Engineering Virtual Domain-Specific Service Platforms

Specific Targeted Research Project: FP7-ICT-2009-5 / 257483

Decision Support Framework for Platforms as a Service (Interim)

Abstract

INDENICA aims at more effective engineering for the integration and domain-specific customization of service platforms. Often, there is no unique answer to the many engineering challenges that exist in this context. Thus, correctly identifying the many possible engineering alternatives throughout and proposing systematic ways of deciding upon them is of central importance to effective engineering.

This deliverable describes the INDENICA decision support framework. It addresses and integrates decision making activities throughout the entire INDENICA platform development lifecycle with a main focus on the requirements engineering and architecture steps. This first version of the deliverable provides an overview of the many issues that arise in this context and the different possibilities of addressing them. Deliverable D1.3.2 will focus on certain parts of the methodology and will provide a prototype solution to support decision making effectively.

Document ID:	INDENICA – D1.3.1
Deliverable Number:	D1.3.1
Work Package:	WP1
Type:	Deliverable
Dissemination Level:	PU
Status:	final
Version:	1.0
Date:	2012-03-31
Author(s):	SUH, SAP, SIE, PDM, TEL, UV
Project Start Date:	October 1st 2010, Duration: 36months

Version

0.1	14. February 2012	Document Structure for integration defined
0.2	9. March 2012	Detailed document template
0.3	16. March 2012	Added the content of Section 6
0.4	26. March 2012	Revised Section 6.1, 6.2
0.5	27. March 2012	Revised Section 6.3
0.6	27. March 2012	Updated the introduction of Section 6.1 and Section 6.3 with the relationship to VbMF in WP3 and added the related publications.
0.8	31. March 2012	Updated Architecture part, parts of the requirements part
0.9	12. April 2012	Updated requirements part, mostly finished.
1.0	20. April 2012	final version

Document Properties

The spell checking language for this document is set to UK English.

Table of Contents

Table of Figures

1 Introduction

Developing virtual domain-specific service platforms is a rather complex engineering task. As a consequence many different decisions need to be adequately made in order to prepare an adequate infrastructure in order to support this task. The same holds true for deriving an individual instance, i.e., a specific virtual service platform, which is adapted to the specific concerns of a situation, from the created infrastructure. Again, it is a non-trivial task to identify the right decisions to make and to ensure the resulting decisions are correctly taken. As a reaction to recognizing this challenge, the INDENICA-approach provides also decision making support to the engineer on a conceptual, methodological, and tool-supported level. Of course, depending on the type of decisions and the influences on decision making not all decisions can be fully tool-supported or even automated. This deliverable will discuss the current status of the contributions to INDENICA-decision-making. In a second deliverable, we will build on these results and describe tool prototypes that support decision-making within the INDENICA methodology.

INDENICA decision-making support focuses on two main aspects: requirements-level support and architecture-level support. These are covered in the following two Sections 2 and 3, respectively.

Further relationships to other INDENICA deliverables are:

- D.1.1: Report on state of the art discusses a number of problems and issues, which are relevant to this deliverable as well.
- D1.2.1: The requirements engineering framework is of fundamental importance to Section 2.
- D2.1: Open Variability Modelling Approach for Service Ecosystems provides the basis for modelling of variability. This is taken up again in Section 2.3.
- D3.1: View-based Design Time and Runtime Architecture for Tailoring Virtual Service Platforms is related to the work on architecture decision making, as discussed in Section 3.

Comments on the relation to previous work:

- The contributions described in Sections 2 and 3 were developed as part of the INDENICA project and were motivated by the project. There exist relations (as described above) to previous deliverables, however.
- Further, especially the contributions in Section 3 are currently submitted for publication or already published.

2 Requirements Decision Making

The first step in the decision-making process is to determine what to build, both from the perspective of the possible range of virtual service platforms, as well as for a single, specific virtual service platform. This is the realm of requirements decision-making. In this section, we will discuss how the requirements engineering approach IRENE is used to identify the potential decision space that is available during requirements engineering. In practice, usually not all requirements that are desirable will also be realized. As a consequence, we require a requirements prioritization approach as a first filtering step after identifying the decision space. This is discussed in Section 2.2. INDENICA aims not only at supporting a specific infrastructure, but to provide the basis for significant, domain-specific customization. This is realized by identifying variability and resolving this variability while deriving a specific platform. This is described in Section 2.3.

2.1 Requirements-based identification of possible decisions

This section addresses requirements engineering as a fundamental part of the INDENICA (decision making) methodology. Requirements engineering in INDENICA is based on IRENE, a goal-oriented approach, and is supported by the IRET tool. We think a platform could be specified as if it were a "conventional" single solution, whose aim is to provide services to others, which in turn may create different applications.

IRENE (Indenica Requirements ElicitatioN mEthod) borrows from KAOS [Vla09] to provide the user with a "complete" solution to elicit the "usual" functional and non-functional requirements, but also to state the foreseen adaptation capabilities and the variability that should be embedded in the system-to-be:

- Requirements can be classified as either *crisp*, that is, they are either satisfied or not satisfied, and *fuzzy*, to embed flexibility in the system and be able to reason in terms of different degrees of satisfaction;
- Adaptation is specified in terms of *adaptation goals*, that is, foreseen ways to let the platform-to-be to adapt to some external events. These goals can also be quite vague and identify general-purpose exception-handling solutions;
- Variability (from a requirements' perspective) is stated by means of dedicated forms associated with the different modelling elements. The idea is to let the user identify the type of variability, the possible values, and also some hypothetical constraints.

IRENE offers a set of graphical symbols, to let the user state the structure of the requirements, and textual annotations, to refine and better specify the concepts. Annotations can be added in the form of natural language, to ease the user in

his/her work, but also to allow for an incremental specification of requirements, but they can also be stated using formal notations.

A prototype tool called IRET (IREne Toolset) supports all these aspects. IRET is implemented as an Eclipse plugin and fully supports IRENE to allow users to easily define complete and coherent requirements models.

The interplay between requirements and decisions in INDENICA is summarized in Figure 1. Besides the general assumption that decisions (on the actual implementation of the platform) are made based on the requirements identified so far, IRENE provide some peculiar elements that can be properly exploited to guide (help) the decision process. More specifically, requirements can work for INDENICA decisions in different ways:

- Stated requirements identify the "solution space", that is, they specify the part of the universe of interest for the particular platform.
- They provide the basis for making any decision since requirements define what the platform is supposed to offer.
- They provide the constraints for decisions (i.e., as long as decisions do not impact constraints, then decisions are fine).
- Adaptation goals help decide the adaptation capabilities that must be embedded in the platform and how they should be provided.

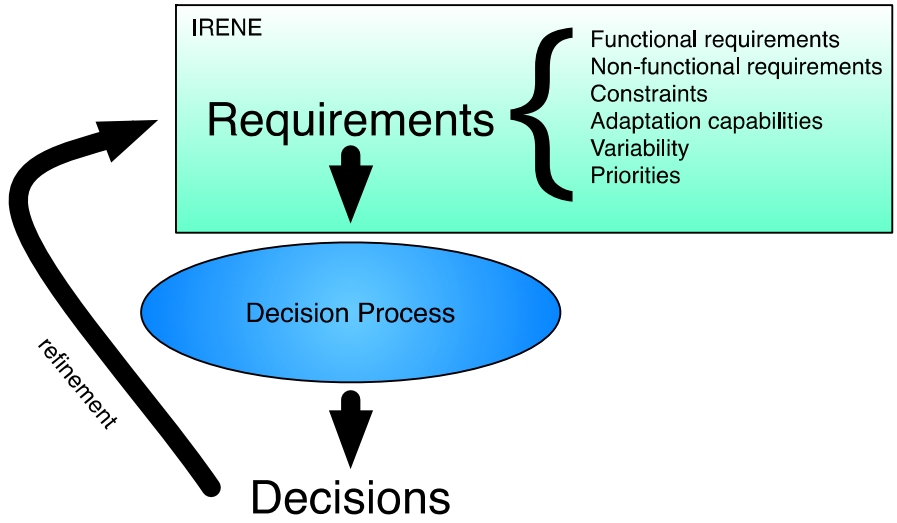


Figure 1: Interplay between Requirements and Decisions in INDENICA.

- Decisions can help prioritize requirements. IRENE already supports priorities associated with requirements; the decision process helps decide among the views of the different stakeholders.
  They provide explicitly a set of (variability) decisions, which will actually impact the externally visible behaviour of the system. This is a basis for the variability in the product line.

Besides the usual top-down flow, that is, requirements that feed the decision process, we must also flip the perspective, and adopt a bottom-up view: decisions can be used to refine/complement the requirements, and thus iteratively further refine taken decisions, if needed.

The main task of IRENE is to develop and describe the decision-making space to begin with. One can summarize that there exist three levels on which decision-making is important from a requirements engineering perspective:

- *Multi-view inconsistencies:* different views (either due to the fact that different representations are used or different stakeholders have different interpretations) may be inconsistent with each other.
- *Prioritization:* even, if we have a consistent picture of desirable requirements that should be realized, for each specific virtual service platform, we are still bound to a maximum amount of resources that are available. As a consequence, a prioritization needs to occur.
- *Variability:* some aspects may simply be relevant only in specific contexts or for specific application scenarios. These need to be conserved and enabled as explicit variability. This enables to postpone decision-making from platform infrastructure development time to platform-reuse time (or even platform use time, where this overlaps with adaptivity).

IRENE supports all these levels, albeit in different ways:

1. IRENE provides graphical means to integrate different views into a single, coherent representation of the requirements, and thus into a unique starting point for the decision process. Usually, requirements specifications comprise different views, and IRENE provides automatic solutions to merge the views and obtain a single specification where all “similar” concepts are merged, and all the differences are highlighted properly.
2. Inconsistency that derives from differences in stakeholder perception can of course not be avoided this way. Here, IRENE aims at capturing the different perspectives and at supporting the integration by providing a basis for discussion. Each stakeholder is free to add his/her view, but then the integration process merges what is equal in the different views and highlights the differences among the other elements. IRET does not merge any inconsistency automatically, but the user is in charge of solving them.
3. The detailed goal-oriented description that IRENE supports provides a good basis for prioritization. The details of the prioritization approach envisioned for INDENICA is described in Section 2.2. Each goal is associated with a foreseen priority, and again this is the starting point for the prioritization process. Note that different stakeholders may assign different priorities to the same goals, and thus IRET will help the users identify a single priority associated with each goal (as described in Section 2.2).

