Project no. FP7-216556
Instrument: Integrated Project (IP)
Objective ICT-2007.1.2: Service and Software Architectures, Infrastructures and Engineering

Deliverable D.A1a Framework architecture (full lifecycle)

Keywords:
Service Level Agreement, Service-Oriented Infrastructure, Architecture, Models

Due date of deliverable: 31st July 2010
Actual submission to EC date: 10th September 2010
Start date of project: 1st June 2008
Duration: 38 months
Lead contractor for this deliverable: SAP
Revision: V.1.0 (8th September 2010)

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th>PU</th>
<th>Public</th>
<th>Yes</th>
</tr>
</thead>
</table>

SLA@SOI – FP7216556  Deliverable D.A1a – Framework Architecture  Page 1 of 88
## Document Status

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliverable Lead</td>
<td>Wolfgang Theilmann</td>
</tr>
<tr>
<td>Reviewer 1</td>
<td>Sam Guinea</td>
</tr>
<tr>
<td>Reviewer 2</td>
<td>Giovanni Falcone</td>
</tr>
<tr>
<td>PMT Reviewer</td>
<td>Ramin Yahyapour</td>
</tr>
<tr>
<td>Complete version submitted to reviewers</td>
<td>June 21, 2010</td>
</tr>
<tr>
<td>Comments of reviewer 1 received</td>
<td>July 4, 2010</td>
</tr>
<tr>
<td>Comments of reviewer 2 received</td>
<td>July 5, 2010</td>
</tr>
<tr>
<td>Revised deliverable submitted to PMT</td>
<td>July 25, 2010</td>
</tr>
<tr>
<td>PMT Approval</td>
<td>August 15, 2010</td>
</tr>
</tbody>
</table>

## Contributors

<table>
<thead>
<tr>
<th>Partner</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAP</td>
<td>Wolfgang Theilmann, Jens Happe, Tariq Ellahi</td>
</tr>
<tr>
<td>ENG</td>
<td>Francesco Torelli, Keven Kearney</td>
</tr>
<tr>
<td>TID</td>
<td>Juan Lambea, Beatriz Fuentes</td>
</tr>
<tr>
<td>XLAB</td>
<td>Miha Vuk</td>
</tr>
<tr>
<td>PMI</td>
<td>Sam Guinea</td>
</tr>
<tr>
<td>Intel</td>
<td>Andrew Edmonds, Michael Nolan</td>
</tr>
<tr>
<td>FZI</td>
<td>Franz Brosch</td>
</tr>
<tr>
<td>UDO</td>
<td>Costas Kotsokalis</td>
</tr>
</tbody>
</table>

## Notices

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law. Copyright 2009 by the SLA@SOI consortium.

* Other names and brands may be claimed as the property of others.

---

This work is licensed under a Creative Commons Attribution 3.0 License.
<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>29.04.2010</td>
<td>Jens</td>
<td>Added first draft of SCM</td>
</tr>
<tr>
<td>0.2</td>
<td>11.5.2010</td>
<td>Wolfgang</td>
<td>Added foundations &amp; building blocks</td>
</tr>
<tr>
<td>0.3</td>
<td>04.06.2010</td>
<td>Jens</td>
<td>Written text and reworked pictures for component interactions</td>
</tr>
<tr>
<td>0.4</td>
<td>15.6.2010</td>
<td>Wolfgang</td>
<td>Added Exec summary, conclusions; various fixes</td>
</tr>
<tr>
<td>0.5</td>
<td>15.6.2010</td>
<td>Miha Vuk</td>
<td>Added Monitoring</td>
</tr>
<tr>
<td>0.6</td>
<td>18.6.2010</td>
<td>Jens Happe</td>
<td>Detailed Interaction Flows</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Minor Corrections</td>
</tr>
<tr>
<td>0.7</td>
<td>21.6.2010</td>
<td>Beatriz Fuentes</td>
<td>Adjustment (section 5.3.3)</td>
</tr>
<tr>
<td>0.8</td>
<td>19.07.2010</td>
<td>Wolfgang/Jens</td>
<td>Addressed reviewer comments</td>
</tr>
<tr>
<td>0.9</td>
<td>10.8.2010</td>
<td>Jens</td>
<td>Check before PMT – Review</td>
</tr>
<tr>
<td>1.0</td>
<td>8.10.2010</td>
<td>Wolfgang</td>
<td>Included PMT review comments</td>
</tr>
</tbody>
</table>
Executive Summary

This document describes the innovative architecture for the comprehensive SLA management framework created during the second year of the European research project SLA@SOI. It serves as formal deliverable D.A1a for reporting the work progress of work package A1 at project month 26. It is complemented by the deliverable D.A1b which provides the actual implementation of the SLA framework.

This architecture directly responds to the mission of SLA@SOI: “to deliver and showcase an innovative open SLA Management Framework that provides holistic support for service level objectives - enabling an open, dynamic, SLA-aware market for European service providers. SLAs will be managed autonomously throughout the complete service lifecycle, spanning the entire services stack from the business layer through to infrastructure. Arbitrary domains will be supported, as demonstrated by evaluations in wide-ranging, grounded, use cases”. Furthermore it integrates the contributions on the four top-level technical objectives of (1) Consistent SLA-management framework, (2) Adaptive SLA-aware infrastructure, (3) Engineering predictable service-oriented systems, and (4) Comprehensive business management suite for e-contracting into one consistent architecture.

The reference architecture definition was driven by the requirements of four industrial use cases but also by other internal and project-external stakeholders.

The reference architecture represents a key innovation of SLA@SOI as it realizes the first comprehensive architecture of a consistent SLA-management framework. Four main novelties can be highlighted: (1) the architecture supports multi-layered SLA management where SLAs can be composed and decomposed along functional & organizational domains; (2) it supports arbitrary service types (business, software, and infrastructure) and SLA terms; (3) the architecture covers the complete SLA and service lifecycle with consistent interlinking of design-time, planning and run-time management aspects; (4) the actual implementation supports flexible deployment setups, where actual components can be flexibly selected, extended and connected and where founding data models can be extended.

This document provides a complete overview of the framework architecture including a high-level overview, a detailed discussion of relevant foundational concepts, a description of the modelling foundation i.e. the most important metamodels that are shared between different components of the architecture, and the actual architecture overview with its building blocks, components and interactions.

The technical architecture has been successfully implemented into an SLA management software framework. Details about the technical/scientific evaluation of the SLA framework can be found in deliverable D.B1a. Furthermore, the software framework has been successfully applied against four industrial use cases in the domains of ERP hosting, Enterprise IT management, Telco Service Aggregation and eGovernment.

The reference architecture has been successfully published in a journal [6]; a book chapter is currently in preparation [22].
# Table of Contents

1. Introduction ...................................................................................... 11
2. Contribution Overview ........................................................................ 12
   2.1 Key Innovations ........................................................................... 12
   2.2 Framework Contributions .............................................................. 12
   2.3 Task-level activities ...................................................................... 13
3. Scope and Foundations ....................................................................... 15
   3.1 Scope ......................................................................................... 15
   3.2 Services, Resources and SLAs ....................................................... 16
   3.3 Management of Services and SLAs ................................................. 18
      3.3.1 Definition of "SLA Management" ................................................ 18
      3.3.2 Service Lifecycle ...................................................................... 19
      3.3.3 Management domains................................................................... 20
   3.4 Building blocks ............................................................................ 20
      3.4.1 Main components......................................................................... 21
      3.4.2 Component setups ....................................................................... 23
4. Modelling Foundation .......................................................................... 25
   4.1 Overview .................................................................................... 25
   4.2 SLA & SLA Template Model............................................................ 26
   4.3 Service Construction Metamodel .................................................... 29
      4.3.1 Service Hierarchy ..................................................................... 29
      4.3.2 Software Landscape ..................................................................... 30
      4.3.3 ServiceType ............................................................................... 31
      4.3.4 ServiceImplementation ................................................................. 32
      4.3.5 ServiceBuilder ............................................................................. 34
      4.3.6 Service Instance .......................................................................... 35
      4.3.7 Example ..................................................................................... 35
      4.3.8 Monitoring System Features and Configuration ................................ 37
5. Architecture Specification .................................................................... 41
   5.1 Goals and Overview ..................................................................... 41
   5.2 Main components ......................................................................... 43
   5.3 Component Interaction ................................................................. 45
      5.3.1 Negotiation interaction ................................................................. 45
      5.3.2 Provisioning interaction ................................................................. 48
      5.3.3 Adjustment & Reporting interactions .............................................. 51
      5.3.4 Renegotiation interactions ............................................................. 52
   5.4 Manageability approach ................................................................ 53
      5.4.1 Manageability .............................................................................. 54
      5.4.2 Monitoring ................................................................................... 55
      5.4.3 Adjustment ................................................................................. 62
6. Adoption Examples ............................................................................. 64
   6.1 ERP Hosting ................................................................................ 64
   6.2 Enterprise IT ............................................................................... 66
   6.3 E-Government ............................................................................. 67
7. Conclusions ....................................................................................... 70
   7.1 Summary .................................................................................... 70
   7.2 Outlook on Future Work ............................................................... 70
8. References ........................................................................................ 71
   Appendix A: Glossary ........................................................................ 73
   Appendix B: Abbreviations ............................................................... 76
### Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Service concepts and their relations to each other.</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>Service Lifecycle.</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>Generic building blocks and their relations.</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>A possible domain split.</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>Building blocks in the envisioned interaction.</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Relations Between SLA(T) Modelling Levels</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>Service hierarchy and the relations of the different layers.</td>
<td>29</td>
</tr>
<tr>
<td>8</td>
<td>Structure of the SCM's elements inside a landscape.</td>
<td>31</td>
</tr>
<tr>
<td>9</td>
<td>Attributes of a ServiceType.</td>
<td>31</td>
</tr>
<tr>
<td>10</td>
<td>Internals of the ServiceImplementation.</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>Detailed view of ImplementationArtefacts.</td>
<td>33</td>
</tr>
<tr>
<td>12</td>
<td>ServiceBuilder and its associated elements.</td>
<td>34</td>
</tr>
<tr>
<td>13</td>
<td>Attributes of a ServiceInstance.</td>
<td>35</td>
</tr>
<tr>
<td>14</td>
<td>Illustration of the different concepts.</td>
<td>35</td>
</tr>
<tr>
<td>15</td>
<td>ComponentMonitoringFeature</td>
<td>37</td>
</tr>
<tr>
<td>16</td>
<td>MonitoringSystemConfiguration</td>
<td>39</td>
</tr>
<tr>
<td>17</td>
<td>Top-level view of the SLA@SOI framework.</td>
<td>42</td>
</tr>
<tr>
<td>18</td>
<td>Component Specialisation Hierarchies.</td>
<td>43</td>
</tr>
<tr>
<td>19</td>
<td>Negotiation interactions on the business layer.</td>
<td>46</td>
</tr>
<tr>
<td>20</td>
<td>Negotiation interactions between software and infrastructure layer.</td>
<td>47</td>
</tr>
<tr>
<td>21</td>
<td>Interactions during the creation of an agreement.</td>
<td>48</td>
</tr>
<tr>
<td>22</td>
<td>Explicit provisioning triggered by the customer.</td>
<td>49</td>
</tr>
<tr>
<td>23</td>
<td>Implicit provisioning of infrastructure services triggered by a timer inside the Infrastructure Manager.</td>
<td>49</td>
</tr>
<tr>
<td>24</td>
<td>Implicit provisioning of software services triggered by the Software Service Manager.</td>
<td>50</td>
</tr>
<tr>
<td>25</td>
<td>Interactions for reporting the status of SLAs and their violation.</td>
<td>51</td>
</tr>
<tr>
<td>26</td>
<td>Interactions for the renegotiation of SLAs.</td>
<td>52</td>
</tr>
<tr>
<td>27</td>
<td>Manageability, Monitoring and Adjustment system architecture</td>
<td>53</td>
</tr>
<tr>
<td>28</td>
<td>Layers of Monitoring System</td>
<td>56</td>
</tr>
<tr>
<td>29</td>
<td>LLMS inner structure, interfaces and interactions</td>
<td>57</td>
</tr>
<tr>
<td>30</td>
<td>LLMS Interaction sequence diagram</td>
<td>59</td>
</tr>
<tr>
<td>31</td>
<td>Sensor interface</td>
<td>60</td>
</tr>
<tr>
<td>32</td>
<td>Reasoning component gateway interface</td>
<td>61</td>
</tr>
<tr>
<td>33</td>
<td>Reasoning component gateway methods</td>
<td>61</td>
</tr>
<tr>
<td>34</td>
<td>Adopted framework architecture for ERP Hosting use case.</td>
<td>65</td>
</tr>
</tbody>
</table>
Figure 35: Adopted framework architecture for Enterprise IT use case. .......... 67
Figure 36: Adopted framework architecture for eGovernment use case. ........... 69
Figure 37: <<Prepare_Software_Services>> sequence diagram .................... 78
Figure 38: Specification of IPrepareSoftwareServices. .................................. 79
Figure 39: Managed_Software_Services interaction sequence ......................... 82
Figure 40: IManageSoftwareServices Specification ....................................... 83
Figure 41: High-level Interaction of POC (via SLAM) and ServiceEvaluator........ 85
Figure 42: Evaluation Result Class Diagram.................................................. 86
Figure 43: IServiceEvaluation interface ...................................................... 87
1 Introduction

This document describes the reference architecture for the SLA management framework created within the second year of the European research project SLA@SOI. The architecture definition includes foundational concepts on terms and lifecycles, a modelling foundation and the actual architecture in terms of components and interactions.

This document serves as formal deliverable D.A1a for reporting the work progress of work package A1 at project month 26. It is complemented by the deliverable D.A1b which provides the actual implementation of the SLA framework. Further insight into specific modules of the overall framework is given within the context of the other scientific work packages A2-A6 via deliverables D.A2a-D.A6a. The technical/scientific evaluation of the SLA framework is subject to deliverable D.B1a, and to the deliverables of the various use cases D.B2a-D.B7a.

The framework’s architecture mainly focuses on separation of concerns, related to SLAs and services on the one hand, and to the specific domain (e.g., business, software, and infrastructure) on the other. Service Managers are responsible for all management activities directly related to services. This includes the management of information about available services, supported types of services, as well as their offered functionality and their dependencies. SLA Managers are responsible for all actions that are related to the service-level agreements. They are involved in the negotiation with customers and they are responsible for the planning and optimization of new services that are to be provisioned. Furthermore, they monitor the terms a provider and customer have agreed upon and react in case of violations. SLA Managers can negotiate with each other in order to find the best offer for a customer. The provisioning of a service is a joined effort of all SLA Managers and Service Managers involved. In order to support multiple domains with our framework, multiple SLA managers and multiple Service Managers can collaborate inside the framework as well as across framework boundaries. Thereby, each SLA Manager and Service Manager is responsible for SLAs and services of a particular domain.

The remainder of this document is structured as follows. Chapter 2 provides an overview of the contributions made by this work package in terms of innovations, actual framework development, and also a task-level overview. Chapter 3 introduces the actual scope and foundational concepts, and provides an overview of the main architectural concepts. Chapter 4 describes the modelling foundation i.e. the most important meta-models that are shared between different components of the architecture. Chapter 5 describes the reference architecture with its main components and interactions. Chapter 6 provides a few examples how the architecture is applied in industrial use cases, and Chapter 7 concludes with a short summary and outlook on future work.
2 Contribution Overview

This chapter provides an overview of the contributions that have been made by the work package A1 in terms of key innovations and contributions to the overall framework. Furthermore it summarizes the actual activities and progress at task-level for the reporting period.

2.1 Key Innovations

There is one single key innovation provided by work package A1: the comprehensive reference architecture of a consistent SLA-management framework.

