0.jar
```

By starting the Translate service provider bundle, the client component will be activated and the methods included in the automatically generated Java class will be executed.
<?xml version="1.0" encoding="UTF-8"?>
<ipojo
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="org.apache.felix.ipojo
   http://felix.apache.org/ipojo/schemas/CURRENT/core.xsd"
xmlns="org.apache.felix.ipojo">
   <component classname="ipojo.sdp.translate.client.TranslateClient">
      <requires field="m_translate" />
      <callback transition="validate" method="starting" />
      <callback transition="invalidate" method="stopping" />
      <properties>
         <property field="m_name" name="translate.name" />
      </properties>
   </component>

   <instance component="ipojo.sdp.translate.client.TranslateClient">
      <property name="translate.name" value="clement" />
   </instance>
</ipojo>

**Figure 4.13.** The metadata.xml file for the translate service client
Service delivery platforms are the ideal solution for communication service providers to expose the interfaces of telecom and Internet services, and make them available for the communities of developers to design and implement new applications, thus delivering new value for end-users [9, 10, 14, 15]. In order to turn such SDPs into popular platforms, it is necessary that they comprise services which have all the functionalities required by developers [4, 57]. Moreover, the aim of the project was to enable this in a service-oriented manner. Nevertheless, achieving this goal was rather challenging, since most service providers choose to describe their services textually, most often in the form of Web pages.

In this study, having identified the need of automation on the SDPs, our main objective was to clarify how external Internet and telecom services can be semi-automatically integrated in a service delivery platform for a Telecom operator. To fulfill this, a concept was first defined, which conducted to the need of investigating both service description languages and code generation alternatives. The research methods utilized were the literature review and the design science method.

The literature review consisted of two parts. Firstly, it was used to collect the essential information about different service description languages (WADL, WSDL 2.0, RIDDL, WRDL, WDL, SMEX-D, NSDL, RESEDEL and USDL), which were then compared according to nine criteria. Based on the results obtained, the appropriate service description language for this project was defined. An XML-based, syntactic and REST-oriented language was chosen to meet the needs of this research work, and its comprehensive definition was provided. This way, the first objective - conducting a comparative study on SDLs and defining the SDL to be used for this implementation work - was successfully achieved.

Secondly, the literature review method was used to investigate different alternatives for automatic code generation. Starting from the premise that generative programming is the key solution for the requirements of the SDP, various related options were analysed, focusing on template engines (Jamon, WebMacro, FreeMarker and the Apache Velocity Project). The result of this study was that Apache Velocity is the best alterna-
tive to be implemented. Consequently, the second objective - conducting a comparative study on different possibilities for automatic code generation and choosing the most appropriate option - was achieved.

The third objective, defined according to the specifications of the service delivery platform, was to research how the results of the automatic code generation process can be integrated in the SDP with the OSGi framework. To achieve this goal, an OSGi bundle encapsulating the new service was created, and the corresponding steps and implementation details were presented.

The fourth objective of the study was to assess the generality of the solution implemented using the design science method, in this thesis work. For this purpose, we first introduce the factors which define the generality of the project: provider independence, service-orientation, output format, and number of service parameters.

As far as the provider independence is concerned, the semi-automatic code generation service implemented works for any Internet or telecom REST service provider, which exposes the interfaces of the services. These can be defined in any textual format, as long as they can be understood by the employee of the CSP who fills in the description files.

The service orientation is a criterion referring to the orientation of the services described in the XML-based SDL. The aim of this research work was to bring the functionalities of RESTful services to the SDP, and the orientation is therefore limited to REST services.

Next, the output format provided by the service resource is very open, since the implementation works for any textual (string) response received, irrespective of it being JSON or of another type, thus achieving a high degree of generality.

Lastly, as far as the number of service parameters is concerned, the service designed in this thesis is truly general, allowing any number of parameters. The only constraint to this is that the first one listed in the description file has to be the service resource location (URI), and the others, both compulsory and optional, can follow in any order.

The limitations of this research study are related to the types of services supported by the implementation. Only RESTful services were considered, and the solution provided does not treat SOAP or SIP services. Therefore, the document exclusively discusses resource-centric, REST services, and the main argument for this choice is their increasing popularity among both service providers and communities of developers nowadays.

In conclusion, the objectives initially defined were all achieved, and the generality of the solution can be assessed as high, in the context of RESTful services.
Conclusion and Future Work

This Master’s Thesis introduced a new concept for the semi-automatic service integration of telecom and Internet services in a service delivery platform, and presented the design and important implementation details of the project.

In the process of designing and developing the service, some limitations related to the existing alternatives for describing RESTful services were encountered. This led us to defining a proprietary description language, which was used in this research work. By contrast, for the automatic source code generation, a rather popular template-driven option was chosen: the Apache Velocity Project.

Generally, the solution implemented has a significant impact on the way external telecom and Internet services are imported in an SDP, in a service-oriented fashion. In summary, this work contributes to the knowledge in the field of automatic code generation for service integration on service delivery platforms.

As far as the future work is concerned, in the near future, the extension of the specified service description language can be considered. This should be done to achieve even a higher degree of generality, by describing any type of telecom or Internet service, not only RESTful ones. For example, it could be compatible also with SOAP and SIP services. In addition, a method to parse the output received from the service providers could be investigated. At the moment, the project provides an option for parsing JSON responses, but it is limited to these.

For the far future, the automation of the project can be enhanced by using semantics. Such an approach could be applied in order to reduce, or even substitute the human involvement in the first stage of the working mechanism. Therefore, the service descriptions would be annotated with semantics, and this would greatly contribute to the overall automation of the service integration process in the SDP.
Implementation Details

This appendix contains the templates required by the project and the automatically generated pom.xml files used for building the OSGi bundle for the translate service.