4. IRENE explicitly captures base information for variability decisions and constraints among those decisions. Each element of IRENE can be associated with variability-related comments. Variability here means: what can vary, it's possible values, when this variability should be resolved (design-, deployment-, and run-time), constraints with respect to other variability requirements, and free notes for the developers. The purpose of this information is twofold: it (i) feeds the actual design of the variability that belongs to the platform-to-be and (ii) provides the basis to initiate the decision process.

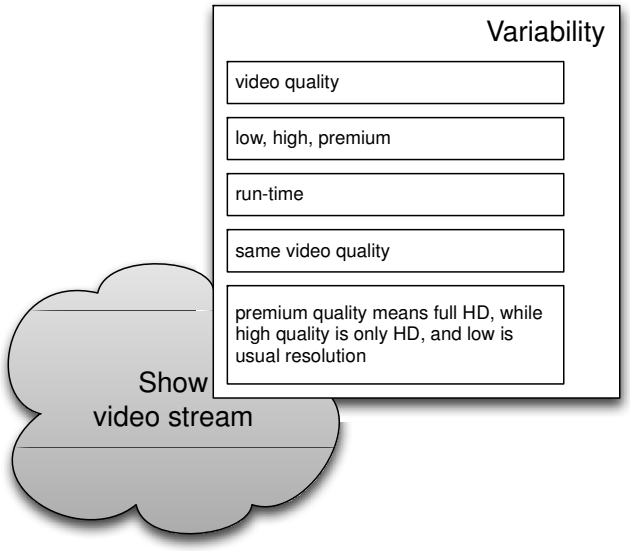


Figure 2: Example variability hypotheses in IRENE.

For example, Figure 2 shows a simple case of how variability can be described in IRENE. If we suppose a simple functional goal of a video-audio conferencing platform, one may say that the quality of the video stream can assume three different values: low, high, and premium. This variability must be dealt with at runtime. The notes specify the meaning of the three values, and the constraint states that the video quality must be paired with “similar” audio quality.

Thus, while IRENE itself is not part of the actual decision making, it provides the basis for the further decision-making oriented procedures within INDENICA.

2.2 Requirements Prioritization as Basis of Decision Making

The prioritisation of requirements is a discipline of Requirements Engineering that brings the business view into the early phases of a development process. But it must not be seen as a distinct phase, rather priorities evolve during the RE process and changes to a set of requirements also change priorities.

The interaction between decisions and priorities is manifold:

a) stakeholders decide on a priority assigned to a requirement

b) priorities of different stakeholders are in conflict, they are negotiated and a decision on final priorities is taken

c) prioritized requirements are a major input for planning decisions (e.g. release plan, iteration plan)

d) prioritized requirements are a major input for architectural and design decision

There is virtually no development of a product, a system or a solution where the limiting conditions (time, cost, resources) allow fulfilling all requirements and often the result cannot be achieved in "one shot" but require a set of releases. This also applies for service platforms.

2.2.1 Prioritizing requirements

Assigning priorities to requirements is always part of a decision process. At first stage priorities are often allocated to requirements by "gut feeling" but this can lead to inadequate priority distribution ("everything has priority 1"). There are a number of prioritization techniques like

- Ranking,
- Kano classification,
- On-Criterion classification
- Top-ten selection etc.

An overview on these techniques is given in [Po09], page 632.

In practice the number of items to be prioritized should not exceed 100. For propagating priorities to higher and lower levels of detail there are strategies in place; a widely applied method for this is the Analytic Hierarchy Process (AHP, see [SaVa01]). It allows finding strategies for aggregating priorities on higher levels and inheriting priorities to lower levels.

2.2.2 Resolving conflicts in stakeholder evaluations

It is obvious that different stakeholders or stakeholder groups have different priorities for the same requirement. Here a negotiation is necessary that leads to decision on final priorities that are used in the further architecture and design decisions as an essential input. At the same time this supports the identification of the overall relevant decision space.

Below we will outline how the INDENICA Requirement Engineering Framework supports this negotiation [D1.2.1]. It consists of two methods that are part of RE for service platforms:

- IRENE, the INDENICA Requirements ElicitatioN mEthod, which supports the definition/elicitation of the requirements of innovative service platforms,
- User Centred Requirements Engineering, which supports the negotiation of different priority views on a set of given requirements.

IRENE provides a hierarchical model of goals that easily allows selecting the appropriate level of granularity for prioritization.

User Centred Requirements Engineering was elaborated in detail in [D1.2.1]; here we provide a short summary:

Requirements can be characterized on three levels:

a) the user level
b) the requirements level
c) the importance level

The user level tries to give a structure to various groups of stakeholders. This is of importance especially in the context of (virtual) service platforms. Two examples:

- In integration projects, users come from different levels of the automation pyramid: Enterprise resource planning, manufacturing execution, supervisory control, process control, etc. On all these levels the users have different views on the integrated system, different goals, different knowledge and thus different priorities on the requirements.
- In the IDENICA case studies [D5.1] there are two major user groups: the application developers and the end users, who again may have different priorities on the Virtual Service Platform depending on their focus on the warehouse, the yard management, the monitoring, or the overall integration.

The requirements level brings in a (as far as possible) complete and structured set of requirements. A good structure is necessary for selecting the appropriate level of abstraction and granularity of the requirements. Here, the goal-based hierarchical approach of IRENE provides a suitable input. It further shows requirements dependencies and conflicts.

The importance level shows the priorities that user groups allocate to individual requirements or requirements groups on the selected level of granularity.

The approach does not contain or predetermine a prioritization technique like ranking, Kano classification, top ten selection etc.

For applying the User Centred RE method it is necessary to use the same technique with all user groups. In the following examples we subsume a prioritization model with four values from 1 to 4, which mean “unimportant”, “fairly important”, “important” and “very important”.

Within a user group the requirement and priorities span a two-dimensional vector space that allows conclusions on the quality of voting:

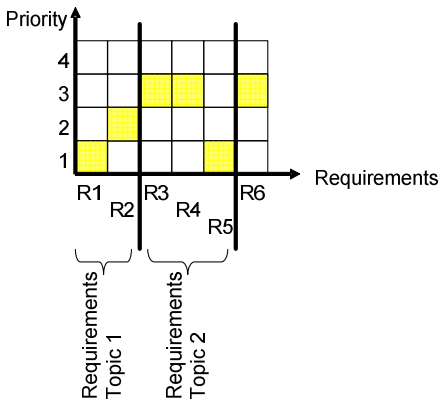


Figure 1 Requirements-Priority table

The rows show for each priority level the weighted number of votes. Here a good distribution is an indicator for a clear decision basis. There are then requirements which clearly are identified as important or very important and thus candidates for implementation or for early releases.

If there is an accumulation on one priority level with a big number of requirements ranked high, and only few ranked low could indicate a weakness in the prioritization technique and its application. In this case the change of the priority model should be considered.

For comparing the different sets of priorities from the user groups a graphical representation was presented in [D1.2.1]:

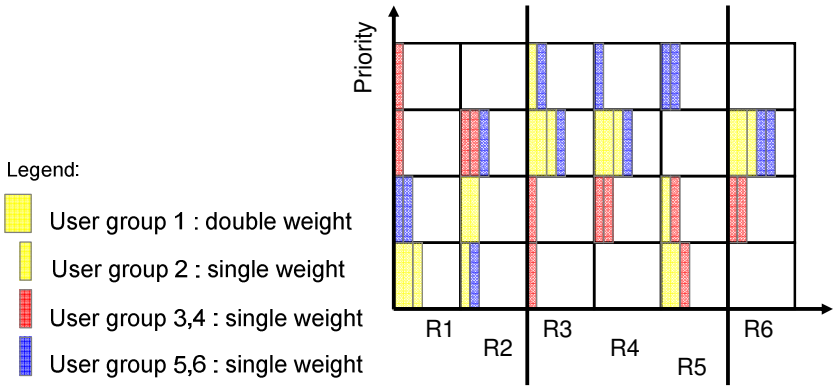


Figure 2: Priority mosaic

User groups can also have dedicated weights for their voting. The example in Figure 2 shows a consolidated voting of six user groups where group 1 has a higher (double) weight than the others.

The columns show for each requirement the voting over all user groups. This gives valuable input for the final priority: a homogenous voting gives a clear final priority. An inhomogeneous voting triggers the negotiation with user groups. And for final decisions on requirements to be implemented a further analysis of implementation cost, time and resources will be necessary.

2.2.3 Release and iteration planning based on priorities

A set of requirements with finalized priorities is a valuable and crucial input for planning processes.

On a product portfolio level a product roadmap or product evolution plan can be derived. A systematic approach for this was presented by Ullah, Ruhe and Garousi with the COPE+ method [UlRuGa10].

Also service platforms evolve and affect a Virtual Service Platform with multiple and independent life cycles. The need for coordination of platform lifecycles and the respective planning process was elaborated in the Deliverable on Governance of Virtual Service Platforms [D3.2].

2.2.4 Priorities as input for variability decisions

The analysis of priority distributions allows more conclusions that lead directly to decisions in the context of product line engineering:

- A homogeneous allocation of high priority indicates that the requirement is important for many user groups; it is a *candidate* for a commonality.
- An inhomogeneous distribution of priorities indicates that the requirement is specific for one or only a few user groups; consequently it is allocated to a specific product or solution that is based on a platform.