This innovation directly responds to the mission of SLA@SOI: “to deliver and showcase an innovative open SLA Management Framework that provides holistic support for service level objectives - enabling an open, dynamic, SLA-aware market for European service providers. SLAs will be managed autonomously throughout the complete service lifecycle, spanning the entire services stack from the business layer through to infrastructure. Arbitrary domains will be supported, as demonstrated by evaluations in wide-ranging, grounded, use cases”. Furthermore it integrates the contributions on the four top-level technical objectives of (1) Consistent SLA-management framework, (2) Adaptive SLA-aware infrastructure, (3) Engineering predictable service-oriented systems, and (4) Comprehensive business management suite for e-contracting into one consistent architecture.

The main novelty of this architecture consists of the following aspects:

1. Support for multi-layered SLA management where SLAs can be composed and decomposed along functional & organizational domains.
2. Support for arbitrary service types (business, software, and infrastructure) and SLA terms.
3. Support for the complete SLA and service lifecycle with consistent interlinking of design-time, planning and run-time management aspects.
4. Support for flexible deployment setups, where actual components can be flexibly selected, extended and connected and where founding data models (such as service construction model and the SLA model) can be extended.

More details on this innovation and it’s linkage to the state of the art can be found in [12], [13] respectively.

The reference architecture has been successfully published in a journal [6]; a book chapter is currently in preparation [22].

2.2 Framework Contributions

The actual contribution of this work package to the SLA@SOI framework is primarily of conceptual nature, namely the specification of foundational concepts and their relations (described in Chapter 3), the specification and harmonization of basic metamodels (described in Chapter 4) and the actual architecture specification in terms of building blocks and their interactions (described in Chapter 5).
The analysis of the various data models and views provided by other technical work packages lead to the identification of a conceptual gap as discovered by the architecture task force [23]. This gap hindered the proper interlinkage of different framework components, namely SLA managers, service managers and service evaluation components. To close this gap, work package A1 also specified and implemented the “service construction model” which is described in detail in Section 4.3.

Last, this work package also included an analysis of externally available components that could be reused for our framework. The reuse of lower-level technology components (for logging, modelling, etc.) is further described in deliverable D.A1b [3]. The reuse of higher-level components related to SLA management was analysed in depth, in particular regarding the results provided on the platforms IT-Tude.com and gridsforbusiness.eu. Conceptually, both platforms provided very valid input to our framework architecture. However, the actual reuse of technical components was not reasonable, because of different scope (focus on grid services), conflicting licences (e.g. usage of GPL), and different technology decisions (e.g. usage of MS Visual Studio). Other components (such as Licence Management, Resource Selection Optimization, SLA Accounting for GT4, or Application Virtualization) could be well used as a complement to our framework for specific setups. Further details on this assessment are provided in [13].

Last, it is important to stress that the development of our framework components is done in a way that also supports the selected reuse of single components independent of the overall framework. Thus users can either adopt the complete framework or take it as a toolbox and select those elements relevant for them.

2.3 Task-level activities

The main activities and their progress at task level are summarized below.

Task A1.1 Terminology Alignment performed a thorough revision of the project wide glossary - based on experiences gained within the consortium but also reflecting feedback from external stakeholders. Several terms have been refined, many new terms have been added (e.g. on the service lifecycle and relevant roles). The most important changes are summarized in Chapter 3. The complete revised glossary is provided in [1].

Task A1.2 Service Integration performed the functional harmonization of the various technical components developed for the framework. As part of this activity also a few missing pieces and required adjustments where identified such as the need for a Software Service Manager component (assigned to work package A3), the generalization of the service prediction component to a Service Evaluation component (assigned to work package A6). The result of these activities formed the basis for the overall architecture definition and can be seen in Section 3.4 and in more detail in Chapter 5.

Task A1.3 Model Integration performed the alignment of the various metamodels used throughout the project, namely the SLA model, the service construction model, the Open Cloud Computing Interface, and the Palladio Component Model (PCM). In particular, the previous core model developed in year 1 has been merged with the SLA model and the service construction model has been newly introduced. The result of these activities is described in Chapter 4.

Task A1.4 Architecture Definition performed the overall definition of the framework architecture. This required substantial work in order to extend, to flex, and to enhance the concepts developed in the adhoc architecture within the first project period and eventually lead to a significant re-design at the top-level
though many components below that level could be reused from the adhoc architecture. The result of this activity is described in detail in Chapter 5.

Task A1.5 Technology Integration performed the actual integration work for developing the full SLA management framework in the reporting period. The result of these activities is described in detail in [3].
3 Scope and Foundations

This chapter provides the foundation for the architecture discussion as it sets the basic scope and introduces the main concepts/terms and their relationship. Note that more detailed definitions around the notion of services, SLAs, roles and IT systems can be found in the SLA@SOI glossary [1]. A summary, of those terms is given in Appendix A.

3.1 Scope

Functional Goals

The primary functional goal of our SLA management framework is to provide a generic solution for SLA management that

1. supports SLA management across multiple layers with SLA composition and decomposition across functional and organizational domains;
2. supports arbitrary service types (business, software, infrastructure) and SLA terms;
3. covers the complete SLA and service lifecycle with consistent interlinking of design-time, planning and runtime management aspects; and
4. can be applied to a large variety of industrial domains and use cases.

In order to achieve these goals, the reference architecture is based on three main design principles. First, we put a strong emphasis on a clear separation of concerns, by clearly separating service management from SLA management and by supporting a well layered and hierarchical management structure.

Second, a solid foundation in common metamodels for SLAs as well as their relation to services and the construction of actual service instances is an important aspect to support clear semantics across different components of the framework.

Third, design for extensibility/adaptability is a key aspect in order to address multiple domains. Therefore, we clearly distinguish between generic solution elements and places where domain specific logic/models need to be provided. Furthermore, we seek for an architecture where even generic parts can be replaced by domain specific versions, which might be dictated by already existing (legacy) management functionality.

Technical goals

A set of technical requirements and goals has been collected from various industrial use cases and external stakeholders. They fall into the four categories of Framework Configuration & Setup, Framework Model Configuration, Framework Operation, and Framework Access.

Model-related requirements are mainly about model extensibility and are addressed by the design of the SLA model and the service construction model.

Other requirements relate to the usage of certain technology standards and are taken into account by the actual framework development [3].

Full details about these requirements and their evaluation can be found in deliverable D.B1a [14].
Use-Case driven Approach

Both, functional and technical aspects have been developed based on requirements collected from other stakeholders. The main stakeholders considered are: the four industrial use cases within the project, project partners contributing direct insight from their organization, and project-external organizations interested in the topic of SLA management. Further details on this process are described in [14].

Positioning against adhoc architecture

Comparing these goals with the capabilities of the adhoc SLA management framework architecture as developed in the first project year [2], we see two main discrepancies:

First, the adhoc architecture focused on the specific use case around the open reference case. The initial version of this use case\(^1\) has a specific and predefined setup in terms of organizational relationships (1 single service provider), the focus on software-based services and also the types of underlying required infrastructure resources.

This focus prevents a direct application of that adhoc framework to other use cases with other organizational setups (multi-party), other types of services (e.g. business-level services, networking services) or other types of resources (complex software systems, human resources, ...).

Second, the adhoc architecture was realized as a fully hard-wired framework but does not allow for reuse of individual framework components. This prevents a flexible application of the framework to use cases that can only make use of partial functionality of the overall framework.

In summary, there is a clear need to evolve the adhoc architecture in order to support its application for a large variety of different use cases.

3.2 Services, Resources and SLAs

A first and fundamental concept for the architecture is the clear distinction between resources, services and SLAs.

Following the ITIL definitions [7] and in accordance with the SLA@SOI glossary we can define them as follows:

- **Service**: A means of delivering value to Customers by facilitating Outcomes Customers want to achieve without the ownership of specific Costs and Risks.

- **Resource**: A generic term that includes IT Infrastructure, people, money or anything else that might help to deliver an IT Service. Resources are considered to be Assets of an Organisation.

- **SLA**: An Agreement between a service provider and a customer. The SLA describes the service, documents service level targets, and specifies the responsibilities of the service provider and the customer. A single SLA may cover multiple services or multiple customers.

\(^1\) Within the next project phase we will extend the scope of the open reference case so that it covers a broader set of requirements and features.
To further stress the distinction between these concepts we can state that services are about the activity to bring value to customers. They are not about the artefacts needed at the provider side to deliver that value. However, it’s probably safe to assume that services always require some resource(s) as means for their delivery. The following 2 examples show this distinction:

- A hotel booking service listed in a public registry requires hotel rooms and some software as resources for the actual service delivery (and possibly also human resources). A service endpoint may refer to a hotel room but neither the hotel nor a hotel room is a service.
- A compute service provided by Amazon requires infrastructure resources but also some management software resources. A service endpoint may refer to an infrastructure entity (such as a computer) but the entity is not a service.

In order to close the gap between abstract services and concrete resources that are eventually needed for a deliverable, we further refine the concept of services and their concreteness via 3 specializations:

- **Service Type**: Specifies the service as a fully abstract entity via its external interface.
- **Service Implementation**: describes specific resources or artefacts (such as software components, or appliances) which allow for instantiating the service. Service implementations may still depend on other services. There can be different implementations of a given service type.
- **Service Instance**: is a running and accessible service which is ready for consumption by service users. It has one or more service endpoints (for service consumption) and a management endpoint (for service monitoring & control). Service instances might have multiple service/management endpoints if their service type specifies a bundled service.

As an example for these concepts we can take a database service. The abstract type of such a service is mainly specified by the fact that the service is exposed via an SQL interface.

Different service implementations may exist for such a database service, for example a MySQL database or an IBM DB2 database. These implementations may rely on other services such as a storage service.

For each implementation, multiple services instances might be created. These may differ from each other either in their concrete configuration. For example, one instance might be configured for optimized read access, another one for fast write access.

The following diagram shows now the main concepts coined for services and SLAs as well as their relationships.
A few further explanations:

SLA Templates specify the types of SLA offers a service provider advertises it is willing to accept.

An SLA represents a potential agreement between a service provider and a customer that describes the service, documents service level targets, and specifies the responsibilities of the service provider and the customer. An agreed SLA also refers to the endpoints of exactly one service instance. For example, an SLA for an instance of the address validation service contains the Web service endpoints for invoking the validation functionality.

Service dependencies relate to service types a given implementation relies upon. These dependencies must be resolved to concrete instances of the depending sub service in order to instantiate the higher level services.

### 3.3 Management of Services and SLAs

Following the core concepts, we now briefly sketch our notion of management and the related lifecycles of SLAs and services.

#### 3.3.1 Definition of "SLA Management"

The term management is interpreted here as "control" - in the classic control-theory sense; synonymous with "applied constraint". In particular, the relation between ’manager’ & ’managed’ is defined as a control relation – formally:

Given a "managed" system, $S$, with $n$ degrees of freedom, the "manager" $M$ "manages" $S$ if $M$ acts upon $S$ to reduce the degrees of freedom of $S$ to $m < n$.

We also assume that the management actions enacted by $M$ upon $S$ are goal-based – i.e. they serve to satisfy one or more management objectives, which are in some way dependent on the state of $S$. To this end, the management relation is necessarily bi-directional; the application of management entails a continuous feedback loop in which $M$ observes the dynamic state of $S$, and acts upon $S$ in order to constrain its state dynamics in some way. Both observation (sensing) and action are necessarily mediated by information exchange (appropriate interfaces are defined in Appendix B). Finally, management systems can be:
- **hierarchically organised**: each level operates under the constraints imposed by higher levels, and serves in turn to constrain lower levels.
- **distributed**: to the extent permitted by the communication channels supporting the management relation.

We interpret “SLA Management“ as the management of service delivery systems in order to meet the QoS objectives (goals) specified in SLAs. SLA management covers all stages in the SLA lifecycle:

- SLA Template design : ensuring that offered QoS guarantees are realistic,
- SLA negotiation : ensuring that agreed QoS guarantees are realisable,
- SLA runtime (effective period of SLAs) : ensuring that QoS guarantees are satisfied,
- SLA(Template) archiving : ensuring that previous experience is available to future cycles

### 3.3.2 Service Lifecycle

The management of SLAs happens in the context of the overall service lifecycle, which is depicted in the following figure.

**Figure 2: Service Lifecycle.**

The lifecycle consists of the following stages:

- Design and Development: development of artefacts needed for service implementation
- Service Offering (incl. SLA template design): offering a service (type) to customers; results include specification of SLA templates
- Service Negotiation (incl. parts of SLA negotiation): actual negotiation between customer and provider; results in an agreed SLA
• Service Provisioning (incl. parts of SLA negotiation): all activities required in system preparation and setup for allowing service operation, including booking, deployment (if needed), and configuration. Note that provisioning does not necessarily imply deployment as for example in a multi-tenant environment the provisioning of a new tenant might be a simple reconfiguration of the running system.

• Service Operations (incl. SLA runtime): an actual service instance is up and running; it might be adjusted in order to enforce an SLA

• Service Decommissioning: the service instance is stopped and can no more be accessed by the service customer

A more detailed view on this service lifecycle is provided in [2].

### 3.3.3 Management domains

Another important aspect of management is the notion of management domains. So far we distinguish two main kinds of domains, the first driven by business considerations, the second driven by technical considerations.

1. **Administrative domains** are areas of organizational coherence, e.g. an independent organization or a department that operates largely as a profit centre. Within an administrative domain two main views can be considered:
   a. The business view representing basically a sales department, i.e. the activity to sell services via SLAs.
   b. The management view that oversees all the offered or active SLAs within a certain domain and which is responsible for the eventual SLA operation.

2. **Technical domains** are areas where a certain kind of resources or artefacts can be coherently managed, e.g. domains for infrastructure artefacts, software artefacts, business artefacts or even subdivisions of these.

Technical domains could be understood as horizontal layers within a business/IT stack, while administrative domains relate more to vertical, cross-cutting pillars within an organization (though they can form a hierarchy as well). Section 3.4.2 gives insight on how the notion of domains impacts the architecture.

### 3.4 Building blocks

In this chapter we introduce the main building blocks that constitute our framework, explain their responsibilities, show how they can be specialized for specific domains, and explain how they can be combined in order to serve different scenarios and setups.
3.4.1 Main components

Figure 3: Generic building blocks and their relations.

Figure 3 gives an overview of the framework’s main components and their relations. The “leading” component is the Business Manager which is responsible for business related information and business-driven decisions. It controls the SLA Manager which is responsible for SLA templates and actual SLAs. It uses the Service Manager for querying service implementations and orchestrating provisioning activities. The Service Manager is responsible for managing actual service implementation. It uses Manageability Agents for triggering run-time management activities. Last, the SLA Manager also relies on the Service Evaluation for retrieving predictions of service qualities. More detailed discussions of these components and their relations follow below.

Taking the business-rooted ambition of an SLA management framework, the root of the management hierarchy is the Business Manager component. It is responsible for asserting overall business constraints on the system in order to meet business objectives and for maintaining customer and provider relations. To that extent, it captures knowledge about pricing schemes (incl. rewards, promotions and discounts), customer profile information, 3rd party service provider profiles and business rules for taking cost/profit-aware decisions. Business managers may contain sensible data that must not be shared among components. The actual functionality of a Business Manager includes:

- searching and publishing of products
- customers & service providers management
- negotiation and establishment of agreements/contracts with customers and service providers
- notification of bills & penalties to customers and service providers

Following the concepts introduced in Section 3.1, we consider the following concepts as core to our architecture: SLA, service, and resource. Consequently the architecture contains dedicated management components for all three of them.
The **SLA Manager** component is responsible for managing a set of SLA Templates and SLAs in its domain. Furthermore, it captures knowledge about negotiation and planning goals (such as utility functions or policies). Depending on the specific context/requirements of a use case a separate SLA Manager may be set up for a complete organization, per department, or for each individual service. The actual functionality of an SLA Manager includes:

- searching and publishing of SLA templates
- negotiation of SLAs with customers and 3rd parties including conversion between different SLA formats
- SLA planning and optimization
- SLA provisioning and adjustment

The **Service Manager** component is in charge of managing the elements necessary to instantiate a service. In particular it knows about the structure of service implementations and keeps track of existing service instances. Service Managers can be created for any technical domain which needs consistent management. The actual functionality of a Service Manager includes:

- publishing of service implementations
- maintenance of a service landscape, incl. elements required for instantiating a service implementation
- reservation and booking of service instances
- mediation of management/adjustment operations to service instances and Manageability Agents
- triggering of actual service provisioning

The **Manageability Agent** component acts as gateway to actual resources. It knows about the available sensors and effectors that can be used for managing a certain service instance and its resources. Manageability Agents may exist per resource, per service instance or per collections of these. The actual functionality of a Manageability Agent includes:

- sensing/monitoring the status of service instances and resources
- searching for and executing manageability actions

Finally, to support pro-active management decisions, at all levels, the framework also provides a **Service Evaluation** component. It relies on background information (from design-time, run-time or historical) about the quality characteristics of services. It provides functionality for a priori quality evaluations of services – depending on influencing factors such as customer behaviour or lower-level service qualities. Service Evaluation components may exist per SLA Manager or even for sets of these.