```xml
#set( $serv = ".service" )

<project>
    <modelVersion>4.0.0</modelVersion>
    <packaging>bundle</packaging>
    <groupId>ipojo.sdp</groupId>
    <artifactId>$class.Name$serv </artifactId>
    <version>1.0.0</version>
    <name>$class.Name Service</name>

    <build>
        <plugins>

            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>2.0.1</version>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName>
                        <Export-Package>ipojo.sdp.$class.Name</Export-Package>
                    </instructions>
                </configuration>
            </plugin>

        </plugins>
    </build>
</project>
```

Figure A.1.: POM file for the service - Template
public interface $class.Name {
    #set( $action = "Action" )
    String $class.Name$action (String... arguments);
}

Figure A.2.: Service interface - Template

<?xml version="1.0" encoding="UTF-8"?>
<ipojo
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="org.apache.felix.ipojo
        http://felix.apache.org/ipojo/schemas/CURRENT/core.xsd"
    xmlns="org.apache.felix.ipojo">
    <component classname="ipojo.sdp.$class.Name$implementation$class.Name$impl"
        name="$class.Name$prov">
        <provides />
    </component>
    <instance component="$class.Name$prov" name="$class.Name$serv" />
</ipojo>

Figure A.3.: Metadata for the service provider - Template
Figure A.4.: POM file for the service provider - Template
package ipojo.sdp.$class.Name.$cl;
import ipojo.sdp.$class.Name.$class.Name;

public class $class.Name$cl implements Runnable {
    private static final int DELAY = 10000;
    private $class.Name [] m_$class.Name;
    private boolean m_end;

    public void run() {
        while (!m_end) {
            try {
                invoke$class.Name$serv;
                Thread.sleep(DELAY);
            } catch (InterruptedException ie) {}
        }
    }

    public void invoke$class.Name$serv {
        for (int i = 0; i < m_$class.Name$len ; i++)
            System.out.println(m_$class.Name[i].$class.Name$action
                               ("","1.0","en|fi");
    }

    public void starting() {
        Thread thread = new Thread(this);
        m_end = false;
        thread.start();
    }

    public void stopping() {
        m_end = true;
    }
}

Figure A.5.: Service client - Template
Figure A.6.: Metadata for the service client - Template
Figure A.7.: POM file for the service client - Template
Figure A.8.: The pom.xml file for the translate service
Figure A.9.: The pom.xml file for the translate service provider
The pom.xml file for the translate service client

Figure A.10.: The pom.xml file for the translate service client
References


32. Hewitt, E. Java SOA Cookbook; OReilly, 2009.


34. Burke, B. RESTful Java with JAX-RS; O’Reilly, 2009.


46. Wang, Y.; Song, M.; Song, J. An extended distributed OSGi architecture for implementation of SOA. *Advanced Intelligence and Awareness Internet (AIAI 2010), 2010 International Conference on*, 2010; pp 416–419.


58. Blum, N.; Boldea, I.; Magedanz, T.; Staiger, U.; Stein, H. A Service Broker providing Real-

59. Takase, T.; Makino, S.; Kawanaka, S.; Ueno, K. Definition Languages for RESTful Web Ser-


2011.

63. Mangler, J.; Schikuta, E.; Witzany, C. Quo Vadis Interface Definition Languages? Towards
a Interface Definition Language for RESTful Services. 2009 IEEE International Conference on

64. Mangler, J.; Beran, P. P.; Schikuta, E. On the Origin of Services Using RIDDl for Description,
Evolution and Composition of RESTful Services. 10th IEEE/ACM International Conference on


accessed: June 2011.


68. Norm’s Service Description Language. [http://norman.walsh.name/2005/03/12/nsdl]
accessed: June 2011.


70. Simon, L.; Bansal, A.; Mallya, A.; Kona, S.; Gupta, G.; Hite, T. Towards a Universal Ser-
vice Description Language. International Conference on Next Generation Web Services Practices
(NWeSP), 2005.

71. Simon, L.; Mallya, A.; Bansal, A.; Gupta, G.; Hite, T. A Universal Service Description Lan-
pp xxxiii–856.

Service Description Language. IEEE 8th European Conference on Web Services (ECOWS), 2010;

for the Internet of Services: Requirements and First Developments. IEEE International Con-
ference on Services Computing (SCC), 2010; pp 602 –609.

74. Maleshkova, M.; Kopecky, J.; Pedrinaci, C. Adapting SAWSDL for Semantic Annotations of


77. Maleshkova, M.; Pedrinaci, C.; Domningue, J. Supporting the creation of semantic RESTful service descriptions. In The Open University’s Repository of Research Publications and Other Research Outputs; 2009.