As a final result of the prioritization all requirements have a priority that is agreed upon by the user groups, but is still not free of conflicts. In the best case there is a bunch of requirements clearly identified as the most important ones. In case an iterative approach is applied, they shall be treated in the following analysis in the first iterations. At the same time it is clear that there are also other decision criteria for allocating requirements to a platform or to the application level.

At this point the analysis goes into the solution space by evaluating them against architectural options and constraints. Requirements priorities ensure that in these following decisions on architectural and design options the “Voice of the Customer” is respected and the resulting system or platform will satisfy the expectations.

2.2.5 Prioritization Conflicts

Defining the VSP capabilities means at first stage selecting from a set of requirements where the priority is a main decision criterion. The other set of decision criteria comes from the definition of architectural constraints. Bringing these two dimensions together leads to the core of the decision for VSP; here extreme situations may occur:

- A highly prioritized requirement cannot be implemented at all on the architecture level. In this case the solution space has to be enlarged and options beyond the VSP must be evaluated:
    - Extension of one of the base platforms

  - Integration of another platform that allows fulfilling the requirement
  - Dropping the VSP approach and evaluating the development of an application on pure base platforms.

- The VSP approach conflicts mostly with low priority requirements. In this case the affected requirements are not selected for further analysis and implementation.
- None of the constraints conflicts with any of the requirements. In this case all requirements can be fulfilled unless they are de-selected by other criteria (e.g. cost, time, resources…)

In practice there will always be a continuum between these extreme options and this makes it more necessary to invest effort in a thorough prioritization of requirements.

2.3 Variability Decision Making

Variability management is typically considered to consist of two main parts: domain engineering and application engineering [vLRS07]. While domain engineering aims at establishing a precise model of the decisions to be made, application engineering focuses on the use of these decisions for deciding on the particular properties of an individual solution (in our case, an instantiated, virtual and domain-specific service platform). We will first address the domain engineering phase in INDENICA, then the application engineering phase.

2.3.1 Domain Engineering in INDENICA

Typically, the first step in domain engineering is to identify the potential range of variation that requires support. This is typically called *scoping*. Scoping is typically done by developing a set of potential products and identify their commonalities and differences [JE09] and part of product management for product lines [HSH06]. In INDENICA, the approach differs and an analysis of the relevant requirements, including their scope is done by IRENE, the requirements elicitation and analysis methodology. The result of IRENE is a set of requirements with identified potential variabilities. These variability decisions are only identified on an informal level. The next step is the prioritization (cf. Section 2.2), which identifies requirements of lower importance, which may be removed from the requirements. It is the focus of INDENICA decision modelling to refine and formalize the identified decisions using the approach described in [D2.1]. Thus, the derivation of decisions amounts to a three-step approach as shown in Figure 3, left side.

The result of domain engineering in INDENICA is actually not a single model, but rather a refined, two-layered model. This is shown in Figure 3, right side. On the upper level it contains the overall (requirements level) variability, as specified in [D2.1]; on the lower level it contains the various requirements, along with descriptions defining which requirements relate to which decision.

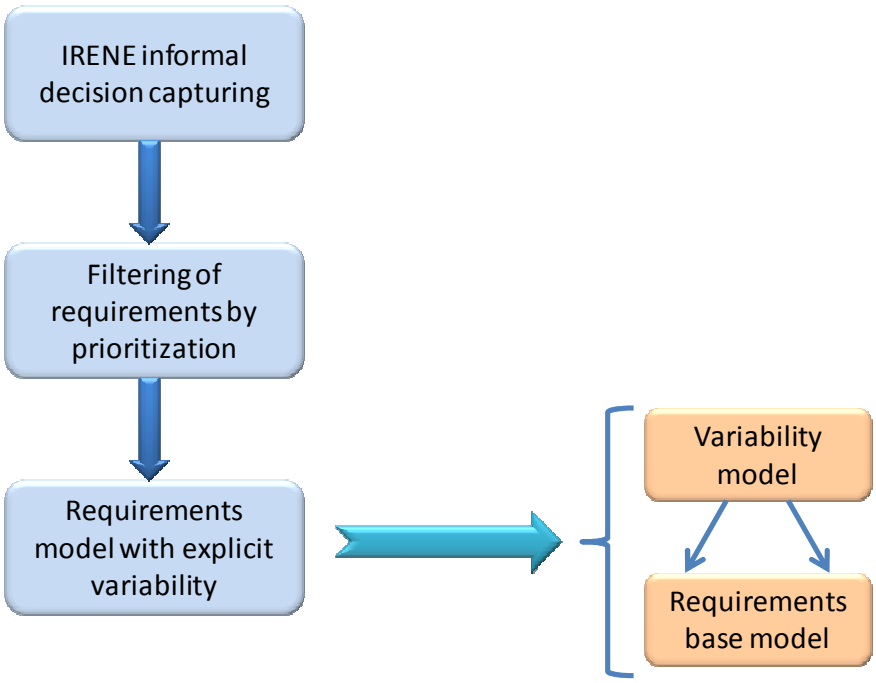


Figure 3: Development of an explicit variability model in INDENICA.

In order to derive the full variability model, we need to refine the information contained in the IRENE model. In particular, we need to identify the following aspects, which are not (a formal) part of the IRENE model:

- *Value range* - what are the different (perhaps alternative) resolutions of a certain variability.
- *Multiplicities* - can we select multiple values or can only a single value (true alternative) be picked.
- *Dependencies* - what constraints and dependencies exist among individual variabilities?
- *Hierarchy* - often it is useful to identify a hierarchical ordering among individual variabilities.
- *Binding Times* – at which time in the lifecycle must/can a decision be made.

An important observation when translating from IRENE to a variability model is that goal-models like IRENE are basically forming and-or-trees. Thus, they already provide a logic structuring of the requirements, including hierarchical dependencies among the individual requirements.

In order to identify a relevant variability decision based on IRENE, we will usually assume that this is already identified in an informal way in IRENE. For example, a comment (or a requirement) may define that it must be possible to have support for emergency shipping. We conclude that this is a feature that is not necessary for all instances, thus, we identify it as a variability decision (example 1). In another case there might be different requirements for the types of trucks that need to be

supported (e.g., cooling, different sizes, etc.) and we conclude that this is not a difference among stakeholders, but rather that this is a difference due to different installations (example 2).

In both cases, in order to be sure that these are indeed relevant differentiations that should amount to variabilities, we can use the information from prioritization, but still need to make additional checks: if there is a significant difference among stakeholders regarding the priority of such a requirement (e.g., cooling in example 2), then there might be two reasons: the stakeholders simply have different views, although there is only a common variant. In this case, requirements prioritization needs to combine with requirements negotiation and common ground regarding what is more appropriate needs to be found. However, if the stakeholders vary in their judgement because they are actually judging different cases, e.g., an installation in the US for a small company vs. a customer-specialized solution for a large European company, then this might actually give rise to a variability. This difference can only be identified through additional stakeholder information.

Whenever we judge that we have to deal with a true variability (this may also be directly identified as a goal), then we come up with a name for the corresponding decision.

The *value range* is defined accordingly by inspecting the various alternatives identified in the IRENE-model. In the first example this is very simple as it amounts to a Boolean decision. The second is more complex: we might identify an enumeration for the different transportation types like cooling, road train size, large size, etc. However, in practice further analysis of such an enumeration often reveals that several concerns are intermingled. In the given example, actually there is a form of goods (cooling, normal, etc.) and sizes intermingled. It might then be more appropriate to derive these basic concerns explicitly as different decisions.[1]

Multiplicities need to be identified as well from a combination of model analysis and explicit information elicitation as the informal model information will typically not be sufficient. There are, however, two main cases, which will usually be easy to identify from the IRENE models: a) the different alternatives cannot be combined *or* b) they can be arbitrarily combined. In the first case, we can directly model them as an enumeration, while in the second case, the INDENICA Variability Modelling Language (IVML) allows to model this as a set or list (if ordering is important).

Besides the basic decisions also *dependencies* among the decisions must be identified. To some extend this can be deduced directly from the IRENE models. As these models provide on the one hand formal relations among individual requirements (and-or), these can be translated into constraints among different decisions (respectively values of decisions). As further information also informal

[1] It should be noted that the resulting decisions will typically be interdependent (e.g., cooling might only be available in combination with certain truck sizes). As these decisions are not necessarily identified during the initial requirements modelling, we will also not have requirements model information on this. Thus, the necessary constraint information needs to be identified from scratch in this stage.

constraints can be described in IRENE (cf. Section 2.1). Often this will be sufficient as a starting point and can be translated into formalized constraints among decisions. Even though these constraints may be informal, IVML provides a very rich constraint language, which should be powerful enough to easily translate these constraints. In cases, where the input model is not sufficiently detailed, further elicitation activity is necessary to complete the constraint model.

Finally, it is sometimes useful to structure the various decisions in a *hierarchical* manner. This hierarchical structure can easily be deduced from the structure of the IRENE models on one hand (if the basic decision is described already in the IRENE model) or through domain information on the other hand. The idea is to have a more specific decision as a part of a more general decision. In IVML this can be easily described using the compound construct. A particular case of such a hierarchical model is the possibility to deal with multiple abstraction levels.

2.3.2 Model Composition in INDENICA

When considering the INDENICA variability decisions we should keep in mind that INDENICA addresses the domain-specific customization of (base) platforms as well as the integration of the individual platforms into a virtual platform. When we develop a new virtual platform according to the concepts outlined above, we will usually rely on existing (customizable) base platforms. In such a situation it is important to not only derive a variability model for the integrated platform, but also to describe the mapping of the variability to the base platform variabilities. This is conceptually shown in Figure 4. The IVML supports such mapping explicitly by its variability composition support and its interface concepts.