While all these components have clearly distinct responsibilities, they also need to have some common understanding of the overall problem domain.

- Service types must be commonly understood by SLA managers, service managers and service evaluation.
- The identity of service implementations must be commonly understood by service managers and service evaluation, though both components may rely on completely different data models in order to deal with service implementations.
- SLA terms and their quality aspects must be commonly understood by SLA managers and service evaluation.
• SLA terms and available monitoring handles must be commonly understood by SLA managers and service managers.
• The SLA negotiation process with customers and thirds parties must be commonly understood by business managers and SLA managers.

3.4.2 Component setups

As stated in the design goals for the framework architecture, a key goal is the support for flexible configurations and setups where different domain cuts can be realized.

Our architecture supports cuts along the two main criteria mentioned in Section 3.3.3, i.e. via administrative domains and technical domains. Administrative domains are characterized by having a dedicated SLA Manager, which is in charge of all the SLAs within that domain. Technical domains are characterized by having a dedicated Service Manager, which is in charge of all the artefacts needed for a (set of) service implementation(s).

The following figure shows an example of how such a domain cut can be realized for a single service provider organization that interacts with customers and 3rd parties. Basically, the service provider organization is split into 2 main administrative domains: one might be for application services, the other for infrastructure services. Furthermore, there is a split into 3 technical domains, each represented by a service manager. For example, one might be for application artefacts, one for middleware artefacts and the last one for infrastructure artefacts.

![Figure 4: A possible domain split.](image)

Other examples for domain splits are shown in Chapter 6, where we sketch the adoption architecture of the four main use cases pursued in our project.
Last, we want to show the relation between the introduced building blocks and the original SLA@SOI vision expressed in an interaction diagram (shown for example in [2]).

The following figure shows exactly the original interaction diagram and extends it by displaying where the concepts of Business Manager, SLA Manager, Service Manager and Manageability Agent apply.

![Diagram showing building blocks in the envisioned interaction.](image)

**Figure 5: Building blocks in the envisioned interaction.**

Note, that this figure does not show Service Evaluation components in order to keep it at reasonable complexity. Nevertheless, Service Evaluation components can be associated in here to each SLA Manager.
4 Modelling Foundation

4.1 Overview

Models play a central role in the SLA@SOI framework. Different parties use them to negotiate and plan services and their respective SLAs. They hold information needed for a priori quality evaluation, for runtime monitoring of SLAs, and for provisioning services. In this section, we introduce the Service Construction Metamodel as an integral part for the planning and evaluation of services. Furthermore, we give an overview of the SLA-Model, a domain-independent model for the specification of SLAs and SLA-Templates. A detailed description of the SLA-Model can be found in deliverable D.A5a [19]. Both metamodels are the central elements for internal and external communication of the SLA@SOI framework.

In addition, the SLA@SOI framework employs metamodels for managing infrastructure services in a cloud and for a priori quality evaluation of software services: OCCI (see deliverable D.A4a [18]) and the Palladio Component Model (PCM) (see deliverable D.A6a [20]).

The Open Cloud Computing Interface (OCCI) is an open protocol for infrastructure service in a cloud computing environment. OCCI is a REST (Representational State Transfer) based protocol and hence it adopts a resource oriented architecture. Each resource i.e. a compute node is identified by URL(s) and has one or more representations. At the heart of OCCI protocol is a metamodel which defines the objects and their relationships. The metamodel is designed with extensibility in mind to manage other types of cloud resources in the future like storage, networks, application etc. For now, the metamodel is used to described compute resources and their inter-relations. An interface exposes “kinds” (category of resources distinguished by some common characteristic or quality) which have “attributes” and on which actions can be performed. The attributes are exposed as key-value pairs and applicable actions as links.

The PCM is a metamodel and toolset for software quality analysis. Its focus lies on software performance evaluation and reliability prediction. As such, it follows the paradigm of model-driven performance (and reliability) engineering. Software architects specify their system in a language specific to their domain, i.e. component diagrams, deployment diagrams, and activity diagrams. These diagrams are annotated with information necessary for performance and reliability prediction. For quality prediction, the architectural model can be transformed into a discrete-event simulation or analytical models, such as Layered Queueing Networks (LQNs), stochastic process algebras, or in case of reliability prediction discrete time Markov chains.

In the remainder of this chapter we give an overview of the SLA Model (Section 4.2) as well as a detailed description of the Service Construction Metamodel (Section 4.2).
4.2 SLA & SLA Template Model

In this section, we provide a brief summary of the SLA & SLA Template Model, henceforth SLA(T) Model. A more complete account of the SLA(T) modelling objectives & approach can be found in the A5 deliverable (D.A5a [19]), together with a detailed description of the SLA(T) specific model layer, and a complete list of the “Standard Terms”. Business-related aspects are dealt with in the A2 deliverable. In this document, we provide only a brief overview of the SLA(T) Model. In Y1, various elements comprising the SLA Model were developed in parallel by different work-packages, in particular; the Core Model (A1), Business SLA Model (A2) and the Conceptual SLA Model & Common Terms (A5). In Y2, these elements have been elaborated and combined into a single coherent model, which is constructed in a hierarchical fashion as schematised in Figure 6.

Each level in the hierarchy builds on the specifications at lower levels, which define:

- **data primitives**, 
- **ground expressions**: encapsulating constraints, events & functions, and
- **service descriptions**: encapsulating the functional properties of services (e.g. interfaces & operations).

These lower level modelling elements are not specific to SLA(T)s, but rather serve as a common information model for the SLA@SOI framework as a whole. Their specification extends and generalises the Core Model work carried out in Y1. The SLA(T) Model properly builds on these lower levels to encapsulate non-functional (QoS) descriptions of services, and to formalise the notion of a QoS Guarantee. The highest ‘Business Product’ layer is reserved for the various business-related aspects of SLAs, e.g. aspects relating to service costs, penalties, customer information, exclusion clauses, termination conditions, and the like.

The model as a whole is conceived as an abstract SLA(T) syntax, meaning that:

- the model specifies the formal (syntactic) constraints on the information content of SLA(T)s, but
- these constraints are defined only at a coarse level, without committing to any specific syntactic tokens.
Concrete instantiations of the abstract syntax are generated by specifying a
language (e.g. XML, Java, BNF Grammar, etc) and a vocabulary of domain-
specific terms (i.e. specific tokens for the nominal entities in the model). The
particular vocabulary developed & employed in SLA@SOI is referred to as the
“Standard Terms” vocabulary.

As shown in Figure 6, the lowest-level of the SLA(T) Model hierarchy comprises a
taxonomy of data primitives that classify different kinds of data assumed by the
model. Ground expressions build on the primitives specifying a set of common
generic expression that support the representation of annotations, functions,
events and constraints. Annotations provide an open-ended means for domain-
specific applications to customize content according to their own needs. Functional expressions have a functional form and semantics, denoting a mapping
relation from some input to output domain (i.e. in principle, functional expression
can be evaluated). Event expressions identify specific classes of events. Finally,
constraint expressions serve to define limits, or bounds on values. For example,
“X <= 10”, is a constraint which places an upper bound of 10 on the value of X.

The SLA(T) model provides means to describe the services whose quality
attributes are to be defined within an SLA. The primary service description model
employed for this purpose is a generalisation of WSDL 2.0; abstracting from the
specific notion of web-service to the more generic concept of invocable, or
message-based service. Accordingly, the functional aspects of services are
captured as an interface, whose formal definition is referred to as an interface
specification, which essentially comprises a set of interface operations. Each
operation, in turn, is specified as a signature, comprising the name of operation,
together with zero or more typed input, output & related parameters, and zero
or more so called standard terms denoting any faults (or exceptions) that may
occur during execution of the operation.

The SLA(T) Model and the Standard Terms vocabulary build on the primitives,
ground expressions and service descriptions (cf. Figure 6). Here we present just a
summary account of the SLA(T) Model & Standard Terms vocabulary, a more
detailed description is provided in Section 4.1.2. of deliverable D.A5a [19].

The SLA(T) Model provides an abstract syntax for expressing the content of
SLA(T)s. For present purposes, the difference between an SLA and an SLA
Template is negligible; both essentially comprise:

- a means to identify the various parties to the agreement, in particular the
  service provider & customer,
- a means to identify the functional properties of offered services – i.e.
  service descriptions,
- a means to specify the terms of the agreement, that is; the various
  obligations that parties commit to. These obligations are encoded in the
  form of guarantees, which come in two basic flavours:
    - guaranteed states; expressed as constraints over non-functional
      (QoS) service properties, and
    - guaranteed actions; concerning, for example, requirements for
      reporting SLA status, or for the payment of penalties in case of SLA
      violation.

---

2 The abstraction to a generic concept of service is a requirement of B-Line use-cases (in particular B6 e-Government).
3 A ‘related’ parameter is a parameter whose value can be determined during execution of the
operation, but which is not explicitly part of the input or output of the operation. Examples of related
parameters include; the time at which the operation is invoked, the identity of the invoker, etc.
The Standard Terms vocabulary, instead, defines the default SLA@SOI tokens to be used by concrete instantiations of the abstract SLA(T) syntax. These terms include, among others:

- the set of **logical**- & **comparison**-operators to be used in constraint expressions,
- various common arithmetic & set/aggregate functions, e.g. covering *addition, multiplication, summation, mean/mode/median*, etc.
- **time-series functions**; for specifying sampling schedules, and
- common **QoS metrics**, such as; service *completion-time, availability, supported-standards, authorisation*, etc.

Following standard web-ontology practices, all the Standard Terms are defined as URIs, such that different namespaces may be employed to signify local variations. Domain-specific applications can define their own additional terms as required.

Finally, at the time of writing, the abstract SLA(T) syntax is available in three concrete forms; as a Java API, an XML-Schema, and in ‘human friendly’ notation as a semi-formal BNF Grammar.
4.3 Service Construction Metamodel

The Service Construction Metamodel (SCM) is motivated by the need to store and manage information about services inside the SLA@SOI framework. ServiceManagers have to provide data about the types of services offered, about alternative realizations of these service types, and about the service instances that have already been provisioned. Furthermore, SLAManagers require information about the dependencies of a service on other services, about features of the service itself, and about features of its associated monitoring system. Based on this information, SLAManagers can plan and negotiate SLAs with their customers and acquire other (external) services that are required.

The Service Construction Metamodel is driven by the information necessary to create, evaluate, and maintain services and their associated SLAs. As such, it is an essential part for the communication of SLAManagers, ServiceEvaluation, and (Software-)ServiceManagers. With the SCM, (Software-)ServiceManagers can maintain information about provided and required service types, available service implementations, and running service instances. SLAManagers and ServiceEvaluation can retrieve information about service dependencies, about features of the service, and about its monitoring system. Furthermore, they can resolve dependencies to external services and provide particular configurations of the service and its monitoring system in a generic way. ServiceManagers instantiate services that have been negotiated by the SLAManager based on their configuration.

In the following, we will introduce the basic elements of the SCM and describe their purpose. In Section 4.3.1, we introduce the hierarchical structure of ServiceTypes, ServiceImplementations, and ServiceInstances. Section 4.3.2 illustrates how the different elements of the SCM are stored and maintained inside the service landscape. Sections 4.3.3 to 4.3.6 introduce all elements of the SCM in more detail. Finally, Section 4.3.7 illustrates the concepts and their usage by means of an example.

4.3.1 Service Hierarchy

During its life-cycle, a service exists in different aggregate states that have to be reflected inside the SLA@SOI framework. The service hierarchy shown in Figure 7 reflects the different states a service can assume during its life cycle: ServiceType, ServiceImplementation, and ServiceInstance.

![Service hierarchy and the relations of the different layers.](image_url)
A **ServiceType** describes the functionality that a service provides. For example, it contains pointers to the WSDL definitions of the service's interfaces. The same ServiceType can be realized by multiple ServiceImplementations.

A **ServiceImplementation** consists of a set of **ImplementationArtefacts** (such as software components, or appliances for virtual machines) that are required to instantiate the service. Each ImplementationArtefact has a set of **Dependencies** to other services. For example, a software service that is realized by an appliance depends on an infrastructure service that is able to host that particular appliance. Similar to the relation of implementation and instance in object-oriented languages, an arbitrary number of ServiceInstance can be created for each ServiceImplementation.

A **ServiceInstance** describes the properties of a service that is (about to be) provisioned and accessible. For example, a ServiceInstance contains the endpoint of a particular service. The endpoint either refers to a running service instance or points to the location where the service will be available according to the time constraints defined in the corresponding SLA.

In order to instantiate a service for a customer, various degrees of freedom have to be resolved. For example, all dependencies of an implementation on other services need to be bound to offers of an external service provider or of another SLA Manager. The **ServiceBuilder** provides a generic way to resolve service dependencies and provide custom configurations for a service and its associated monitoring system. For each dependency, the ServiceBuilder holds a **ServiceBinding** that maps the Dependency to an SLATemplate or one of its specializations (SLA and BusinessProduct). The SLATemplate contains all information necessary to assess and access a service outside of the current SLAManager's domain. It includes quality constraints and, after the SLA has been agreed, endpoints of the service.

In the following section, we describe how the elements introduced above are maintained inside a software landscape. Even though this can be considered an implementation detail of ServiceManagers, it supports the understanding of the overall model.

### 4.3.2 Software Landscape

Inside the ServiceManager, the **Landscape** is the central element holding and managing all elements introduced in the previous section (ServiceTypes, ServiceImplementation, ServiceInstance, and ServiceBuilder). The Landscape contains and organizes the various services that are being offered by a service provider. As such, the Landscape is the SCM’s root element and can only exist once in each ServiceManager. Figure 8 shows the Landscape and its relations.
Figure 8: Structure of the SCM’s elements inside a landscape.

A Landscape contains a set of provided and required ServiceTypes, a set of implementations of these types, and all instances that are (about to be) provisioned. We explicitly distinguish between ServiceTypes that are offered to customers and ServiceTypes that are required to fulfill their functionality. Furthermore, the Landscape holds all ServiceBuilders that have been used to provision a ServiceInstance. These ServiceBuilders allow to retrieve information about the service’s configuration at runtime. In addition to the elements of the SCM, the Landscape contains a ServiceTopology which specifies how different software elements are connected to each other. Details about the software landscape model can be found in the deliverable D.A3a [17]. In the following, we describe the SCM’s core elements in more detail.

4.3.3 ServiceType

Figure 9: Attributes of a ServiceType.

A ServiceType specifies the externally accessible interfaces of a service. It basically describes what functionality a service provides. For this purpose, a ServiceType contains a set of interfaces that specify its exact functionality by a set of operations it makes available. A detailed specification of interfaces can be found in the description of the SLA Model (cf. [19]). ServiceTypes establish a link to particular interfaces by means of a unique identifier encoded in a string. It can either contain a full interface reference as specified in the SLA Model (cf. [19]) or be a unique name for an interface without a formal definition. An example for the latter would be the user interface of an ERP system.

The remaining attributes serve the purpose of identification by machines (ID) and humans (ServiceTypeName and Description).
4.3.4 ServiceImplementation

ServiceImplementations (cf. Figure 10) realize a specific ServiceType. They describe i) specific artefacts and assets (such as software components, or appliances) that are required to instantiate a service, ii) the dependencies of assets and artefacts on other services, and iii) the configurable features of the particular service implementation and its associated monitoring system. For example, the “All-In-One” implementation of the ORC’s services depends on one infrastructure service that hosts its appliance. The threadpool size of its Active BEPL engine can be adjusted according to the services usage and deployment. Furthermore, the monitoring system allows for instrumenting each service operation and extract response times and throughput.

To express such properties, ServiceImplementations contain a set of ComponentMonitoringFeatures, some ProvisioningInformation, and a set of ImplementationArtefacts. ComponentMonitoringFeatures specify the capabilities of the monitoring system associated to a service. The contain information about the available sensor, effectors and reasoners. Section 4.3.8 provides further details on the monitoring system’s metamodel.

ProvisioningInformation and ImplementationArtefacts hold the information necessary to plan and execute the provisioning of a service. ProvisioningInformation contains the LeadTime that is needed to start a particular software system. Furthermore, it specifies the boot order in which multiple ImplementationArtefacts are to be started. Each ImplementationArtefact represents a single unit such as appliances or software archives that have to be deployed separately. The metamodel of an ImplementationArtefact contains information about its Dependencies and the ConfigurableServiceFeatures associated with this artifact. Figure 11 presents a detailed view of the ImplementationArtefact and its elements.
A **Dependency** refers to ServiceTypes that are necessary to instantiate a given ServiceImplementation. Dependencies are part of ImplementationArtefacts to allow a direct association of a dependency to the artifact that requires it. This is necessary, if multiple artefacts require the same ServiceType with different quality characteristics. In this case, multiple SLAs need to be established for the same service type. Thus, a unique mapping between each SLA and the requiring ImplementationArtefact is necessary. A typical example for such a scenario is multiple dependencies to infrastructure services. If a ServiceImplementation comprises multiple appliances (e.g. one for the application server and one for the database), the model needs to reflect which appliance is to be hosted on which virtual machine.

Furthermore, **ConfigurableServiceFeatures** describe properties of a service that can be adjusted for each instance. Again, ConfigurableServiceFeatures are directly associated to ImplementationArtefacts in order to avoid disambiguates. ConfigurableServiceFeatures comprise a unique identifier (ID), a configuration type (e.g. property file or environment variable), a pointer to a file (ConfigFile), and an identifier of the parameter to be adjusted (ParameterIdentifier). Identifiers depend on the configuration and file type considered. For example, the identifier of a parameter in an XML file can be an XPath expression.

ImplementationArtefacts are abstract entities that have to be specialized for different domains. In Figure 11, we show subclasses for ResourceArtefacts, DeploymentArtefacts, and DataArtefacts. DeploymentArtefacts are further refined to VirtualAppliances and SoftwareArchives. These elements contain detailed information necessary to deploy the artifact. A detailed description of these elements is given in deliverable D.A3a [17].

Please note that ImplementationArtefacts are only used inside a ServiceManager. Thus, the information about the internal structure of a service implementation does not have to be understood by SLAMangers and ServiceEvaluation. For this purpose ServiceImplementations contain explicit operations that aggregate the Dependencies and ConfigurableServiceFeatures for external processing.
4.3.5 ServiceBuilder

ServiceBuilders serve as a communication datastructure to be used by SLAManagers, (Software-)ServiceManagers and ServiceEvaluation. ServiceBuilders are associated with a ServiceImplementation for which they construct a new service instance. They are used throughout the whole negotiation and provisioning process and are stepwise enriched with information. Basically, ServiceBuilders serve as configuration objects for new service instances. SLAManagers, ServiceEvaluation, and ServiceManagers exchange information on a potential service using the ServiceBuilder. The implementation of the ServiceBuilder follows the Builder pattern of Gamma et al. [21]. For a single ServiceImplementation, multiple ServiceBuilders can exist that are associated with it.

A ServiceBuilder (cf. Figure 12) is responsible i) for resolving dependencies of a ServiceImplementation by offers of an external service provider or another SLA Manager, ii) for the configuration of specific service features, and ii) for the configuration of specific monitoring features. For each dependency of a ServiceImplementation, the ServiceBuilder can hold a ServiceBinding that maps the dependency to an SLATemplate or one of its specialisations (SLA and BusinessProduct). The SLATemplate represents a contract for a service of the required type. It contains all information necessary to assess and access a service outside of the current SLAManager’s domain. It includes quality constraints and, after the SLA has been agreed, endpoints of the service.

For configuration purposes of a service, ConfigurationDirectives assign new values to ConfigurableServiceFeatures. The setup of the monitoring system is given in the MonitoringSystemConfiguration. Details about monitoring features and configurations follow in Section 4.3.8.

Figure 12: ServiceBuilder and its associated elements.
### 4.3.6 Service Instance

A **ServiceInstance** (cf. Figure 13) refers to the instantiated version of a ServiceImplementation. As such, it contains information about the runtime aspects of the deployed services. For example, it contains the Endpoints of a particular service. The Endpoints either refer to a running and accessible service or point to the locations where the service will be available according to the time constraints defined in the corresponding SLA. Each ServiceInstance can contain multiple endpoints. Additionally, a ServiceInstance contains information about the date and time of its creation (InstantiatedOn) and the usual means for identification by machines (ID) and humans (ServiceInstanceName, Description).

#### 4.3.7 Example

In the following, we give a simple example of ServiceImplementations, Dependencies, ServiceBuilders, ServiceBindings and their usage in the overall system.

Figure 14 shows a ServiceImplementation called "MyComponent" with an explicit dependency on an external service type called "IExternal". The ServiceImplementation is depicted as a component that provides a service of type "IMyService". The link between the implementation and the ServiceType reflects the Dependency of the metamodel. This concept is analogue to (required) roles in common component models. In Figure 14, the dependency is resolved by a ServiceBinding that links the Dependency to an SLATemplate of an external provider. The SLATemplate contains the specification of interface "IExternal". The external service type can again be realised by a ServiceImplementation.
In the following, we illustrate how the ServiceBuilders can be used for communication between SLAMangers, (Software-)ServiceManagers, and ServiceEvaluation.

1. The SLAManager requests the ServiceImplementations of a particular ServiceType from the ServiceManager

2. For each ServiceImplementation
   1. The SLAManager creates a ServiceBuilder
   2. The SLAManager resolves the dependencies of the ServiceImplementation using available SLATemplates, SLAs, and BusinessProducts
   3. When all dependencies have been resolved, the ServiceBuilder is passed to ServiceEvaluation which assesses the expected quality of the setting given by the ServiceBuilder.

4. Steps 1, 2 and 3 may be repeated several times

3. When a particular ServiceBuilder is to be instantiated, the SLAManager (tries to) agree on the selected SLATemplates (or SLAs) of the depending services and adds the corresponding Endpoints to the SLATemplates. The resulting object is passed to the ServiceManager which instantiates the requested service based on the settings given in the ServiceBuilder.
4.3.8 Monitoring System Features and Configuration

Component monitoring features

The ComponentMonitoringFeatures class of the ServiceImplementation model represents the capabilities of a service component implementation that offers certain monitoring features. ComponentMonitoringFeatures are passed as input to the check_monitorability method of the MonitoringManager component. The MonitoringManager is used during SLA planning in order to cross-check monitoring requirements as they are derived from SLA agreement terms against the capabilities of the monitoring system (further details on this are described in the architecture of the Generic SLA Manager in [19]).

A component provides information about its monitoring features by creating an instance of the ComponentMonitoringFeature class. The figure below shows the ComponentMonitoringFeature class and its sub-classes.

![ComponentMonitoringFeature Diagram]

**Figure 15: ComponentMonitoringFeature**

A ComponentMonitoringFeatures class has two instance variables:

- **type**: holds the type of the component. The allowed types are: SENSOR, EFFECTOR, and REASONER. A sensor provides information about a service, an effector changes the properties of a service, whilst a reasoner processes information to produce a result, e.g., consumes information provided by sensors and reports whether an SLA is violated or not.

- **uuid**: uniquely identifies the component to whom belong the monitoring features. It has the component UUID same value.

- **monitoringFeatures**: is a list of MonitoringFeature objects.

A ComponentMonitoringFeatures class has a list of MonitoringFeature objects. That list can be empty. A monitoring feature can be a function, a primitive, or an event. All monitoring features must the following information:

- **name**: is a unique label that identifies the feature. For instance, in case of reasoners, it can be the name of supported operations, e.g., response_time, throughput, availability. In case of sensors, it can be the list of events it provides, e.g., request_event, response_event.

- **description**: is a human readable description describing the feature itself.
Basic extends MonitoringFeature and it is used to distinguish between simple monitoring features, e.g., Event and Primitive, and complex monitoring feature, e.g., Function. It has one instance variable:

- **type**: it specify the type of the monitoring feature. In the case of Primitive monitoring features allowed types correspond to the Java primitive types, e.g., Long, Boolean, String. In the case of Event monitoring feature allowed types are defined by the SLA(T) Abstract Syntax document, e.g., REQUEST, RESPONSE, COMPUTATION.

*Primitive* monitoring features are used to advertise abilities to report about primitive information, e.g., cpu_load, logged_users, available_memory. Sensors are the typical components exposing this kind of feature. The class Primitive has two instance variables:

- **type**: holds the variable type. It can be, for instance, one of the Java standard primitive type. It can also be any other type defined in the SLA@SOI standard vocabulary.
- **unit**: holds the monitoring feature unit of measurement, e.g., mt, km, kg.

*Event* monitoring features are used to advertise abilities to report about service interactions or service state, e.g., service operation requests and responses, service failures. Sensors and Reasoners are the typical components exposing this kind of feature. The class Event has one instance variable:

- **type**: holds the event type, it can be REQUEST, RESPONSE, COMPUTATION, or any other event type defined in the SLA@SOI vocabulary.

*Function* monitoring features are used to advertise abilities to perform a computation and report its result, e.g., availability, throughput, response_time. Reasoners are the typical components exposing this kind of feature. The class Function has two instance variables:

- **input**: holds the list of the function input parameters
- **output**: holds the output parameter

**Monitoring system configuration**

SLA@SOI key functionality is SLAs run-time monitoring. To dynamically setup the whole framework many of its components need to be configured. The Monitoring System Configuration (MSC) model structures all needed information into a coherent inter-level representation.

A MSC instance can be created for two main purposes: to configure the monitoring system when a new SLA needs to be monitored, to perform adjustments to an existing configuration.

The MSC main class is **MonitoringSystemConfiguration**. There must exist one MonitoringSystemConfiguration instance per SLA. The uuid field uniquely identifies the instance and its values can be the same of the SLA id.
A MSC contains a list of **Components** whose attributes are:

- **uuid**: the unique component id. This id can be the SLA id, already unique with the SLA@SOI framework.

- **type**: the component type, e.g., SENSOR, EFFECTOR, REASONER. This attribute must be set accordingly with the kind of configuration to perform. If Component element contains SensorConfiguration elements then its type attribute must be set to SENSOR.

Each component contains one or more **ComponentConfiguration** instances uniquely identified, within the SLA@SOI framework, by their ids, i.e., the **configurationId** field. There are three different component configuration types:

**SensorConfiguration**: it is used for configuring sensor. It contains a monitoring feature name (or identification) and a list of receivers identified by their UUIDs. The **eventType** field tells the kind of event a sensor have to emit. For instance, a BPEL sensor A receives a configuration in order to report information about a service operation calls. This information must be sent to reasoning component B.

**ReasonerConfiguration**: it is used to configure a reasoning component. It contains the monitoring specification, i.e., the actual specification used by the reasoning component to start monitoring and SLA or a part of it. The monitoring results are sent to the list of receivers. For instance, the reasoning component A receives the specification MTTR<123. It means that the reasoning component have to start monitoring that expression and sending the results to the list of receivers.

**EffectorConfiguration**: it is used to configure an effector. Effectors, like sensors, are service-specific components, but opposite to sensors they are used for modifying service properties. Effectors know how to interact with services and perform requested actions, e.g., start/stop a service execution. The ability of effectors to perform run-time service configurations is very useful in multiple scenarios. For example service adjustments can be executed if required SLAs are
not satisfied in order to avoid penalties. Effectors are part of Y2 architectural, but their implementation is planned for Y3.
5 Architecture Specification

In this chapter, we present the top-level view of the SLA@SOI framework architecture, including its components and interactions.

After a first overview in Section 5.1, Section 5.2 provides the introduction to all top-level components. The role and purpose of the interaction stereotypes is explained in Section 5.3 in the context of broader interaction sequences. Last, this chapter provides details a cross-component aspects, namely the common Manageability approach (Sections 5.4).

5.1 Goals and Overview

The overall SLA@SOI Framework is conceived as a possibly distributed, hierarchical management system providing consistent SLA management across the service delivery stack.

At the highest level, we assume the operation of the SLA@SOI Framework serves ultimately to satisfy the goals of some business entity. Consequently, all management activities supported by the framework should eventually relate to the needs of the business entity.

Technically, the framework architecture is build along the following design goals:

1. Clear separation of concerns, e.g. having different components with clear responsibility on business, software, infrastructure, SLAs or 3rd party aspects.
2. Flexible configuration / setups, which can support a variety of scenarios and domains which
   a) require different setups of the framework, and
   b) already have (legacy) management functionality in place.
3. Simplicity, which is important to make the whole architecture understandable to a large audience and to make the actual framework adoptable for industrial use cases.

Design goals 1 and 2 are indispensable in order to support the different scenarios that are introduced by the different industrial use cases. Goal 3 is a more pragmatic goal but may in some cases also conflict with goal 2 as flexibility typically increases complexity. The main approach taken to resolve such conflicts is by providing default implementations of respective components or interaction channels.

In the remainder of the section, we first provide an overview of the architecture, its main components (Section 5.2) and interaction flows (Section 5.3). Finally, we elaborate the Manageability Fabric System (Section 5.4).

Top-level Architecture

In the following, we present the top-level view of the SLA@SOI framework architecture. It is derived from the general motivation and approach. For its representation, we chose a simplified version of UML component diagrams. Boxes represent components and connections represent stereotyped dependencies that translate to specific sets of provided and required interfaces.
Figure 17: Top-level view of the SLA@SOI framework.

Figure 17 illustrates the main components of the SLA@SOI framework and their relationships. On the highest level, we distinguish between the core framework, Service Managers (infrastructure and software), deployed service instances with their Manageability Agents, and Monitoring Event Channels. The core framework
encapsulates all functionality related to SLA management. Infrastructure- and Software Service Managers contain all service specific functionality. The deployed service instance is the actual service delivered to the customer, and is managed by the framework via Manageability Agents. Monitoring Event Channels serve as a flexible communication infrastructure that allows the framework to collect information about the service instance’s status. Further details are described in Section 5.3. In order to achieve a good generalization of the framework architecture, several components are realized as specialization of abstract components, namely SLA manager components and service manager components. The component hierarchy assumed for the top-level view is depicted in Figure 18.

![Component Specialisation Hierarchies.](image)

Please note that Figure 17 shows only one possible instantiation of the framework architecture with its types of components and its possible interactions. Thus, where appropriate, there will be different instantiations of the framework architecture for each use case of the B-line. For example, use cases might choose another way to structure the SLA/service hierarchy, to have specific managers for non-IT SLAs/services, or to create managers specialized for IT areas such as BPEL.

### 5.2 Main components

In the following, we briefly describe the main components of our framework. Detailed specifications of the components can be found in the deliverables of the respective work packages.

The **Business Manager** is responsible for controlling all business information and communication with customers (<<customer_relations>>) and providers (<<provider_relations>>). For example, it realizes the customer relation management (CRM) necessary to efficiently sell the offered services. Furthermore, the Business Manager governs the (Business-, Software-, and Infrastructure-) SLA Managers (<<control/track/query>>). For this purpose, SLA Managers have to adhere to business rules defined by the Business Manager ('control') and have to inform the Business Manager about their current status and activities ('track').