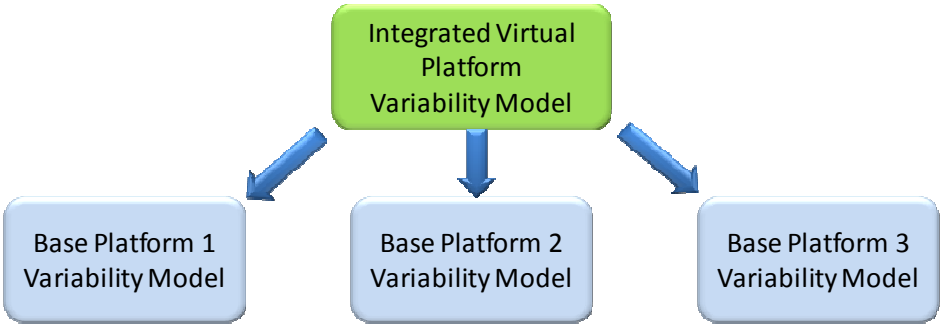


Figure 4: Mapping of variability from the virtual platform to base platforms.

In IVML we can encapsulate the variability of the various base platforms using the concept of interfaces. Thus, we can identify an arbitrarily rich variability model for each of the different base platforms, independently. Then we define an interface that describes the variability, which should be available to the construction of the virtual service platform in terms of a variability interface. This interface may also introduce an arbitrary renaming, in case this is helpful to the variability definition.

The variability model definition of the virtual platform may now reuse an arbitrary base platform variability model by importing it (respectively its interface). This allows

access to this variability. The important step is now to connect the variability on the virtual service platform level with the lower level variability of the base platform.

In IVML this can be done using constraints. Decisions regarding the virtual platform variability model, may be defined to determine lower-level decisions (i.e., base platform variability). According to the current IVML-status [D2.1] this is the main approach to support platform integration on the variability model level. However, we currently analyze further approaches like dependency generation based on composition models (cf. [RS10] for an example). Another approach, which could provide further support would be the one described in [DN+08]. However, while the former makes assumptions which are not necessarily appropriate in our context, the second does not simplify the generation of the variability model per se, but does mainly provide support in the evolution phase.

Thus, it is part of our further research to improve the composition of the variability model.

2.3.3 Making Variability Decisions in INDENICA

In the previous subsections, we focused on constructing decision models for variability management. Here, we will now focus on the application engineering part, i.e., making variability decisions.

There are various aspects that need to be taken into account in order to support decision making in application engineering. Some examples are:

- *number of decisions* - variability decision models for large and complex systems may typically contain several thousand decisions. Organizing this effectively, i.e., minimizing the number of decisions that need to be taken manually is thus very important.
- *roles* - usually people involved in the decision making process fill different roles in the organization. This may include sales people, architects, developers, etc.
- *lack of domain knowledge* - usually people involved in the decision making process and domain engineers are different persons. These people may be lacking the necessary knowledge to perform a complete configuration. In order to overcome this problem, these people should be supported with appropriate background knowledge.

The INDENICA decision support framework will offer strategies to address the aforementioned problems. Different types of strategies can be distinguished according to the following dimensions:

- by order of decision making (abstract to detailed, according to different sub-areas, according to different responsibilities and roles, according to different organizational units).

- focusing on local and global constraints (i.e., within and between these groupings)
- strategies for structuring the work (e.g., using explicit workflows, visibility, hierarchy, constraints)

We discuss some of these strategies below, which we assume may play an important role in the INDENICA project. Where appropriate, we will rely on available related work [Hub12, RD07, RGD07, Sch10]:

- *filtering support* - concepts like filtering and searching of decisions address large and complex variability decisions models by reducing the *number of decisions,* which must be taken into account by a single application engineer.

  Some approaches use *role*-specific views to support decision making. It allows people to focus on those variabilities that are important to them and also insures that, for example, sales personal does not accidentally set low-level technical decisions.

- *historical data* - some approaches rely on the reuse of configuration data of older projects. Hence, the users can use this data if they are *lacking of domain knowledge*. This approach could be extended to a recommendation service, which will offer suggestions based on the current situation and old configurations.

- *value resolution support* - if a high number of decisions values and constraints is available, this will significantly constrain the further solution space. Such information can also be used to optimize the question: which information to identify next as the various possible decisions will differ in terms of how much they further constrain the decision space.

- *additional guidance* - another approach is to introduce further (softer) information. This is, for example, used in package management systems like Eclipse, Debian, etc. Here, soft constraints like *recommends, suggests,* etc. provide guidance to the user.

- profiles - high level decisions that provide guidance for the selection of low-level decisions (market-type=Telecommunication may lead to setting thousands of other decisions), is actually pretty common, e.g., in systems like SAP. From a more formal point of view, this can be considered as a form of default reasoning. In this form it is also supported by IVML.

While, so far, individual of these strategies have been proposed in various works, their combination and integration has not yet been supported. We assume that this will be necessary to significantly reduce the complexity of decision making in complex and large-scale multiplatform scenarios.

3 Architecture Decision Making

A typical service-based application often relies on functions provided by different service platforms specialized for different domains. As a consequence, many applications pose the requirement for integration of services from one or multiple heterogeneous platforms. However, platform integration is a rather challenging task as the software architects and developers are confronted with several architectural design decisions (ADDs) at different levels of abstractions and different levels of granularity. There is a considerable amount of approaches targeting various aspects of service-based integration and adaptation [GHJV94, BMR+00, BHS07a, Fow03, HW04, VKZ05, HZ09, Dai12, HZ12]. Unfortunately, these approaches, on the one hand, have documented architecture decisions with a different focus. On the other hand, walking through several patterns scattered in different literature in order to achieve a reasonable solution for service integration is tedious and time-consuming.

During the course of building up the architectural decision, for instance, by using a pattern language, software architects have to justify how a design pattern will fit into the overall architecture and identify its dependencies to the others. Unfortunately, there are several factors introducing different sources of uncertainty for the task of selecting architectural design pattern alternatives, variants and implementations. Some examples of the sources of uncertainty are:

- Most of patterns can introduce solutions to several similar problems and can appear in many different variations.
- When sorting out appropriate patterns during the decision making process, the software architects have to balance various forces and consequences in the context of these patterns as well as numerous of related requirements that are often vague, ambiguous, and possibly competing to each other.
- Given a decision problem at hand, the software architects still have to adapt the appropriate patterns to technology and system specific contexts and options.
- In the pattern literature, design patterns are described in a rather informal narrative style using slightly different text structures by the authors [Zdu07].

Apart from the aforementioned issues, the lack of a formal mapping of the ADDs and corresponding software designs (e.g., component models, deployment models, or Service Component views in INDENICA) leads to inconsistencies and low traceability when either or both of them evolve.

The INDENICA architectural decision making support approach presented in this section aims at addressing the problems mentioned above. In particular, the main contributions are:

- To revisit the existing patterns and design decisions regarding service-based integration and adaptation of platforms and organize them in a comprehensive *pattern language* which software architects and developers can systematically reference and follow to build up an appropriate platform integration and

adaptation solution. We briefly describe the pattern language in Section 3.1 and provide detailed explanation of the pattern language in Appendix 6.1

- To propose *a novel fuzzy logic based approach* to resolving the uncertainty and variability of pattern-based decisions and providing semi-automated decision support. This approach will be presented in detail in Section 3.2.
- To propose a *constraint-based mapping approach* that enables explicit formalized mappings of architectural design decisions onto software designs such as Service Component views and supports the consistency checking between the decisions and the corresponding Service Component views. Our approach will bridge the gap between requirements and design, between ADDs and architectural views, combining ADDs with Service Component views in a formal way. This approach is also the result of the integration between the INDENICA architectural decision making support tool developed in WP1 and the view-based design time and runtime architecture developed in WP3 (see [D3.1]). Section 3.3 will elaborate our constraint-based mapping approach.

So far, we have partially published relevant scientific results at conferences in the fields of software architecture and software design, which are:

1. I. Lytra, H. Tran, and U. Zdun. Constraint-Based Consistency Checking between Design Decisions and Component Models for Supporting Software Architecture Evolution. 16th European Conference on Software Maintenance and Reengineering (CSMR), March 27-30, Szeged, Hungary, 2012.
2. I. Lytra, S. Sobernig, H. Tran, and U. Zdun. *A Pattern Language for Service-Based Platform Integration and Adaptation*. Accepted to the shepherding phase of EuroPLoP 2012.

We have also partially developed a prototypical implementation of the aforementioned contributions aiming at supporting software architects and developers in making architectural decisions and building up appropriate domain-specific service platform integration solutions. The final prototype will be reported in M36.

3.1 A Pattern Language for Service-Based Platform Integration and Adaptation

In the context of platform integration, the diversity of service platforms with respect to their functional and non-functional properties often leads to several alternative ways for successfully tailoring, adapting, and integrating those platforms. In other words, the software architects and developers are usually confronted with numerous design decisions at different levels of abstractions and different levels of granularity to arrive at a reasonable platform integration solution. The main focus of this section is a comprehensive pattern language that software architects and developers can systematically reference and follow to develop an appropriate platform integration and adaptation solution. The pattern language presented in this paper considers four essential high-level architectural decision categories in the context of service platform integration, which are *Adaptation and Integration*, *Interface Design*,

Communication Style, and *Communication Flow*. Each category constitutes a number of architectural design decisions described in terms of relevant patterns and their relationships along with their variations or alternatives and the decisive reasons leading to choosing these patterns. Based on the descriptions of this pattern language, the functional and non-functional properties of the service platforms, and particular requirements of the service-based applications built on top of the platforms, one might develop not only a platform integration solution but also a number of alternative configurations of the solution.