The **(Business-, Software- & Infrastructure-) SLA Managers** are responsible for the management of all SLA related concerns. The Business SLA Manager, Software SLA Manager, and Infrastructure SLA Manager are specializations of an abstract generic SLA Manager (cf. Figure 18). SLA Managers are responsible for the negotiation of SLAs, and for the SLA Management of services (subject to SLAs). All SLA Managers can act as "service customers"; negotiating SLAs with other SLA Managers inside the same framework, or with external (3rd party) service providers (including other framework instances). As "service providers" all SLA Managers can negotiate SLAs with other SLA Managers in the same framework. Only the Business SLA Manager, however, can negotiate with
customers who are external to the framework. To avoid confusion, we refer to external customers as "business-customers", and use the term "product" to denote the (SLA governed) services offered by the framework to business-customers. Product descriptions are published in a 'product catalogue' (accessible via <<query_product_catalog>>) maintained by the Business Manager. Once an SLA has been agreed with a customer, it is the responsibility of the Business Manager to send reports on SLA status to the customer. The <<negotiate/query/coordinate>> relation captures all framework internal negotiation and querying interactions. These negotiation part is equally used at business-level for the customer interaction (<<negotiate/coordinate>>) where business-level considerations (e.g. billing) are intercepted by the Business SLA Manager into the negotiation protocol. Finally, all SLA Managers can consult Service Evaluation to a priori evaluate the potential quality of a service (<<evaluate>>). This evaluation can be based on prediction, historical data, or predefined quality definitions, and supports the SLA Manager in finding service realisations with an appropriate quality.

Infrastructure- and Software Service Managers encapsulate all service-specific details. Both are specialisations of the abstract Service Manager concept (cf. Figure 18). Service Managers provide information about the service implementations supported, such as the service’s realization and its dependencies with other services (<<prepare_infrastructure_services>>, <<prepare_software_services>>). SLA Managers provision services using the management functionality of their corresponding ServiceManager (<<manage_software_service>>, <<manage_infrastructure_service>>). Furthermore, Service Managers control the service instances they have provisioned. SLA Manager can manipulate service instances via generic management functions provided by the Service Manager.

The Monitoring & Adjustment Management System provides the underlying fabric across different layers (i.e. across software and infrastructure layer) supporting the monitoring and management of actual service instances. The Monitoring Event Channel is the basic component via which arbitrary monitoring events (e.g. SLO violations) can be propagated to relevant SLA managers. Access to this channel is realized via the <<publish_event>> and <<subscribe_to_event>> interaction stereotypes. Manageability Agents support the actual configuration and management of service instances. The access to manageability agents for SLA managers is always mediated via a specific service manager. Due to the domain specific nature the interactions between service managers and manageability agents is represented by the <<native_service_management>> stereotype which is not further refined by this architecture. It should be noted that Manageability Agents need not necessarily run within the same administrative domain as the service instance but importantly the sensors and effectors, which are part of the Manageability Agent must reside in the same administrative domain of the service instance and have access to their related Manageability Agent via some communication mechanism.

Due to the distributed nature of the overall monitoring & adjustment management system, the overall system is specified in an integrated way in Section 5.4.

Please note that Figure 17 shows only one possible setup of the architecture. The framework itself supports a much more flexible combination of different SLA Managers and Service Managers. In general, the following component cardinalities can be assumed per framework instance:
### Component Table

<table>
<thead>
<tr>
<th>Component</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLA Manager</td>
<td>1..*</td>
</tr>
<tr>
<td>Service Manager</td>
<td>1..*</td>
</tr>
<tr>
<td>Service Evaluation</td>
<td>0..*</td>
</tr>
<tr>
<td>Business Manager</td>
<td>0..1</td>
</tr>
</tbody>
</table>

The minimal framework setup contains a single SLA Manager and its Service Manager. In such a scenario, no business functionality (such as CRM) is supported but the only focus lies on SLA negotiation, provisioning as well as monitoring and adjustment. Further setups are described in Section 6.

## 5.3 Component Interaction

In this section, we provide an extended example of how actual interaction sequences may take place. For this purpose, we assume that the framework is instantiated so that all components exist once as shown in Figure 17. The interactions are described in terms of sequence diagrams. The chosen representation is semi-formal as it tries for an individual interaction step to clarify (a) the relation to the top-level interaction stereotype and (b) the relation to the more detailed activity step (as it is specified within the respective specification of an interaction) that takes place. Both aspects are separated by a ":" resulting to the following description style: `<<name_of_top-level_stereotype: name_of_activity_step>>`.

The sequence diagrams shown in the following are still on an abstract level. Appendix C gives a detailed example for the specification of interactions involved in the integrated planning cycle for software services.

### 5.3.1 Negotiation interaction

In the following, we describe the basic steps of a customer-initiated negotiation process. We distributed the process among three sequence diagrams that show the interaction on the business layer (Figure 19), the negotiation process between software and infrastructure layer (Figure 20), and the creation of an agreement (Figure 21).
The sequence diagram in Figure 19 shows the interactions between a customer, a Business Manager and a Business SLA Manager. As a first step, the customer registers at the business manager, for example using the framework’s web portal. Then the customer browses through the product catalogue and inspects the SLA Templates of specific products. If the customer found an interesting product, he or she authenticates at the business manager and can now start the negotiation process. For this purpose, a negotiation request regarding a specific product is sent to the Business SLA Manager. The Business SLA Manager first checks the credentials of the customer before it passes his or her request to the other framework components for further negotiation with other SLA managers (cf. Figure 20). Once the Business SLA Manager received a set of potential SLA Templates from the other components of the framework, it calls back the Business Manager to adjust the templates for the particular customer. For example, power users get a discount of 20%. The customised templates are then returned as an individual offer to the customer. Finally, the customer selects one of the offers, adds his or her individual requests and triggers the creation of an agreement. During this process, the Business SLA Manager first calls the Business Manager to assess the incoming request. If the request is valid, the framework internal process for the creation of an agreement is triggered (cf. Figure 21). Once all necessary steps, such as the reservation of resources, have been taken the final SLA is again adjusted for the particular customer and an SLA is returned. The SLA contains all information necessary for the customer, including the service’s endpoint and the date and time when the service will be available. In the following, we elaborate the negotiation process between the software and infrastructure layer in more detail.
Figure 20: Negotiation interactions between software and infrastructure layer.

The sequence diagram in Figure 20 illustrates how the negotiation process continues from the Business SLA Manager across the different layers starting with the Software SLA Manager. To provide a set of possible SLA Templates to the Business SLA Manager, the Software SLA Manager queries the available service implementations from its associated Software Service Manager (SSM). In order to determine the quality of service that can be provided for the customer’s request, the Software SLA Manager needs to resolve all dependencies that the implementation may have. For example, one of the implementations may rely on an infrastructure service hosting its appliance. Therefore, the Software SLA Manager queries available SLA Templates from an Infrastructure SLA Manager to identify potential alternatives for the service needed by the service implementation. Based on the alternatives for the dependent services, the Software SLA Manager starts its internal planning. For this purpose, it calls the Service Evaluation to estimate the quality of service that can be achieved by a particular set of external services used by an implementation. Judging the achieved quality of service against the required one, the Software SLA Manager decides which alternatives are to be taken into account. For these alternatives, it starts the negotiation process with the Infrastructure SLA Manager which in turn reserves the necessary resources. Finally, the Software SLA Manager itself reserves all necessary software resources calling the corresponding Software Service Manager. When the negotiation process has been completed, the customer can select one of the returned SLA Templates, add his or her adjustments, and create an agreement described in the following.
Figure 21: Interactions during the creation of an agreement.

Figure 21 shows a sequence diagram for the interactions necessary to create an agreement among all involved parties. The customer triggers the creation of an agreement. Once the Business Manager has assessed its validity, the Business SLA Manager propagates the request to the Software SLA Manager. The Software SLA Manager first needs to ensure that all dependent services are available. For this purpose, it tries to create an agreement with the connected Infrastructure SLA Manager which (if the request can be handled successfully) commits the previous reservation and makes the corresponding infrastructure available. Once all dependent services have been acquired, the preparation on the Software Service Manager’s side can start. It books all the resources needed to provision the software (such as software licenses) and then schedules the provisioning of the service calling startInstance with the date and time when the service must be available. Finally, the agreed SLA is returned to the Business SLA Manager that returns it after a customization by the Business Manager to the customer. This concludes the negotiation process. Next the service needs to be provisioned which is described in the following section.

5.3.2 Provisioning interaction

In this section, we describe two variants of the provisioning process: explicit provisioning (Figure 22) and time triggered provisioning (Figure 23 and Figure 24). In the first case, a customer explicitly triggers provisioning. In the second case, it is executed implicitly by SLA managers based on the times specified in the SLA.
Figure 22: Explicit provisioning triggered by the customer.

Figure 22 shows a sequence diagram illustrating how provisioning can be triggered explicitly by the customer. Once triggered, the provisioning request is propagated from the Business SLA Manager, to the Software SLA Manager, and to the Infrastructure SLA Manager. In the sequence diagram shown, the infrastructure services required for the software service are provisioned first. Once they are available, the Software SLA Manager can proceed and provision its services. Finally, the Business SLA Manager can finish the provisioning sequence. When the request returns to the customer, he or she can immediately use the requested service. However, provisioning is often a lengthy and time-consuming process. Therefore, customers do not want to trigger provisioning manually, but require the service to be available at a particular time. This can be achieved by the implicit provisioning described in the following.

In order to enable implicit, time-triggered provisioning, detailed planning of the provisioning process by the SLA Managers is essential. SLA Managers must communicate the time necessary to provision the requested services. Based on these times, a provisioning schedule is determined. Each SLA Manager schedules its provisioning actions according to this schedule. Furthermore, SLA Managers have to notify each other as soon as individual services are available. The notifications are necessary to cope with potential delays during provisioning.

Figure 23: Implicit provisioning of infrastructure services triggered by a timer inside the Infrastructure Manager.

Figure 23 shows a potential interaction flow for the time-triggered provisioning of an infrastructure service. At the point in time that has been determined by the Infrastructure SLA Manager, the Infrastructure Manager starts the creation of a
new instance of the requested infrastructure service. This can translate, for example, into booting a virtual machine. Once the creation as well as the internal book keeping has been finished the Infrastructure Manager publishes a message on the event bus notifying all involved parties that the infrastructure service (e.g. the virtual machine) is available now. In Figure 23, the Infrastructure SLA Manager receives the event and in turn notifies the Software SLA Manager (as it has been agreed on in the SLA).

![Diagram](image)

**Figure 24: Implicit provisioning of software services triggered by the Software Service Manager.**

The process is similar for the provisioning of software services. In Figure 24, the provisioning of the software service is triggered at a particular point in time. Alternatively, it may be triggered by an event of the Infrastructure Manager. In any case, the Software Service Manager creates a new instance of the software service. The creation of the new service instance can be realised by very different means. For example, it can be done by deploying and activating an application on a middleware platform or by creating a new tenant in a multi-tenant system. Once the service has been instantiated and configured, the Software Service Manager puts an event on the message bus signalling the successful creation of the software service to all involved parties. In Figure 24, the Software SLA Manager is triggered by this event and notifies the Business Manager about the successful service creation using its tracking interface. The Business Manager can propagate this information further to the customer. The customer now knows that his or her service is accessible and ready for usage.

In the following sections, we illustrate the main scenarios at runtime, namely the customer or framework triggered reporting, the processing of SLA violations, and customer-initiated renegotiation.
5.3.3 Adjustment & Reporting interactions

Interaction 1 and 2 in Figure 25 illustrate two different ways of how information about a service’s status can be brought to the customer. In the first case the customer pulls information from the Business Manager using an explicit call (getReport). In the second case, the framework pushes information about a service’s status to the customer in regular intervals or in case of specific events. Both scenarios are valid for the framework. However, the terms and conditions have to be defined explicitly in the SLA.

Interaction 3 in Figure 25 illustrates the notification of the customer in case of an SLA violation observed by the framework. The Manageability Agent and Monitoring System continuously observe the service’s quality. If the observed quality is worse than the quality agreed in the SLA, it publishes the resulting violation on an Event Channel. In the scenario depicted here, the SLA Manager retrieves the notification and initiates countermeasures (i.e., calls manage_<T>_service, where <T> can either be infrastructure or software in this case). If successful, the countermeasures allow the framework to restore the quality of service agreed on in the given SLA. The violation and the taken countermeasures are reported to the Business Manager via the <<tracking>> interaction. The Business Manager records the violation and decides based on the severity of the violation and the success or failure of the countermeasures whether to report the violation to the customer or not. Sometimes the violation cannot be handled by the adjustment of the service stack and renegotiation with the customer is necessary. For example, this can be the case if the customer violates the SLA by putting too much load on the system. In the following, we describe the renegotiation process in more detail.
5.3.4 Renegotiation interactions

Renegotiation interactions can be triggered either by the Business Manager or the customer. In the first case, the Business Manager may observe an SLA violation (caused by the framework or the customer) that cannot be handled by the framework itself. In this case, the Business Manager triggers an internal (re-)negotiation process of the framework and its connected SLA Managers to solve the problem. Based on the possible solutions found, the renegotiation with the Customer is initiated.

In the second case, the customer requires changes in the quality agreement made in the SLA. Therefore, he or she triggers renegotiation with the Business SLA Manager. Basically, the interactions for renegotiation are similar to the ones shown in Figure 19 and Figure 20.
5.4 Manageability approach

In this section, we provide a detailed introduction of the manageability approach employed inside the SLA@SOI framework. Monitoring and adjustment are to a large extent cross-cutting concerns of our framework. As such, a detailed understanding of the manageability part of our architecture is essential. Therefore, this deliverable captures its most important aspects in detail.

Manageability, monitoring and adjustment systems collaborate to provide reliable services to customers, assuring the agreement between customer and service provider. Due to their cross-cutting nature, they serve as a fabric underlying the rest of the framework and support the service and SLA management in a flexible but still harmonized way.

Figure 27 shows the main components of the manageability system, while Figure 27 shows a typical interaction.

Figure 27: Manageability, Monitoring and Adjustment system architecture
The purpose of the Monitoring and Adjustment Management System (MAMS) is to compliment the service manager by providing the means to monitor a provisioned service instance in the context of the requested guarantee terms contained in the SLA. The system needs to be general enough such that its architecture is applicable to both software and infrastructure service managers. In the following we describe the key components of the MAMS (printed in blue in Figure 27).

The MonitoringManager (MM) coordinates the automatic configuration of the monitoring system. It decides, for any SLA it receives, which is the most convenient monitoring configuration according to configurable selecting criteria. A monitoring configuration determines which components are to be configured as well as their configurations.

Adjustment (PAC) collects information from the Low Level Monitoring System, analyzes the incoming events, and triggers the best corrective or proactive action in case of (potential) problems. If the PAC cannot solve the problem at a local level, it escalates the issue to a higher level. In case of an SLA violation, the PAC can trigger re-planning, re-configuration and/or alerts at higher levels. These capabilities are considered to be important in order to guarantee best user perception preserving underlining resources.

The LowLevelMonitoringSystem is a central entity for storing and processing monitoring data. It collects raw observation data, processes them, computes derived metrics, evaluates rules, stores monitoring data, and offers the data to other components (accessible through Service Managers). It is general by design and thus capable of supporting infrastructure and software services as well as other use cases.

Sensors collect information about a service instance. Their design and implementation is domain-specific. Sensors can be injected into a service instance, e.g., by direct instrumentation or it can intercept service invocations outside the service instance. Sensors send the collected data to the communication infrastructure. Furthermore, other components can request (query) information from it. Various kinds of Sensors exist that collect different kinds of information. All sensors implement a common interface that provides methods for starting, stopping, and configuring a Sensor.

Effectors are components for configuring the behavior of a service instance. Their design and implementation is domain-specific. Effectors can be used to configure a service instance during its provisioning as well as during runtime. Effectors are the main means for the execution of adjustment actions.

In the following, we describe the concepts for manageability, monitoring, and adjustment in more detail.

### 5.4.1 Manageability

In the SLA@SOI framework we provide a unified manageability interface for managing services through appropriately deployed sensors and effectors. The interface is implemented by a generic manageability agent component, a gateway to managing one or more services, hiding the domain-specific details of the the management itself.

The Manageability Agent provides two generic interfaces: the IManageabilityAgent interface and the IManageabilityFacade interface. The generic Manageability Agent component implements the IManageabilityAgent interface, which provides a gateway for accessing services that need to be managed. In particular it allows the framework to:

- start a new service (method `startServiceInstance`)

---

SLA@SOI – FP7216556  Deliverable D.A1a – Framework Architecture  Page 54 of 88
- stop a service (method `stopServiceInstance`)
- obtain a list of the services being managed through the particular ManageabilityAgent instance (method `getEndpoints`)
- obtain a service instance specific façade component for managing the service (method `getManageabilityAgentFacade`)

Once a façade object is obtained, it can be used to directly manage a service instance through the `IManageabilityAgentFacade` interface. In order to do so, the façade’s internal implementation is necessarily domain-specific. The façade allows the framework to:

- get a ServiceInstance object for the service instance being managed through the façade (method `getInstance`). The ServiceInstance object is part of the Service Construction Model, and contains information for uniquely identifying the instance.
- configure the sensors that will be used to collect run-time data regarding the service instance, and the effectors that will be used to control the service instance (method `configureMonitoringSystem`)
- deconfigure the sensors and effectors associated with the service instance (method `deconfigureMonitoring`)
- get a list of sensor subscription data for the sensors active on the service instance (method `getSensorSubscriptionData`). The sensor subscription data allow the framework to understand what data will be gathered, and how they can subscribe on the XMPP event bus to get them.
- execute a control action on the service instance (method `executeAction`). This allows the framework to invoke domain-specific adjustment actions on the service instance.

Note that more details regarding the ManageabilityAgent and the interfaces it provides can be found in Section 7 of the deliverable “DA3.a - SLA-aware Service Management”.

### 5.4.2 Monitoring

Monitoring provides the following functionalities:

1. Check whether the SLA can be monitored.
2. Perform instrumentation of standard resources (standard infrastructure and SW).
3. Collect instrumentation measurements (from several sources).
4. Store them (HistoryInfoStore).
5. Calculate QoS terms (defined in service configuration). These terms are
6. Expose all monitoring data to other components (PAC, GUI, etc.)
7. Evaluate agreement terms.
8. Send notifications in case of SLA Violations and Warnings

Functionalities 1,7,8 are implemented by SLA Level Monitoring System (SLMS), while functionalities 3-6 are implemented by Low Level Monitoring System (LLMS). Instrumentation is performed by Ganglia and by custom sensor modules.
**Instrumentation**

For all further monitoring purposes basic instrumentation data are needed from all monitored resources and services. A single unit of instrumentation data is called **observation** (also measurement). A type of observation is called **metric**. It typically refers to a single resource/service, but can also refer to arbitrary group. Example metrics: cpuUsage on VM1, request/response of a service MyService.

Depending on the type of metric, instrumentation can be done by standard solution or a custom module (called **Sensor**). In both cases the observations has to be send to LLMS by invoking LLMS Remote API.

Instrumentation is a continuous distributed process, so the observations are to be sent asynchronously with minimal delay after they are observed.
Low Level Monitoring System

The inner structure of LLMS, its interfaces and interactions are shown on Figure 29.

Figure 29: LLMS inner structure, interfaces and interactions
LLMS comprises the following components:

- ReasoningEngine
- DerivedMetricsComputation
- HistoryInfoStore
- ConfigStore
- GenericReporting

**ReasoningEngine** performs reasoning - rule evaluation based on received and historical observations and static parameters. The rules can be data triggered (when new observations arrive) or time triggered (periodic). The action statement of rules can contain arbitrary method call, most typical being sending a (specific) notification or calculating a derived metric observation.

**DerivedMetricsComputation** is used to define and compute some derived metrics that are computed from basic (observed) metrics. Derived metrics are defined using rule definition and computed in runtime.

**HistoryInfoStore** stores the history of raw observation, derived metrics values and sent notifications. It also does the history aggregation of old data.

**ConfigStore** store the resource and service specific configuration (metric parameters and rules).

**GenericReporting** is the common core of domain aware reporting based on monitoring data.

The following public interfaces are implemented:

- **IConfigureMonitoring** – provision, reprovision, free monitoring request
- **IMonitoringDataAndHistory** – query metrics, their last values and history
- **IMonitoring2Resource** – set metric specific sensor parameters
- **IResource2Monitoring** – register new metrics, send observations

The system is configured through IConfigureMonitoring interface at runtime. ILandscape is used to access the data in ServiceManager Landscape databases. The data is used by LowLevelMonitoringManager and exposed through IMonitoringDataAndHistory interface.

Generic monitoring console is implemented by GenericReporting component that is an independent and optional part of the monitoring system. It collects data though IMonitoringDataAndHistory interface and present them in a user friendly way (GUI). Reports export will be supported through IReporting interface.

For data exchange and other communication common PubSub component will be used, so MonitoringEventChannel will be implemented as general PubSub component.
To provide effective monitoring support when replacements of the services deployed in a service based system (SBS) occur at run-time it is necessary to be able not only to check whether the monitorability of the required SLA terms and conditions is affected by the changes but also to modify the deployed monitoring infrastructure in order to ensure the continuous execution of the required run-time checks. Existing monitoring environments and approaches, however, do not offer these capabilities. To address this gap, SLA@SOI is developing a novel SLA layer monitoring system (SLMS). A key characteristic of this system is the separation of the actual service monitoring from the assessment of SLA monitorability, and the dynamic set up of the monitoring resources (i.e., sensors, effectors, and reasoning components) for checking an SLA.

A key characteristic of the approach underpinning the design of SLMS is the distinction between two key layers in service provision, namely the SLA management and service management layers.

- **SLA management layer**: from a monitoring perspective, it incorporates the mechanisms required for performing the SLA monitorability checks and the dynamic set up of monitoring infrastructures that can enable the monitoring of an SLA.
- **Service management layer**: it incorporates sensors and reasoning components required for service event capturing and performing the actual SLA checks.

SLMS key components are: Monitoring Manager (MM), Sensors, Effectors, and Reasoning Components.
Monitoring Manager (MM) coordinates the automatically configuration of the monitoring system. It decides, for any SLA it receives, which is the most convenient monitoring configuration according to configurable selecting criteria. A monitoring configuration tells which components to configure and how their configurations. Components receiving monitoring configurations are, typically, Sensors, Effectors, and Reasoning Component Gateways.

The MM check for monitorability function checks whether an SLA is monitorable with respect to the available monitoring system components. The idea underlying the check for monitorability is to delegate, to many reasoners, the monitoring of an SLA in the case in which one reasoner cannot monitor an SLA only. The SLA monitoring can be delegated also in case in which other criteria are taken into account, e.g., the cost of using many reasoners instead one only.

The MM interacts with the POC. The POC requests MM to check for monitorability. All the information required by the MM is passed to it as part of a ServiceBuilder model. In doing so, a query from the MM is not required and the MM simply operates upon the inputs supplied to it by the POC.

For detailed information about MM refer to D.A3b deliverable.

Sensors are components that implement the SensorInterface interface. A sensor is a component that gives access to specific infrastructure service information.

```
<<interface>>
SensorInterface

+ getURI() : URI
+ setURI(uri : URI) : void
+ getSensorMetrics() : ArrayList<MetricInterface>
+ configure(sensorConfigurationInterface : SensorConfigurationInterface) : boolean
+ start() : boolean
+ stop() : boolean
```

**Figure 31: Sensor interface**

For instance, let’s consider a software service providing a shopping cart for an e-commerce application. One of the operations provided by the service interface is `addItem(Item item):void`. The operation takes as input the item a user want to add to its cart and does not return any value. For monitoring purpose, this method has been instrumented with code for collecting the method’s input parameter. The collected data, depending on the instrumentation configuration, can be processed and stored into a database. In the example, the shopping cart service has a configurable instrumentation; the sensor component provides an interface for configuring the service instrumentation, e.g., enabling or disabling the collection of data.

As another example, let’s consider the infrastructure service CPU. We are interested in collecting information about its workload. To do that there are many tool available, e.g., the command-line application `top` in UNIX based systems. This application produces information in its own format, therefore, we need to implement a component interacting with the `top` application and collecting the application’s output. This component will implement the SensorInterface interface.

Reasoning components are monitoring system components implementing the ReasoningComponentGateway (RCG) interface. A RCG mediates the interactions that take place between the SLA@SOI components and the reasoning engine (RE). A RCG provides a set of operations for configuring, starting,
It also provides operations for retrieving historical monitoring data. A RCG receives monitoring configurations expressed in a *lingua franca* and translates them into RE specific monitoring configurations.

**Figure 32: Reasoning component gateway interface**

The ReasoningComponentGateway interface exposes the following operations:

```java
public ComponentMonitoringFeatures getComponentMonitoringFeatures();
public void setComponentMonitoringFeatures(ComponentMonitoringFeatures compMonitoringFeatures);
public String getUuid();
public void setUuid(String value);
public ArrayList<ReasonerConfiguration> getConfigurations();
public void removeConfiguration(String configurationId);
public void addConfiguration(ReasonerConfiguration reasonerConfiguration);
boolean startMonitoring(String configurationId);
boolean stopMonitoring(String configurationId);
```

**Figure 33: Reasoning component gateway methods**

Each RCG has a unique UUID. The UUID is used to retrieve information about a RCG from the software landscape. Each RCG exposes a list of monitoring features through a ComponentMonitoringFeatures object. A monitoring feature expresses the ability to perform an operation or an action.

A RCG receives its configurations via ReasonerConfiguration objects. Each configuration has its own unique configuration identification. The specification field holds the actual monitoring configuration. Ad-hoc translator, if needed can translate this specification to specific monitoring rules. The translation takes place RCG side.

A reasoner configuration already sent to a RCG can be explicitly started/stopped using the startMonitoring/stopMonitoring methods that take in input the configuration identification.
5.4.3 Adjustment

SLAs can be seen as containers of the functional and non-functional properties that both parties, customer and provider, agree specifying its obligations and rights during the service lifetime. However, this also represents a responsibility for the service provider, since it motivates the need of the implementation of an SLA enforcement process. For this reason, the SLA@SOI framework includes a specific component to care about the enforcement aspects of the service: the Provisioning and Adjustment Component (PAC).

Since the quality of the service must be guaranteed at all the levels of the IT stack, this component is placed in all the layers: at software and infrastructure levels, being part of the corresponding SLAManagers, and at the business level as part of the Business Manager. At each level, PACs will receive different inputs, and will trigger different domain-specific actions.

The software and infrastructure PACs subscribe to an event bus (Monitoring Event Channel) in order to receive events from the monitoring system indicating that a SLA has been violated. Upon the reception of these events, the PACs may behave differently: they may take an action on its own given sufficient certainty about the problem and its solution (e.g. reprovisioning of resources, software reconfiguration) or they may request the Planning and Optimization Component to create a new plan, or even to re-negotiate an existing SLA. They also possible inform the business PAC about the problem that has occurred.

On the other hand, the Business PAC receives violations from the lower level adjustment components, and takes decisions based on business criteria, such the re-negotiation of the agreement with the end-customer, the termination of the offered service or even, in a multiprovider environment, may decide to change the 3rd party provider in which it was relying for a given service.

For the implementation of the above described behaviour, the Provisioning and Adjustment Component offers the <<plan>>, <<control>> and <<query>> interactions, and uses <<query>> interface of the SLA registry, <<subscribe_to_event>> of the MonitoringEventChannel, and the <<manage_T_services>> interface offered by the Service Managers. The later is by definition domain specific, and splits into <<manage_software_services>> at software level and <<manage_infrastructure_services>> at the infrastructure layer.

The <<plan>> interaction subsumes the functionality that is necessary for the communication with the Planning and Optimization Component. It includes interfaces for the POC to order the execution of a plan, as well as to inform about the status of the execution of the plan back to the POC. Furthermore, when a problem appears that cannot be solved locally by the PAC, a re-planning can be triggered using the IReplan interface.

The <<control/track>> interaction has been designed to take into account business level criteria at the lower levels. For instance, the service provider can decide that it is better to accept violations in some SLAs to give priority to others based on business impact. It is even possible that the service provider decides to prevent breaches for some SLAs or specific QoS metrics while applying reaction upon violations for others. To this end, the <<control>> interaction allows the Business Manager to retrieve the current adjustment policies, as well as to set a new list of policies. On the other hand, the <<track>> interaction allows the communication of violations from the lower software and infrastructure levels to the Business Managers. This allows the business layer to be aware of the
problems in the underlying levels, and to calculate and apply the penalties derived from the malfunctioning of the services under its control.

The <<query>> interaction is implemented in the software and infrastructure PACs, and allows an external entity, namely the Business Manager, to query the SLA violations as well as the historical monitoring information. In the second case, the PAC acts as a proxy to retrieve the monitoring data from the Low Level Monitoring System (LLMS) database. This feature is important not only from a business adjustment perspective, but also for the service provider when constructing the reports about the performance of the service.

A more detailed description of the Provisioning and Adjustment Component, also in the context of business, software and infrastructure services, can be found in deliverables D.A2a [16], D.A3a [17], D.A4a [18], and D.A5a [19].
6 Adoption Examples

This chapter serves for illustrating the presented architecture by showing how it can be applied in real-world scenarios.

6.1 ERP Hosting

Background

The Enterprise Resource Planning Hosting (ERP Hosting) solution is targeted at SMEs that cannot afford expensive ERP solutions that include software, hardware and constant support. The service provider provides applications as services (SaaS, Software as a Service) using an online portal. The portal also provides customers with tools for specification of business requirements (providing functional and non-functional information) as well as SLA parameters.

Based on this input, the service provider plans the capacity required to satisfy all requirements, especially the Quality of Service guarantees. Once terms are formally and legally agreed by both entities, the service provider provisions the required infrastructure. It also provides monitoring capabilities for all components (infrastructure, middleware, services) to facilitate appropriate adjustment.

The internal management of the offered SaaS solution is broken down into four separate layers, each of them realizing service offerings of different abstractions and governed via dedicated SLAs. The four layers are:

Solution layer: covering the complete solution offered to the customer including the actual software but also additional support services.

Application layer: covering the offered business applications (software), i.e. the enterprise resource planning functionality.

Middleware layer: covering generic middleware software components such as an application server, database, or enterprise service bus. These can be used across different applications.

Infrastructure layer: covering the actual physical compute infrastructure including virtualized counterparts of resources.

Further details on this industrial use case are provided in [8].

Architecture

The generic framework architecture can be applied for the ERP hosting scenario in the following way. As the use case comprises four different types of services (support, application, middleware, infrastructure) we created dedicated service managers for all of them. From the perspective of the administrative organization, we consider a separation into three different departments, namely business, software and infrastructure department. Consequently, we establish a separated SLA manager for each of these departments. However, for the software part we subdivided the SLA manager into two separate components ones (Application SLA Manager and Middleware SLA Manager) in order to clearly separate the distinct planning and provisioning logic for both areas.

The Business SLA manager is responsible for the overall offering. It uses the support service manager for planning/managing the human support services.
The Application SLA manager is responsible for the SLAs of the complete software application. It uses the Application Service Manager, for managing actual ERP logic and Service Evaluation in order to assess the quality of the different possible software offerings.

The Middleware SLA Manager is responsible for the middleware layer incl. for example the application server.

The Infrastructure SLA Manager is responsible for the in-house IT resources but can also contact external cloud providers to acquire resources for burst scenarios. It uses an infrastructure service manager that deals with all the internal IT resources.

The following figure gives an overview of this adoption architecture. For sake of simplicity we do not show all the actual interaction stereotypes from the generic architecture. Also the manageability and monitoring subsystem is not explicitly shown but considered here as underlying fabric of the service managers.