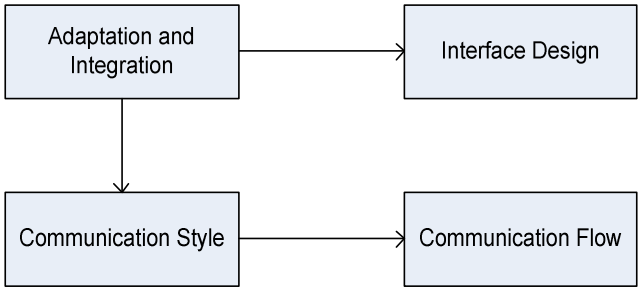


Figure 5: Overview of Pattern Language for Platform Integration

Our proposed pattern language for service-based platform integration and adaptation describes interconnected design decisions using existing patterns material from different sources, such as patterns for general software design [GHJV94], software architecture [BMR+ 00, AZ05], distributed system design [BHS07a], enterprise application architecture [Fow03], messaging [HW04], remoting middleware [VKZ05], service-oriented systems [HZ09], service design [Dai12], and process-driven SOA [HZ12]. Figure 5 gives an overview of the main categories of design decisions that we document in our pattern language. The direction of the arrows implies follow-on decision categories. In our pattern language, we consider the following major architectural decision categories: Adaptation and Integration, Interface Design, Communication Style and Communication Flow.

- **Adaptation and Integration** concerns design decisions regarding the integration of platform services into a service-based integration platform and their interface and protocol adaptation, if required.
- **Interface Design** mainly covers design decisions regarding the design of the exported interface(s) of the service-based integration platform. Decisions in the categories Adaptation and Integration can be performed in parallel to decisions in the Interface Design category. These categories mainly concern developing components and interfaces for the connections between applications, platforms, and the service-based integration platform.
- **Communication Style** describes follow-on design decisions that must be taken for each distributed component connection. The Communication Style category describes design options for connecting two components. That is, these decisions usually reside at a lower level of abstraction than the decisions in the other categories.
- **Communication Flow** describes additional follow-on decisions that must be considered in case the service-based integration platform introduces more

complex communication flows than simple forwarding from exported interface to imported interfaces. For instance, it describes how to handle requirements for aggregating or splitting the messages going through the service-based integration platform.

The aforementioned architectural decision categories aim at covering the core design space of service-based platform integration and adaptation. In Appendix 6.1, we will provide further elaborations of these categories.

3.2 Fuzzy Logic Based Approach to Support the Selection of Design Patterns

The fuzzy logic based approach elaborated in this section aims at supporting the software architects in answering the question "Which pattern fits best in the current design situation?" that appears numerous times during the design. Until now, the pattern-based decision making process has been largely ad hoc and informal and an automated decision support does not exist. Apart from that, existing models do not consider the inherent uncertainty and variability in pattern-based decisions, and it is hard to adapt existing decision models to technology or system specific contexts. Our fuzzy logic based method is the first approach that considers the uncertainty and variability of pattern-based decisions explicitly and provides semi-automated decision support. It systematizes the solution space for a design problem by considering alternative patterns, pattern variants and implementations along with their forces and consequences. Fuzzy logic helps us to deal with the imprecision and ambiguity of the design pattern selection problem and to calculate best-fitting solutions given the requirements. For this, we integrate software patterns and fuzzy logic by creating fuzzy models leveraging experts' knowledge, and provide a fuzzy inference system to make pattern decisions under uncertainty, considering patterns' forces and consequences. Along with the general fuzzy models we derive specialized fuzzy models for specific technologies and system contexts. These fuzzy models are described using a domain-specific language (DSL) and get stored in a repository. Thus, they provide reusable assets for pattern-based decision making.

3.2.1 Background on Fuzzy Logic Inference

Fuzzy Logic

Making decisions that contain uncertainty, such as the uncertainty in natural language is not an easy task. To address this problem Lotfi Zadeh introduced in 1965 the Fuzzy Logic [Zad65]. Key concepts of Fuzzy Logic are the fuzzy sets and their membership functions. Unlike classical sets fuzzy sets contain objects that satisfy imprecise properties of membership [Ros04]. In binary logic the membership function takes only two values: 0 and 1. Zadeh extended this binary membership to express various "degrees of membership" spanned in the interval [0, 1]. A membership function of a fuzzy set is a curve that defines how each point in the input space (the universe of discourse) is mapped to a membership value between 0

and 1. We use the representation $\mu_{\tilde{A}(x)}$ to express the degree of membership of element x in a fuzzy set Ã. Thus we can write:

$\mu_{\tilde{A}(x)}$ = degree to which x ∈ Ã,

$\mu_{\tilde{A}(x)} \in [0, 1]$

Fuzzy Logic allows the numerical encoding of the vague but rather simple linguistic values that humans use in their communication. For example the property scalability could be described as high, medium or low. These linguistic values can be interpreted using fuzzy sets which get mapped to overlapping membership functions (see Figure 6). The membership functions can be triangular, trapetzoidal, gaussian, etc.

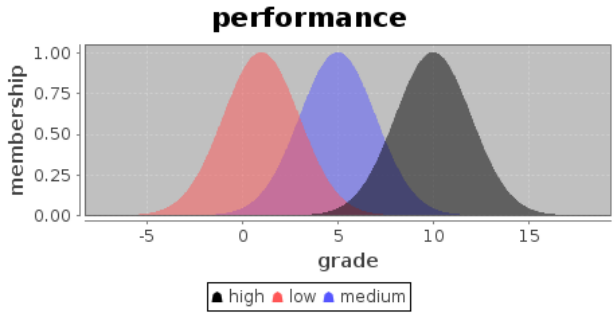


Figure 6: Gaussian membership functions for 3 linguistic values of property performance

Fuzzy Inference System

Fuzzy relations can be assembled from linguistic knowledge, expressed as IF–THEN rules. Fuzzy inference is the process of formulating the mapping from a given input to an output using fuzzy logic. Mamdani method for fuzzy inference [MA99] is the most suitable for capturing expert knowledge, as its rules allow us to describe the expertise in more intuitive and human-like manner. In this case, a fuzzy rule-based system contains simple canonical rules of the following form: "IF condition B, THEN conclusion C". The condition B can contain multiple antecedents in conjunctive or disjunctive form or combination of both.

The conclusion C is of the type "y IS Ã", where Ã is a fuzzy value of the output variable y. The fuzzy inference process comprises the following steps: fuzzify the input, evaluate the fuzzy rules, aggregate the outputs to reach the final decision, and defuzzify the output to obtain a crisp value. In our work we integrate software patterns and fuzzy logic and apply a Mamdani-type inference method to make pattern decisions under uncertainty.

3.2.2 Approach Details

Overview

To have an overview of the steps of our approach our fuzzy logic based method is illustrated in Figure 7. To create our DSL we used the Eclipse toolkit Xtext[2] for

[2] http://www.eclipse.org/Xtext/

creating textual DSLs. Along with our Eclipse tooling we used the open source Fuzzy Logic library jFuzzyLogic[3] which implements the Fuzzy Control Language (FCL) specification [Int00] and provides a DSL and a fuzzy inference system. FCL is a rather general language that implements fuzzy control. We choose, however, to focus on the domain of pattern selection and we develop a simple expressive DSL to describe our fuzzy models which is definitely easier to read and write. The software architects use our Eclipse tools to capture generic and technology-specific pattern-based knowledge. That means that they have to define the patterns with their forces as well as to describe the fuzzy rules using our DSL. After that, FCL files can be automatically generated from DSL files and stored in a repository in order to be ready for use from the fuzzy inference system. The requirements engineers use the Eclipse tooling to give the requirements in crisp values using a grading system (e.g. 1-10) for fuzzy input variables like scalability and reliability. The fuzzy inference system returns the appropriate design patterns and their ranking for the given requirements as we can see for example in Figure 8.