**Figure 34: Adopted framework architecture for ERP Hosting use case.**
6.2 Enterprise IT

Background

Focusing primarily on Infrastructure as a Service (IaaS), and building on the year one Enterprise IT use case specification [15] the Enterprise IT lab demonstrator aims to address three scenarios, or business challenges, which build on each other and represent real barriers for the enterprise today. Using Key Performance Indicators (KPIs) we evaluate the performance of the lab demonstrator in the areas of:

- IT enabling the Enterprise
- IT Efficiency
- IT Investment/Technology adoption

The three scenarios are briefly re-introduced here.

The first scenario, titled “Provisioning”, responds to the issue of efficient allocation of new services on IT infrastructure, SLA negotiation and provisioning of new services in the environment. The second scenario, “Run Time”, deals with day-to-day, point in time operational efficiency decisions within the environment. These decisions maximise the value from the infrastructure investment. The final scenario, “Investment Governance” builds on the first two to demonstrate how they feed back into future business decisions. Taking a holistic cost view, it provides fine-grained SLA based data to influence future investment decisions based on capital, security, computational power and energy efficiency.

In year two of the project we have focused on the implementation of a hardware test bed on which to demonstrate and evaluate the effectiveness of the framework in an Enterprise IT setting.

Architecture

The generic framework architecture can be customised and applied to the Enterprise IT use case as shown in Figure 35.

Interaction with the system is achieved through a web based UI where both customers and administrators have an interface to the framework components. The Enterprise IT SLAT defines use case specific agreement terms that are loaded by the SLA manager to provide the inputs to provisioning requests. In year 2, provisioning requests consist of PaaS services. Therefore, the use case does not need to implement the software SLA Manager. Software services are of course possible by choosing a virtual machine template that contains pre-loaded applications, but the quality of service of these is not part of the agreement terms in year 2.

The SLA Manager passes service-provisioning requests to the Infrastructure SLA Manager whose role is to carry out the creation of the new virtual machines that constitute the service along with monitoring and reporting for that service.

In year 2, the use case has implemented the basic functionality required for the provisioning scenario and some of the features needed to evaluate the runtime scenario. The architecture will be extended in year 3 to realise the remaining features, including the creation of plug-ins to support additional heterogeneous hypervisors, such as VMWare.
Further details on this industrial use case are provided in [9].

6.3 E-Government

Background

The eGovernment use case applies the SLA@SOI architecture to the management of a hybrid service, which involves both automatic and human based activities, as it is typical in the Government domain.

The considered service allows the citizens to receive medical treatments and transport means to move from home to the treatment place, and to book them by phone.

Such a Mobility & Health Service is provided by the so called “Citizen Service Center”, placed in the Italian region of Trentino, and is regulated by a SLA with the local Government (“Provincia Autonoma di Trento”). The service is composed by a Contact Service provided by an internal department of the Citizen Service Center that allow the citizen to book the treatment and trips, from Medical Treatment Services provided by several external Health Care Structures and from Mobility Services provided by specific providers. The implementation of the Contact Service is based on internal phone call operators (human operators or automatic answering machines) and, when needed, also on external Call Centers.

The Government chooses the possible Health Care Structures and Mobility Providers, and their services are regulated by SLAs with the Government. The specific provider used at each execution of the Mobility & Health Service is selected by the Citizen Service Center on the base of the Citizen’s choices and preferences.
The business relationship with the external Call Centers is under the complete control of the Citizen Service Center, which directly establishes the SLA with the selected Call Center.

Therefore, the Citizen Service Center is responsible for selecting, coordinating, and monitoring the component services needed to provide the Mobility & Health Service, under the constraints established by the SLAs established by the Government. The monitoring activity includes the assessment of the Citizen satisfaction that is also performed using human operators provided by the internal department or by the external Call Centers. The involved SLAs includes the payment of penalties in case of violations of the constraints.

The Citizen Service Center wants to adopt an SLA@SOI based software application in order to automate activities that currently are performed manually, such as the coordination between the treatment and mobility service, the aggregation of monitoring data to evaluate the quality of the composed service, the allocation of the phone call operators, the selection of the mobility providers based on the matching between the Citizen request and the SLA established with the providers, the management of the requests of renegotiations coming from the Government, by planning the resources needed by the internal department and re-negotiating SLA with the external Call Center.

The application offers specific dashboards that report the satisfaction of their SLAs and the penalties to pay by each part to both the Government and to the Internal Department of the Citizen Service Center. Moreover an interface is offered to the Government to start a renegotiation, and another interface is offered to the internal department of the Citizen Service Center to see the resource planning and allocation established by the system.

The Health Care Structures and Mobility Providers provide sensors for monitoring their services. Any external Call Center offers a negotiation interface that is compliant with the SLA@SOI standard, but no one of the external providers is required to adopt the SLA@SOI framework.

Further details on this industrial use case are provided in [11].

**Architecture**

The generic framework architecture can be applied for the eGovernment use case as shown in Figure 36.

The coordination of Health & Mobility Service is performed through a BPEL process that is managed by a corresponding Software Manager; in particular it controls the dynamic binding of external service providers. A corresponding Human Service Manager manages the human resources.

While the Health & Mobility Service involves both human and software components, the quality constrained in the SLAs does not depend in a significant way from the quality of the software or infrastructure hardware components. Therefore, one single SLA Manager at business level is sufficient to manage the negotiation of the Government and with the external Call Center and to manage the SLAs with the other external providers.

Moreover the SLA Manager uses a Service Evaluation Component to predict at negotiation time the quality of the service obtainable with the available internal resources in order to determine the SLA to negotiate with the external providers.

The runtime prediction feature of the monitoring system is used during the execution of the service to warn about possible violation of the guarantee terms in the SLA of the Contact Service and to trigger the automatic adjustment of internal human operators.
Figure 36: Adopted framework architecture for eGovernment use case.
7 Conclusions

7.1 Summary

This document presented the reference architecture for the SLA management framework that has been created within the second year of the European research project SLA@SOI.

The reference architecture has been developed based on the requirements of four industrial use cases as well as other (project-external) sources (see also [14]).

The architecture constitutes a clear innovation in its ability to mediate across layers, to support arbitrary domains, and to address the complete SLA and service lifecycle.

This document provided a complete overview of the framework architecture including a high-level overview, a detailed discussion of relevant foundational concepts, a description of the modelling foundation i.e. the most important metamodels that are shared between different components of the architecture, and the actual architecture overview with its building blocks, components and interactions.

The overall architecture is well synchronized and consistent with the more detailed architectures of the various main framework modules. Details on these module-specific aspects can be found in deliverables D.A2a-D.A6a.

The technical architecture has been successfully implemented into an SLA management software framework. Details about the technical/scientific evaluation of the SLA framework can be found in deliverable D.B1a.

Finally, the software framework has been successfully applied in four industrial use cases.

7.2 Outlook on Future Work

Following the broadness of the specified architecture and its applicability to four industrial use cases we expect that the developed architecture will remain largely stable for the next project phase.

Of course adjustment and optimization will be incorporated into the architecture, as these become clear in the detailed evaluation conducted by use cases but also based on ongoing external feedback that is collected.

Significant work is definitely needed to support framework adopters. Therefore, we plan to invest quite some effort into the proper documentation of the architecture as well as into guidelines for its application.
8 References

[1] SLA@SOI project: SLA@SOI Glossary. Annex to this deliverable D.A1a Framework architecture (full lifecycle). July 2010


[8] SLA@SOI project: Deliverable D.B3b Lab Demonstrator ERP Hosting. September 2010

[9] SLA@SOI project: Deliverable D.B4b Lab Demonstrator Enterprise IT. September 2010

[10] SLA@SOI project: Deliverable D.B5b Lab Demonstrator Service Aggregator. September 2010


[16] SLA@SOI project: Deliverable D.A2a Business SLA management. September 2010

[17] SLA@SOI project: Deliverable D.A3a SLA-aware service management. September 2010

[18] SLA@SOI project: Deliverable D.A4a SLA-aware infrastructure management. September 2010
[19] SLA@SOI project: Deliverable D.A5a Foundations for SLA-management. September 2010


## Appendix A: Glossary

The following list shows the most important entries of the SLA@SOI glossary. Note that terms that are specific for the current document and not part of the overall project wide glossary are marked with an asterix *.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agreement Initiator</td>
<td>An agreement initiator is a party to a service level agreement. The initiator creates and manages an agreement on the availability of a service on behalf of either the service customer or service provider, depending on the domain-specific signalling requirements.</td>
</tr>
<tr>
<td>Agreement Offer</td>
<td>An offer is the description of the agreement relationship that is sent from agreement initiator to agreement responder during agreement creation, indicating the relationship which the initiator would like to form.</td>
</tr>
<tr>
<td>Agreement Responder</td>
<td>The agreement responder is a party to a service level agreement. The responder implements and exposes an agreement on behalf of either the service provider or service customer, depending on the domain-specific signalling requirements.</td>
</tr>
<tr>
<td>Agreement Template</td>
<td>An agreement template is an XML document used by the agreement responder to advertise the types of offers it is willing to accept.</td>
</tr>
<tr>
<td>Agreement Term</td>
<td>Agreement terms define the content of a service level agreement.</td>
</tr>
<tr>
<td>Business Service</td>
<td>A business service is exposed/invoked via at least some non IT elements.</td>
</tr>
<tr>
<td>Business Manager</td>
<td>A specialization of service provider: person that defines the SLATs of products and joins available services in a product.</td>
</tr>
<tr>
<td>External Service</td>
<td>External services are exposed across the boundaries of an organization, i.e. across at least two administrative domains.</td>
</tr>
<tr>
<td>Framework Administrator</td>
<td>A specialization of service provider: person that configures/adapts the SLA@SOI framework for a specific application.</td>
</tr>
<tr>
<td>Guarantee Term</td>
<td>Guarantee terms define the assurance on service quality associated with the service described by the service definition terms. They refer to the service description that is the subject of the agreement and define service level objectives, qualifying conditions and business value expressing the importance of the service level objectives.</td>
</tr>
<tr>
<td>Hybrid Service</td>
<td>A hybrid service is a set or bundle of other services where all these services are exposed to the customer but have different service interface types (e.g. an IT service and a business service).</td>
</tr>
<tr>
<td>Infrastructure Manager</td>
<td>A specialization of infrastructure provider: person/system that is interested to measure and control infrastructure properties.</td>
</tr>
<tr>
<td>Infrastructure Provider</td>
<td>A specific kind of service provider that focuses on the provisioning of infrastructure services.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Infrastructure Service</td>
<td>An infrastructure service is a specific IT service which exposes resource/hardware-centric capabilities.</td>
</tr>
<tr>
<td>Internal Service</td>
<td>Internal services are exposed within the boundaries of an organization, i.e. within one administrative domain.</td>
</tr>
<tr>
<td>IT Service</td>
<td>An IT service is exposed/invoked by means of information technology. Specific classes of IT services may be software services, infrastructure services or media services.</td>
</tr>
<tr>
<td>Offered Service</td>
<td>An abstract service (more precisely: service type) which is offered by a specific Service Provider to its Service Customers.</td>
</tr>
<tr>
<td>Operation Level Agreements</td>
<td>A specification of the conditions under which an internal service or a component is to be used by its “customer”.</td>
</tr>
<tr>
<td>Service</td>
<td>A means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks.</td>
</tr>
<tr>
<td>Service Concreteness</td>
<td>The stage a service reaches over time from a fully abstract type to actually instantiated.</td>
</tr>
<tr>
<td>Service Consumer</td>
<td>Person(s) who actually consume/use the provided services. Typically they belong to the service customer.</td>
</tr>
<tr>
<td>Service Customer</td>
<td>Someone (person or group) who orders/buys services and defines and agrees the service level targets.</td>
</tr>
<tr>
<td>Service Description Term</td>
<td>Service Description Terms describe the functionality that will be delivered under the service level agreement. The agreement description may include also other non-functional items referring to the service description terms.</td>
</tr>
<tr>
<td>Service Exposure</td>
<td>Services can be exposed either internally (within the same administrative domain) or externally.</td>
</tr>
<tr>
<td>Service Implementation</td>
<td>A service implementation is a possible concrete realization of a given service type.</td>
</tr>
<tr>
<td>Service Instance</td>
<td>A concrete realization of an offered service which is ready for consumption by service users. It relies on the instantiations of all the resources required for a given service implementation.</td>
</tr>
<tr>
<td>Service Interface Type</td>
<td>Describes the nature of an actually exposed service, i.e. about the nature of his invocation interface.</td>
</tr>
<tr>
<td>Service Level Consequence</td>
<td>An action that takes place in the event that a service level objective is not met.</td>
</tr>
<tr>
<td>Service Level Agreement</td>
<td>An agreement defines a dynamically-established and dynamically managed relationship between parties. The object of this relationship is the delivery of a service by one of the parties within the context of the agreement. The management of this delivery is achieved by agreeing on the respective roles, rights and obligations of the parties. The agreement may specify not only functional properties for identification or creation of the service, but also non-functional properties of the service such as performance or</td>
</tr>
</tbody>
</table>
availability. Entities can dynamically establish and manage agreements via Web service interfaces.

<table>
<thead>
<tr>
<th><strong>Service Level Objective</strong></th>
<th>Service Level Objective represents the quality of service aspect of the agreement. Syntactically, it is an assertion over the agreement terms of the agreement as well as such qualities as date and time.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Service Provider</strong></td>
<td>An organization supplying services to one or more internal customers or external customers.</td>
</tr>
<tr>
<td><strong>SLA Manager</strong></td>
<td>A specialization of service provider: person/system that is responsible for managing SLATs and SLA relationships.</td>
</tr>
<tr>
<td><strong>Software Designer</strong></td>
<td>A specialization of software provider: person that designs/develops the architecture and components of a specific SLA based application.</td>
</tr>
<tr>
<td><strong>Software Manager</strong></td>
<td>A specialization of service provider: person that designs/develops the architecture and components of a specific SLA based application.</td>
</tr>
<tr>
<td><strong>Software Provider</strong></td>
<td>An organization producing software components which might be used by a service provider to assemble actual services.</td>
</tr>
<tr>
<td><strong>Software Service</strong></td>
<td>A software service is a specific IT service which is exposed/invoked by means of software entities such as Web services, user interfaces, or software-based business processes.</td>
</tr>
<tr>
<td><strong>Software Component</strong></td>
<td>Software components are the entities produced at design-time by a software provider.</td>
</tr>
<tr>
<td><strong>Service Type</strong></td>
<td>A service type (or abstract service) specifies the external interface of a service possibly including non-functional aspects. It does not specify any means (components, resources) which are needed for the actual provisioning of that service.</td>
</tr>
</tbody>
</table>
Appendix B: Abbreviations

AOP Aspect Oriented Programming
BM Business Manager
B-SLAM Business SLA Manager
EMF Eclipse Modelling Framework
ERP Enterprise Resource Planning
IE Interaction Event
FCR Finite capacity regions
Infr-SLAM Infrastructure SLA Manager
Infr-SM Infrastructure Service Manager
IoC Inversion of Control
KPI Key Performance Indicator
LLMS Low Level Monitoring System
LQN Layered Queueing Networks
MA Manageability Agent
MRE Monitoring Result Event
MVC Model View Controller
NFP Non-functional property
ORC Open Reference Case
OVF Open Virtualization Format
QoS Quality of Service
QPN Queueing Petri Nets
PAC Provisioning and Adjustment Component
POC Planning and Optimization Component
POJO Plain Old Java Objects
SaaS Software as a Service
SE Service Evaluation
SLA Service Level Agreement
SLAM SLA Manager
SLAT Service Level Agreement Template
SM Service Manager
SME Small and Medium-sized Enterprise
SOA Service Oriented Architecture
SW-SLAM Software SLA Manager
SW-SM Software Service Manager
TCO Total Cost of Ownership
TOGAF The Open Group Architecture Framework
Appendix C: Example Interaction Specification: Integrated Planning

In this appendix, we give an example for the detailed specification of an interaction. The example addresses the integrated planning cycle for software services but similarly applies also for other service interface types.