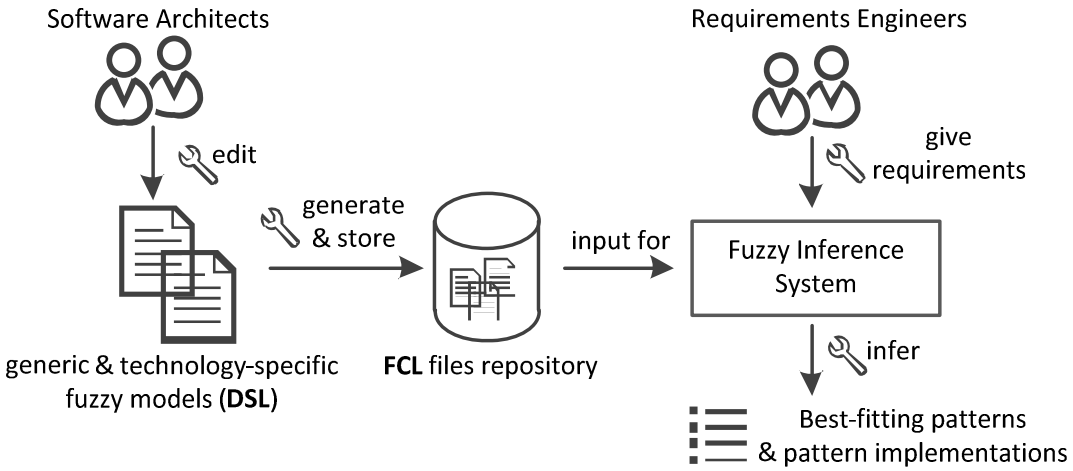


Figure 7: Fuzzy Logic Approach for Pattern-Based Decisions

[3] http://jfuzzylogic.sourceforge.net

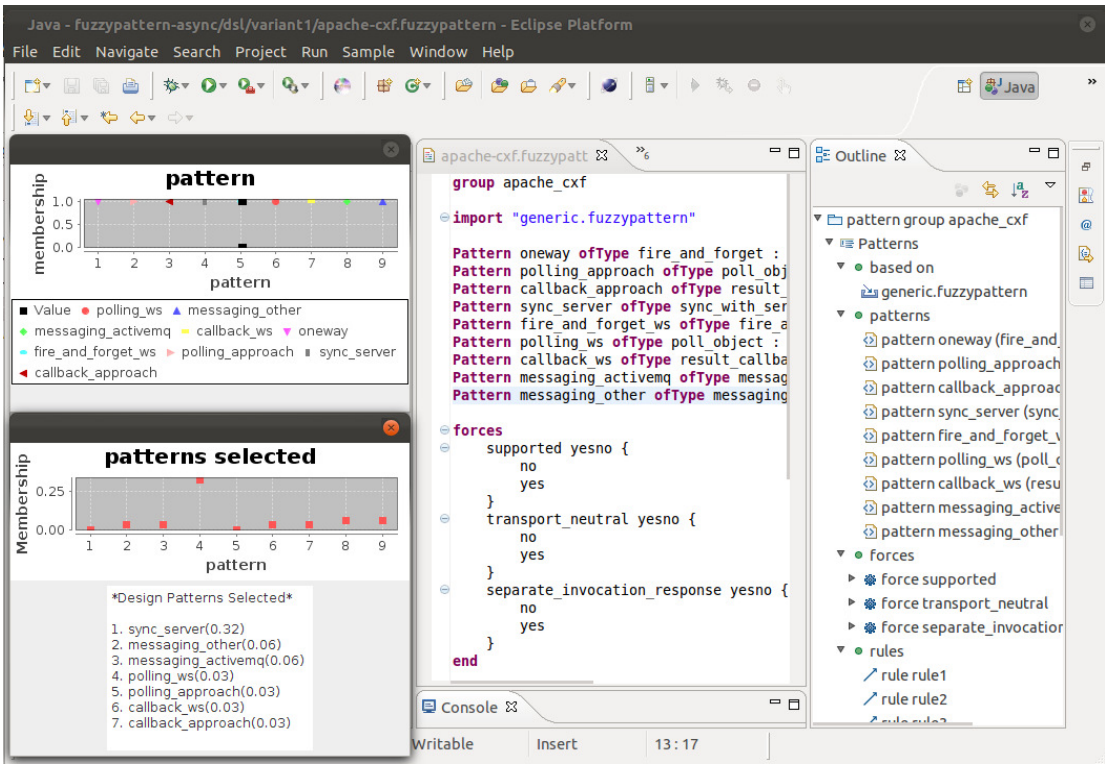


Figure 8: Eclipse Tooling

Domain Specific Language for Fuzzy Logic Models

Fuzzy Pattern-Based Decision Models

In this section we explain how to define our fuzzy pattern-based decision models using the DSL. The pattern documentations help us to find the related patterns and define the forces and consequences of the patterns that are mapped to the fuzzy inputs. Patterns that refer to a specific domain, problem or technology can be organized in groups, for example:

group asynchronous_invocation

Pattern fire_and_forget

Pattern sync_with_server

...

Before we write the rules for the pattern selection we need to define the patterns' forces and consequences and their possible values – the fuzzy sets. Along with the fuzzy sets, we need to select their membership functions. For the quality attributes (e.g. reliability) we may choose triangular, trapetzoidal, gaussian, etc. membership functions, however, for properties that do not contain any fuzziness (e.g. supports acknowledgement can be either yes or no) we use singleton membership functions. The patterns and pattern variants are the values of the fuzzy output variable pattern with a singleton membership function, as they do not contain any fuzziness as well. We define quality attributes with their linguistic values and membership functions as following:

forces

reliability gauss { low medium high }

...

end

schemas

gauss { gauss [1 2] gauss [5 2] gauss [10 2] }

...

end

The precise shape of the membership curves is not so important, as their approximate placement on the universe of discourse, the number of curves used and their overlapping character are the most important ideas [Ros04]. The overlapping of the curves helps us express the ambiguous and vague borders between the linguistic attribute values. But we should keep in mind that the type of the membership functions, the grade of their overlapping and the number of the variable values depend on the engineers' preferences, experience and intuition and affect the precision of the results. Therefore, the membership functions should be tuned manually (e.g. empirically by trial and error).

Finally, the experts' experience gets translated into fuzzy rules as in the following example:

if performance is high and reliability is medium then poll_object

We can also assign weights to the rules according to their importance.

Technology-Specific Decision Models

To include technology and system-specific, knowledge pattern implementations can extend the generic patterns and thus inherit their membership functions and rules:

group apache_cxf

import "generic.fuzzypattern"

Pattern oneway ofType fire_and_forget

...

The generic rules must be also combined with the specialized rules. Therefore, we generate the FCL files from the DSL files according to the following rules:

1. For the generic fuzzy models, all input variables, fuzzy sets, membership functions and rules are translated as they appear in the DSL.
2. For technology-specific fuzzy models the FCL files contain all input variables, fuzzy sets and membership functions they inherit from the generic fuzzy models. They contain also all the rules that refer to patterns which they inherit as well as combined rules that are constructed. Imagine the case where we claim that the

fire and forget pattern provides high performance. We should be allowed to use this pattern even if we are satisfied with low or medium performance.

Fuzzy Inference System

We use the max-min inference to infer the degree of appropriateness of each of the patterns represented by the length of the spikes of the fuzzy singletons. The membership functions of the fuzzy sets of the conclusions are limited to the degree of accomplishment of the condition and then, in turn, get combined to create a fuzzy set by forming a maximum. In our case, the defuzzification does not produce any useful information, because the order of the different software patterns in the output space is arbitrary.

3.2.3 Design space for an example of INDENICA case study

DSL-Based Specification of Design Space

To illustrate our approach we elaborate an example from the INDENICA case study. We consider an excerpt of the remoting pattern language described in [VKZ04] as an example of communication between one of the integrating platforms and the Operator Application. The remoting patterns describe the inner workings of distributed middleware systems (such as Web Services, CORBA, Java RMI and .NET Remoting) and explain how to use them to create distributed systems. In our example we focus on the asynchronous remote invocation patterns. Unlike blocking synchronous invocations, asynchronous invocations allow client applications to resume with their work while waiting for a response to a remote invocation, thus improving scalability and performance. In our example we examine the following patterns:

1. *fire and forget:* A server application provides remote objects that get invoked by the clients without expecting any return value.
2. *sync with server:* The client sends an invocation as in fire and forget and waits for a reply from the server informing it about the successful reception.
3. *poll object:* The server delivers the results to a poll object which is queried by the client in certain intervals.
4. *result callback:* The server notifies the client once the results become available.

Our first step is to define the pattern forces and consequences (our fuzzy sets) and their membership functions. By reading the relevant documentation, we decided to use the following decision criteria for the generic DSL files: performance, reliability, acknowledgement and log application error. We experimented with different number of fuzzy sets and membership functions in order to tune them and report our results. For the writing of the rules we consulted the pattern documentation as well as our experience. In the following, we present an excerpt of a DSL file which captures generic decision knowledge on the asynchronous invocation patterns. Of course, what we present here reflects our interpretation of the design pattern

documentation and our experience and can be modified to include more or less forces, different membership functions, different sets of rules, etc.

group asynchronous_invocation

Pattern fire_and_forget

Pattern poll_object

...

forces

performance gauss { low medium high }

acknowledgement yesno { no yes }

...

end

schemas

gauss { gauss [0 2] gauss [5 2] gauss [10 2] }

yesno { singleton [0] singleton [1] }

end

rules

rule1 --> if performance is atmost high and reliability is

low and acknowledgement is no and log_application_error

is no then fire_and_forget

rule2 --> if performance is atmost high and reliability is

atmost medium and acknowledgement is yes and

log_application_error is yes then poll_object

rule3 --> if performance is atmost high and reliability is

atmost medium and acknowledgement is yes and

log_application_error is no then poll_object with 0.1

...

end

This representation can be enough to capture application-generic explicit knowledge, but software engineers need more tangible technology-specific knowledge to apply these patterns to a problem at hand. To support this transition from application-generic to application-specific knowledge, which is called Utilization in the literature [FB09], we did a research for the existing frameworks and solutions that allow us to embed the asynchronous invocation patterns in the application domain. Thus, we investigated all the pattern implementations as well as the specific

decision criteria for the following frameworks: Apache CXF[4], Apache Axis 2[5], Metro 1.2[6], .NET[7] and CORBA[8]. The technology-specific DSL files import the generic DSL file and the pattern implementations extend the generic design patterns. Following we present an excerpt of a DSL file that captures knowledge about CORBA.

group corba

import "generic.fuzzypattern"

Pattern AMI_polling ofType poll_object : "AMI polling model"

Pattern AMI_callback ofType result_callback : "AMI callback model"

Pattern oneway_sync_none ofType fire_and_forget : "Reliable one-way SYNC_NONE"

Pattern oneway_sync_target ofType fire_and_forget : "Reliable one-way SYNC_WITH_TARGET"

Pattern oneway_sync_server ofType sync_with_server : "Reliable one-way SYNC_WITH_SERVER"

forces

complexity gauss { low medium high }

...

end

rules

rule1 --> if complexity is low then AMI_polling

rule2 --> if complexity is atmost medium then AMI_callback

...

end

3.3 Connect Architectural Decisions to Service Component Views

The relationship between the INDENICA architectural decision making support and the View-based Modeling Framework (VbMF) [D3.1] developed in the context of WP3 reflects a typical issue in the fields of software architecture and software design. That is, the gap between architectural design decisions (ADDs) and software designs will lead to inconsistencies between ADDs and designs and implementations when the system evolves.

[4] http://cxf.apache.org

[5] http://axis.apache.org

[6] http://metro.java.net

[7] http://www.microsoft.com/net

[8] http://metro.java.net

ADDs capture knowledge that may concern a software system as a whole, or one or more components of software architecture. In recent years, software architecture is seen more and more as a set of principal ADDs rather than the components and connectors constituting a system's design [JB05]. The idea behind this new perspective is to document not only components and connectors but also the design rationale of the architecture as well as to contribute to the gathering of Architectural Knowledge (AK). Unfortunately, in practice, the ADDs frequently do not get maintained over time as the requirements and the design of the software system change and they often do not get synchronized with other architectural views that represent the system structures [ZZGL08]. Moreover, the lack of formal mappings of the ADDs and architectural views leads to inconsistencies and low traceability.

Our constraint-based mapping approach aims at bridging the gap between requirements and design, between ADDs and architectural views, combining ADDs with Service Component views in a formal way. For this purpose, we propose a mapping model from ADDs onto the Service Component view that is used to describe the architecture of virtual service platforms [D3.1]. The Service Component view can easily be mapped to popular component-based modeling approaches such as the UML 2 Component Diagram[9]. Based on the mapping model, we can generate the Service Component view and constraints for consistency checking between ADDs and the Service Component view by using model-driven techniques. During the evolution of the software systems, the ADDs may be altered. Accordingly, the Service Component view and the constraints can be re-generated, and therefore, remain in sync with the ADDs. As the Service Component view is changed, the constraints checking shall verify whether these changes invalidate the corresponding ADDs or not. In case inconsistencies between the ADDs and the Service Component view occur, the relevant design elements that invalidate the ADDs shall be highlighted.

[9] Section 8 in http://www.omg.org/spec/UML/2.2/Superstructure/PDF

3.3.1 Approach Overview

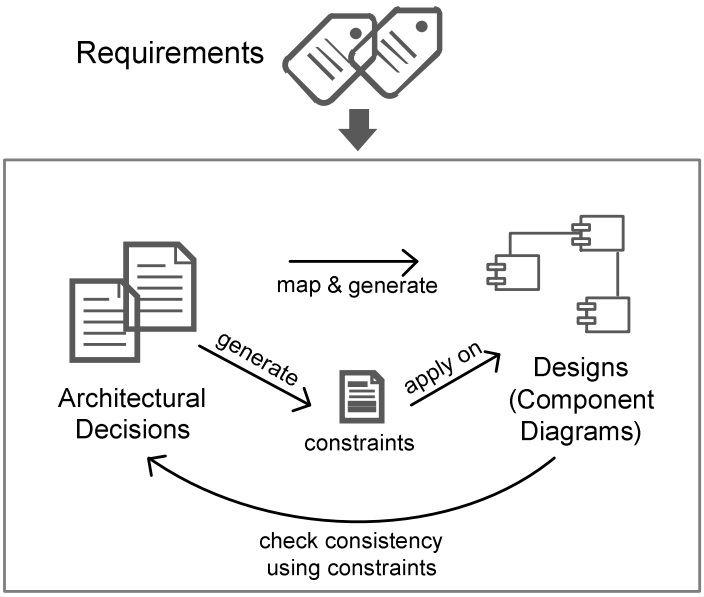


Figure 9: Mapping and consistency checking between ADDs and Designs

Our contributions to the software architecture and design process are summarized in Figure 9. Our approach can be used not only at design time but also during software system evolution and maintenance. The mapping between ADD and the Service Component view can enable traceability and consistency checking between them and automate the generation of an initial instance of a Service Component view that reflects the design decisions. Apart from the generation of the Service Component view, we introduce the generation of constraints that are used for consistency checking between the ADDs and the Service Component view. As the human decision is important in the interpretation of the ADDs and their connection to software architectures, the mapping from ADDs to the Service Component view shall be performed in a semi-automatic manner. After the mapping is established, the generation of Service Component views and constraints is fully automated. The software architects and designers can use our tool to analyze and estimate how the changes of certain ADDs shall affect the design and/or leverage the generation to come up with a recommendation design directly derived from the ADDs rather starting from scratch. Apart from that, we aim at reducing the cost and burden of the maintenance of both ADDs and Service Component views. Information included in the ADDs shall be fully reflected in the Service Component view. Changes at the Service Component view or the ADDs that cause inconsistencies get highlighted. Hence, we aim at ensuring that ADDs and Service Component views remain consistent with each other and are traceable back and forth.

Solution Details

In this section we leverage the Service Component view presented in [D3.1] which is used for specifying the architectures of virtual service platforms in INDENICA. Next, we introduce a mapping model from ADDs to the Service Component view. We elaborate afterwards this mapping model for the generation of the Service

Component view and of constraints for checking consistency using model-driven techniques. We revisit the case study and explain how our approach can be applied to bridge the gap between ADDs and the designs. For the sake of integrating different models, we utilize the concepts such as *NamedElement* and *AnnotatedElement* of the Core model [D3.1].

3.3.1.1.1 Mapping of ADDs to Service Component Views

Capturing architectural design decisions is important for analyzing and understanding the rationale and implications of these decisions and reducing the problem of architectural knowledge vaporization [HAZ07, CBB+ 10]. Several existing approaches have been proposed for addressing the aforementioned challenges [TA05, ZGK+ 07]. However, none of these approaches supports to explicitly capture the causal relationships between the decisions and relevant design artifacts. These relationships are crucial because they enable the traceability between ADDs and the designs for analyzing the design coverage (e.g., checking whether some ADDs have been realized or not), estimating change impacts (e.g., which design artifacts are affected by certain changes of ADDs), checking consistency between ADDs and the designs, and many other tasks. In this paper, formalizing these relationships in order to bridge the gap between ADDs and the designs as well as using them for generating Service Component views and constraints for checking consistency are the key contributions.

We propose a generic concept, namely, AD, for representing ADDs (see Figure 10). Each AD has a number of Outcomes which are inputs for designing Service Component views. An Outcome can be mapped to a certain element of the Service Component view such as a component or connector. The ADD's Outcomes are often expressed in natural language, and therefore, human interventions are necessary for instantiating the Mapping model. For instance, if an Outcome implies a new property of a component, the name of the component and the name, type, and value of the property should be defined manually.

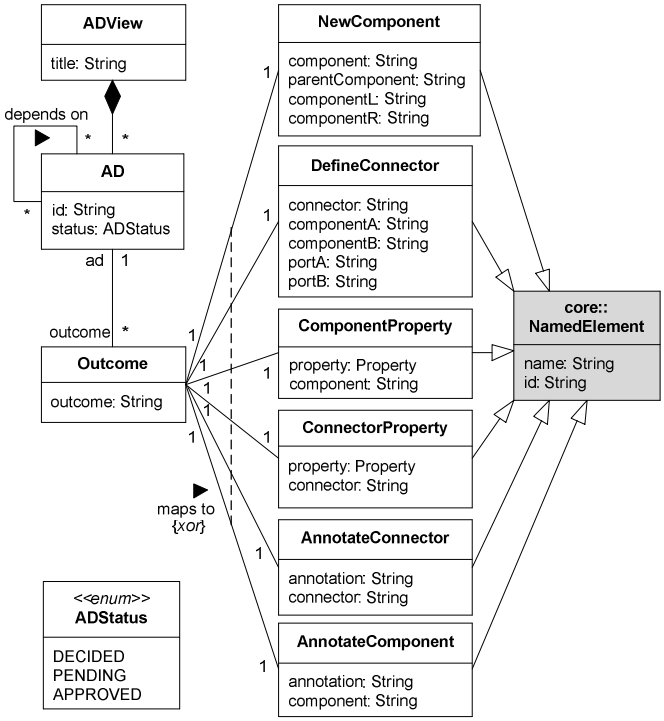


Figure 10: An excerpt of the Mapping model

3.3.1.1.2 Generation of Service Component View

The Mapping model presented in the previous section shall connect the architectural decisions and designs at various levels of granularity. Each mapping represents a relationship between a certain Outcome of an ADD to an element of the design (in this case, the Service Component view) such as a component, port, connector, an annotation, property, and so on. Among the benefits of the Mapping model mentioned before, we can also leverage these mappings to generate an initial Service Component view that can serve as a starting point for designing the corresponding software architecture. In case of a green-field development scenario (i.e., there are no existing designs), this step can save tedious efforts that the software architects and designers have to spend to sketch the designs from scratch. Nevertheless, in case there are existing designs, the generated designs can be referenced for analyzing the deviation as well as estimating necessary changes in order to accomplish the architectural decisions. In addition, constraint checking can be performed beforehand to ensure that the Service Component view can be updated without any errors. The constraint-checking at this stage is necessary not only for generating the Service Component view but also for finding out the issues due to that the Service Component view cannot be generated, if any. For instance, assume that we want to assign a property to a connector linking the ports of two components. We illustrate the templates that will be used for the generation of the constraints that will be checked before the generation of the Service Component view. The variables within the notion $..$ (e.g. $componentA$, $portA$) shall be substituted with concrete values during the constraint instantiation.

```
∃c ∈ Components | c.name = $componentA$
∃c ∈ Components | c.name = $componentB$
∃c1, c2 ∈ Components | c1.name = $componentA$
∧ c2.name = $componentB$
∧ (∃ con ∈ Connectors)
∧ (∃ p1 ∈ c1.ports, ∃ p2 ∈ c2.ports |
(p1.name = $portA$ ∧ p2.name = $portB$)
∧ ((p1 = con.source ∧ p2 = con.target)
∨ (p1 = con.target ∧ p2 = con.source)))
```

Whenever a certain constraint is not satisfied, specific errors shall be reported. The stakeholders have to fix those errors before the Service Component view can be generated properly. The Service Component view gets successfully generated if and only if all constraints are satisfied. Each kind of mapping might imply one or more consequent updates to the Service Component view. For example, suppose that we want to assign the property to a connector, after the constraints are checked and satisfied, a new property should be created and annotated to the connector. We implement the constraint checking using the declarative constraint checking language Check. The Service Component view is generated based on the mapping model by using the expression language Xtend. Xtend and Check are powerful OCL-like expression languages provided by the Eclipse Model-to-Text (M2T) project[10]. The process of constraint-based model validation and Service Component view generation is integrated using the modeling workflow language provided in the Eclipse M2T project.

3.3.1.1.3 Generation of Consistency Checking Constraints

Each of the aforementioned types of mapping is related to a set of constraint templates from which concrete constraint "instances" are generated. The constraint templates have been already defined for each kind of mapping. The constraint "instances" are generated using the Velocity template engine[11] and the attributes to be replaced get values from the mapping of the ADDs to the Service Component view. The generated constraints are also based on the Check language. We illustrate an excerpt of the constraint templates used for creating constraints on the mapping of an ADD to a new property of a certain component. As mentioned above, the variables within the notion $..$ shall be substituted with concrete values during the instantiation of the constraints.

```
∃ p ∈ P roperties| p.name = $name$
∧ p.value = $value$
∧ (∃ c ∈ Components | c.name = $component$)
∧ (∃ a ∈ c.annotations | a = p)
```

These constraints are generated and validated as described in the previous Section for the constraints that have to be checked before the Service Component view

[10] http://www.eclipse.org/modeling/m2t

[11] http://velocity.apache.org

generation. The consistency checking constraints shall check the consistency between the ADDs and the Service Component view.