The integrated planning cycle that is related to the provisioning of software services is realized by three interactions: <<prepare_software_service>> (Section C.1), <<manage_software_service>> (Section C.2), and <<evaluate>> (Section C.3). Each specification provides additional information about the involved components, the detailed interaction flow between the components and the shared data types. The specifications of the other interactions are described in their corresponding deliverables.

C.1 Prepare Software Services

The relationship <<prepare_software_services>> allows SoftwareSLAMangers to search for service implementations provided by a SoftwareServiceManager and retrieve information about their dependencies. Furthermore, it provides a unified access point to the provisioning operation of software services including reservation and booking. The provisioning and management of a service is done via the <<manage_software_service>> relationship.

Involved Components

- SoftwareSLAManager
- SoftwareServiceManager

Overview

The relation <<prepare_software_services>> subsumes functionality that is necessary to find implementations for a particular service type as well as retrieving information about a service implementation or service instance.

Interfaces and Operations

<<prepare_software_services>> subsumes functionality that is necessary to find implementations for a particular service type as well as retrieving information about a service implementation or service instance. This relation is coprised of the following interface.

IPrepareSoftwareServices

- Get all available implementations of a particular service type. The implementation descriptors include information about the service
dependencies, monitorability, configuration options, and provisioning information such as lead times.

- Create a ServiceBuilder that allows the SLAManager to configure the service instance.
- If necessary, check for the availability of (limited) resources such as software licenses and make a time restricted reservation of these. Inform the caller about the success or failure of the reservation.
- If necessary, book/confirm a previously successful reservation and acquire all necessary resources for the specified time. Inform the caller about the success or failure of the booking.

**Overview of Interaction Sequence**

An overview on the interaction sequence is given in Figure 37.

![Sequence Diagram](image)

**Figure 37: «Prepare Software Services» sequence diagram**

These interactions are described as following:

- The SoftwarePOC queries for the available service implementations for a particular service type. SoftwareServiceManager returns an array of the service implementation. The SoftwarePOC selects the appropriate ServiceImplementation.
SoftwarePOC invokes capacityCheck to determine the availability of software resources required for the realization of the service provisioning process.

SoftwarePOC based on the results of capacityCheck requests SoftwareServiceManager to create ServiceBuilder instance for the specific service implementation. This service builder instance will then be used by the SoftwarePOC to configure the service instance.

SoftwarePOC reserves the required resources for materialization of the SLA. The reservation is time bound and times out automatically unless specifically confirmed by the SoftwarePOC.

Finally, the SoftwarePOC confirms the previously reserved resources to be booked for later usage during the service provisioning process.

**Detailed Specification**

**External Datatypes**

- **ServiceType**: Data structures describing service type (see Section 4.3.3)
- **ServiceImplementation**: data structure describing the specifics of service implementation to be used by SoftwarePOC and returned by SoftwareServiceManager (see Section 4.3.4)
- **ServiceBuilder**: data structure holding the configuration and (potential) bindings of a service to be instantiated.

**Local Datatypes**

**Exceptions**:

- **InvalidServiceTypeException**: Exception thrown when invalid service type is received by the SoftwareServiceManager
- **ReservationException**: Exception thrown during reservation request
- **BookingException**: Exception thrown during booking request

**Detailed Specification of Interface**

```
IPrepareSoftwareServices

+ queryServiceTypes(): ServiceType[]
+ queryServiceImplementations(type: ServiceType): ServiceImplementation[]
+ capacityCheck(impl: ServiceImplementation): int
+ createBuilder(impl: ServiceImplementation): ServiceBuilder
+ reserve(builder: ServiceBuilder, from: Date, until: Date): boolean
+ updateReservation(builder: ServiceBuilder, from: Date, until: Date): boolean
+ cancelReservation(builder: ServiceBuilder): boolean
+ book(builder: ServiceBuilder): boolean
+ cancelBooking(builder: ServiceBuilder): boolean
```

**Figure 38: Specification of IPrepareSoftwareServices.**
The interface IPrepareSoftwareServices exposes operations which are used by SoftwarePOC to prepare for the software service provisioning. The interface consists of the following operations:

- **queryServiceTypes** returns the list of service types supported by the ServiceManager. The service types are mapped to SLA(T)s by the SLA Manager.

- **queryServiceImplementation** is used to query for the service implementations available for a given service type. It throws an exception if an invalid service type is provided.

- **capacityCheck** enables the SoftwarePOC to perform a capacity check on the resource availability for example the software licenses available to support the software service provisioning.

- **createBuilder** creates and returns an instance of ServiceBuilder which can be used by the SoftwarePOC to configure the service instances.

- **reserve** is used to reserve the required resources to be used in the future during the software service provisioning process.

- **updateReservation** is used to update previously reserved resources.

- **cancelReservation** is used to cancel the previously reserved resources.

- **book** is used by the SoftwarePOC to confirm previously reserved resources.

- **cancelBooking** is used to cancelling the bookings of resources for certain service implementation.

### C.2 Manage Software Services

The relationship <<manage_software_services>> allows SoftwareSLAMangers to create new service instances and carry out service lifecycle management related operations. This relation complements the functionality enabled through the relation <<prepare_software_service>>.

#### Involved Components:

- SoftwareSLAManager
- SoftwareServiceManager

#### Overview

The relationship <<manage_software_services>> enabled SoftwareSLAManager to interact with SoftwareServiceManager to create new software service instances. Additionally, SoftwareSLAManager can execute service lifecycle management related operations through this relationship.

#### Overview of Interfaces and Operations

The relationship <<manage_software_services>> enabled SoftwareSLAManager to interact with SoftwareServiceManager to create new software service instances. Additionally, SoftwareSLAManager can execute service lifecycle management related operations through this relationship.
**IManageSoftwareServices**

The interface IManagerSoftwareServices subsumes all functionality related to the provisioning and management of a software service.

- trigger the provisioning process that executes the necessary actions to provide access to a booked service instance
- release and cleanup of a provisioned service instance. Disables access to a provisioned service instance
- change the implementation or dependencies of an instantiated service
- (re-)configuration of the service
- (re-)configuration of the monitoring system
- query possible change actions
Overview of Interaction Sequence

Figure 39: Managed_Software_Services interaction sequence

The summarized description of the interaction sequence is given below:

1. After SLAs have been established with the customer, SoftwareSLAManager invokes the startServiceInstance operation to trigger the software service provisioning. The operation invocation can take one of two forms: 1) either provisioning should be triggered immediately or 2) provisioning is scheduled to be triggered at a later time. In this case a start time is provided as an argument to startServiceInstance operation.

2. Once the service is provisioned and operational, SoftwareSLAManager will invoke queryServiceStatus to find out the status of the service instance.

3. If the system is to be monitored the operation getSensorSubscriptionData provides information what information can be monitored (i.e. which sensors are available) and where monitoring events are published.

4. If the monitoring system is to be adjusted, this can be done via the method configureMonitoring.
5. In case of some potential SLA violation, some management operations need to be performed on the service instance. Depending on the required operation to bring service instance in compliance to SLAs, SoftwareSLAManager invoked the executeAction operation which will perform the operation provided as an argument.

6. At the end of the service lifecycle, SoftwareSLAManager will use the stopServiceInstance to trigger the unprovisioning of the software services. Again, this operation can take two forms and the appropriate operation is invoked by the SoftwareSLAManager.

**Detailed Specification**

**External Datatypes**
- ServiceBuilder (see Section 4.3.5)

**Local Datatypes**

**Exceptions:**
- **ServiceStartupException**: This exception will be thrown if some problem occurs during the service configuration and startup
- **ManagementException**: This exception will be thrown if some problem occurs during the execution of a management operation on the service instance

**Detailed Specification of Interfaces**

<table>
<thead>
<tr>
<th>Interface</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>IManageSoftwareServices</td>
<td>startServiceInstance, stopServiceInstance, getSubscription, configureMonitoring, deconfigureMonitoring, executeAction</td>
</tr>
</tbody>
</table>

**Figure 40: IManageSoftwareServices Specification**

The interface IManageSoftwareServices subsumes all functionality related to the provisioning and management of a software service. The interface consists of the following operations:

- **startServiceInstance** triggers the provisioning of a new service instance either immediately or at a given point in time. The instance will be associated with a unique Id. The builder specifies all bindings to external services, configurations, and monitoring options. The notificationChannel points to the message bus and topic where status related events of the framework are to be posted.

- **stopServiceInstance** is used for triggering decommissioning of the service instance. The decommissioning can either take place immediately or is initiated at the time specified.
- **queryServiceStatus** allows the SoftwareSLAManager to query for the status of a particular service instance.
- **getSensorSubscriptionData** returns a list of available sensors and where to find their data.
- **configureMonitoringSystem** applies the given configuration to the monitoring system of the managed ServiceInstance. The given configuration replaces the previous one (if any).
- **deconfigureMonitoring** disables the current configuration.
- **executeAction** executes a domain-specific action for the adjustment of a ServiceInstance.

### C.3 Evaluate

The `<<evaluate>>` interaction allows the **PlanningAndOptimization** (POC) sub component of Generic SLA Manager to evaluate possible realizations of a service regarding their expected quality. The evaluation is done by the **ServiceEvaluation** (SE) component. The general concepts of service evaluation are explained at the ServiceEvaluation page. On this page, the details of the interaction, and the involved data models are presented.

#### Overview of Interfaces and Operations

The `<<evaluate>>` interaction specifies an interface `IServiceEvaluation`, which provides an operation for evaluation of the quality of a target software service. The target service may depend on a list of other services. The quality of the required services may influence the quality of the target service. A set of possible realizations of the target service, including concrete quality descriptions for required services, is given as an input to evaluation.

**Involved Components:**

- PlanningAndOptimization (POC) sub component of the SLAManager (for business, software, infrastructure)
- Service Evaluation
Overview of Interaction Sequence

Figure 41: High-level Interaction of POC (via SLAM) and ServiceEvaluator

The interaction consists of one call evaluate() that is issued by the POC and triggers SE to determine quality estimates for a set of ServiceBuilders. SE returns these realizations in a list, together with the associated quality estimates, back to POC. POC can use these results for its planning functionality (which is only sketched here) and possibly reinvoke the evaluate() method to get quality estimates for further ServiceBuilders.

Detailed Specification

External Datatypes

Following the paradigm that everything related to SLAs can be expressed by SLAs and SLATemplates, the only data types passed in are a set of ServiceBuilders and an SLATemplate. The return value is a list of EvaluationResults.

- **SLATemplate** (from [19]): Contains information about the target service and the terms to be agreed upon. More concretely, it includes:
  - The interfaces / operations of the target service type
  - Agreement terms with ConstraintExpressions specifying the required quality and the envisioned usage of the target service type

- **ServiceBuilder** (from Section 4.3.5): Represents the configuration and used dependent services of a particular ServiceImplementation. It references all information needed in order to evaluate the service quality. In particular, these are
  - The ServiceType to be evaluated (and via the type also the list of service interfaces / operations)
The ServiceImplementation containing a list of required (dependent) service types

- A list of ServiceBindings mapping required service types to SLA or SLATemplates describing the offered service quality and the implied service usage bounds.

**Local Datatypes**

- **EvaluationResult**: Bundles results of ServiceEvaluation as a return value to the calling POC (see Figure 42). Consists of:
  - An SLATemplateEvaluation for the target service type, referring to the SLATemplate given as an input for the target service
  - A list of SLATemplateEvaluations for the dependencies, referring to the SLATemplates given as an input for the dependencies

- **SLATemplateEvaluation**: Describes the evaluation results of a given SLATemplate. In case of software services, it evaluates the ConstraintExpressions of Guaranteed.States within AgreementTerms of the SLATemplate. It consists of:
  - A reference to the evaluated SLATemplate
  - A list of ConstraintEvaluations for the ConstraintExpressions within the template

- **ConstraintEvaluation**: Describes the evaluation of a certain ConstraintExpression within an SLATemplate. Contains:
  - A reference to the evaluated ConstraintExpression
  - A double value indicating the evaluation result

Table 1 lists examples of ConstraintExpressions and ConstraintEvaluations for them.

<table>
<thead>
<tr>
<th>ConstraintExpression</th>
<th>Constraint Explanation Evaluation</th>
</tr>
</thead>
</table>

---

Figure 42: Evaluation Result Class Diagram
For the operation $S$ of the target service, the customer requires a mean completion time below 500 ms. The ConstraintEvaluation contains a value of 350 indicating that the predicted mean completion time (for a certain service realization) is 350 ms. Thus, the given requirement is expected to be satisfied (for the given realization).

For the operation $S$ of a required software service, an upper bound for usage is given through a mean invocation rate of 200 requests per minute. ServiceEvaluation predicts a mean of 150 rpm’s for a given service realization (and target service usage profile). Thus, this realization is valid and can be included into the list of evaluation results for POC.

### Table 1: Examples for ConstraintExpressions and ConstraintEvaluations

<table>
<thead>
<tr>
<th>ConstraintExpression</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{mean}(\text{completion_time}(S))$ &lt; 500.0 ms</td>
<td>350.0</td>
<td>For the operation $S$ of the target service, the customer requires a mean completion time below 500 ms. The ConstraintEvaluation contains a value of 350 indicating that the predicted mean completion time (for a certain service realization) is 350 ms. Thus, the given requirement is expected to be satisfied (for the given realization).</td>
</tr>
<tr>
<td>$\text{mean}(\text{arrival_rate}(S))$ &lt; 200.0 rpm</td>
<td>150.0</td>
<td>For the operation $S$ of a required software service, an upper bound for usage is given through a mean invocation rate of 200 requests per minute. ServiceEvaluation predicts a mean of 150 rpm’s for a given service realization (and target service usage profile). Thus, this realization is valid and can be included into the list of evaluation results for POC.</td>
</tr>
</tbody>
</table>

### Exceptions:
- **IllegalArgumentException**: Indicates that the data given as an input to evaluation are incomplete or inconsistent. For example, service quality descriptions and matching of service dependencies can be incomplete, or constraint expressions of agreement terms might be unpredictable.
- **EvaluationException**: Indicates that an unexpected failure occurred during evaluation. For example, the QoS model might not be available; the prediction server might be not reachable etc.

### Detailed Specification of Interfaces

Figure 4 depicts the single interface `IServiceEvaluation` that is provided by the ServiceEvaluation component.

```xml
<interface>
   IServiceEvaluation
+ evaluate(realizations : Set<ServiceBuilder>, template : SLATemplate) : Set<EvaluationResult> throws IllegalArgumentException, EvaluationException
</interface>
```

**Figure 43: IServiceEvaluation Interface**

The IServiceEvaluation interface contains a single operation:

- **evaluate**(realizations : Set<ServiceBuilder>, template : SLATemplate) : Set<EvaluationResult> throws IllegalArgumentException, EvaluationException

Inputs to the operation are as follows:

- **Realizations**: a set of ServiceBuilders. Each ServiceBuilder must refer to the same ServiceType (via the ServiceImplementation) and match all
existing ServiceDependencies to existing SLAs or SLA templates describing the quality (and possibly usage bounds) that is expected.

- **Template**: an SLATemplate describing (i) a service type through its interfaces and operations, (ii) the required quality of service through Agreement terms and related ConstraintExpressions, and (iii) the envisioned usage of service operations (arrival rates / data volumes).

The output of the evaluation is a set of EvaluationResults. Thereby, each EvaluationResult refers to one ServiceBuilder and describes the expected quality of service for this particular realization, as well as the predicted usage of required (software) services. The returned list of EvaluationResults may be smaller than the list of ServiceBuilders given as an input to the evaluate() operation. This is due to the fact that a certain ServiceBuilder might turn out to be invalid (e.g., because a given upper usage bound on a required service is violated) and thus is excluded from the list of results.

Two kinds of exceptions - `IllegalArgumentException` and `EvaluationException` - may be thrown as described in the previous section and prevent the evaluate() operation from successful completion. This possibility has to be taken into account by POC. A text message comes with exception instances to describe the cause of the failure in detail.