3.3.2 An example scenario in INDENICA case study

To illustrate our approach, we apply it for a simplistic excerpt extracted from the INDENICA case studies [D5.1]. A material flow computer in a warehouse receives orders from an ERP system and the communication between them is accomplished through the VSP. Our approach starts at the development stage where the requirements have been resolved into the ADDs. The ADDs are usually captured in an informal way using document templates or meta-models [TA05, ZGK+ 07]. The key concepts that our approach focuses on are architectural decisions and their implications. Therefore, most of existing approaches for capturing and representing ADDs can be applied because most of them provide these essential concepts. In this example, the architecture decision description template proposed in [TA05] is exemplified to capture ADDs. We show an excerpt of the documented ADDs including three architectural decisions:

- **D01** Expose Place Order functionality as Apache CXF Web Services.
- **D02** Connect Place Order to VSP using encrypted HTTPS connection and compress the messages using standard HTTP/1.1.
- **D03** Implement Order Picking as a BPEL flow that is able to handle asynchronously 1000 orders per minute.

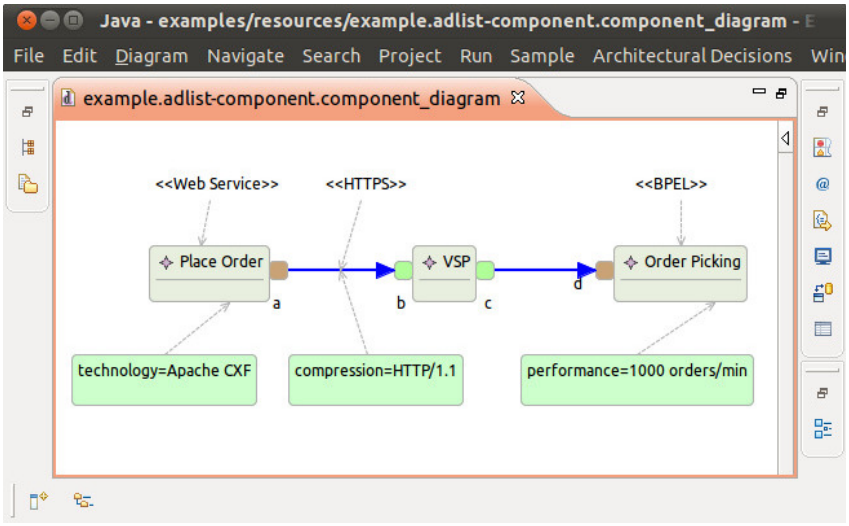


Figure 11: Eclipse Tooling for ADD and Service Component view Development

At this stage, we have necessary information that shall be used as inputs for our approach. The proof of concept tooling of our approach based on EMF[12] and GMF[13] is shown in Figure 11. Our tool can support the development and generation of the

[12] http://www.eclipse.org/emf

[13] http://www.eclipse.org/gmf

constraints as well as the generation and the graphical representation of the Service Component view.

Mapping of ADDs to Service Component view

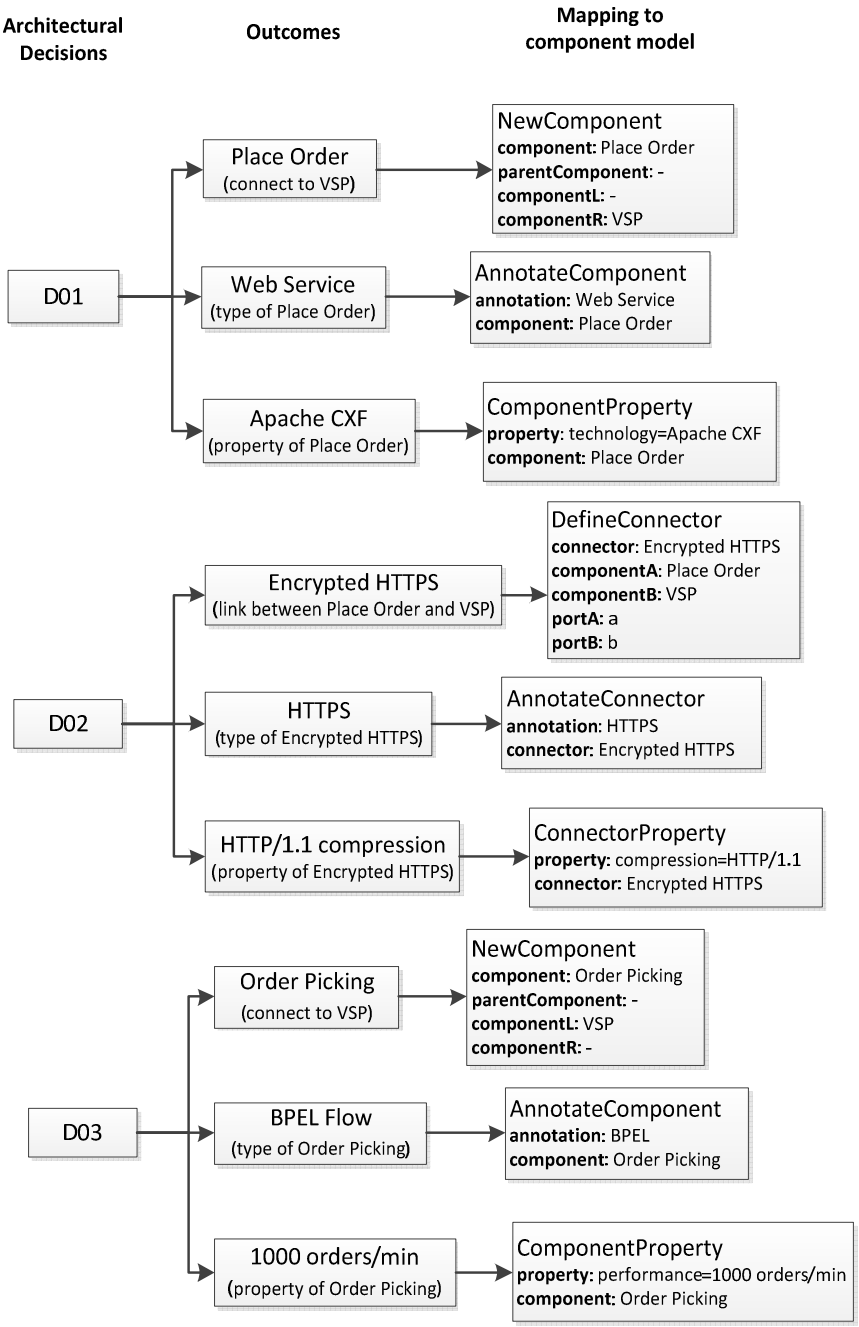


Figure 12: Mapping of architectural design decisions to the Service Component view

We extract the useful information of the ADDs that can be mapped to a component, a connector, an annotation, or a property. Figure 12 presents an excerpt of the Mapping model between the ADDs and the Service Component view. For example, the first ADD refers to a new component (Place Order) connected to the VSP

component which will be implemented as a Web Service (annotation of component Place Order) using Apache CXF (property of component Place Order).

Service Component View Generation

Once the mapping of the ADDs to the Service Component view is accomplished, we can generate an initial instance of the Service Component view. In order to see how the generation is done, let us take the AnnotateComponent mapping from the decision **D01**. Before creating a new annotation and attach it to a component, we must ensure that the same annotation has not been already assigned to the component. If the above function for our component graphical view and for component = "PlaceOrder" and annotation = "WebService" does not return any annotation we proceed with the creation of the new annotation.

```
component::Stereotype componentHasAnnotation(
component::ComponentView cv, component::
Component component, String annotation):
component.annotation.select(a|(a.text ==
annotation) && (cv.annotation.typeSelect(
component::Stereotype).exists(s|s == a)));
```

If the generation of the Service Component view completes without errors, the visualization of the Service Component view that we get using our Eclipse Tooling is shown in Figure 11. A component is depicted in terms of a box associated with its ports. Two ports can be connected by a connector which is an arrow going from the required to the provided one. The stereotypes are shown inside the symbol "" (e.g., HTTPS) and the properties are shown in form of "name[:type]=value" (e.g., "technology=Apache CXF").

```
annotateComponent(component::ComponentView cv,
component::Component component, String
annotation):
let stereotype = new component::Stereotype :
stereotype.setText(annotation)
-> cv.annotation.add(stereotype)
-> component.annotation.add(stereotype);
```

Generation of Consistency Checking Constraints

Now we explain how the constraints that check the consistency of the Service Component view get generated. Let us consider the AnnotateComponent mapping from the ADD **D01**. The AnnotateComponent is mapped to the following constraint template for checking whether a component is associated with a specific stereotype or not (note that a variable is inside the notion $..$).

```
context component::ComponentView ERROR
"(Architectural Decision --> $ad$)
Component $component$ is not annotated as $annotation$":
element.typeSelect(component::Component)
   .exists(c|c.component == "$component$"
    && annotation.typeSelect(component::Stereotype)
       .exists(s|c.annotation.exists(a|a == s && s.text ==
"$annotation$")));
```

In our example, the component “Place Order” shall be annotated as “Web Service” (i.e., annotation=Web Service) according to the decision ADD **D01**. The resulting instantiated constraint is shown following.

```
// Check whether component is annotated
context component::ComponentView ERROR
"(Architectural Decision --> D01)
Component Place Order is not annotated
as Web Service":
element.typeSelect(component::Component).
exists(c|c.name == "Place Order" &&
annotation.typeSelect(component::
Stereotype).exists(s|c.annotation.
exists(a|a == s && s.text == "Web
Service")));
```

The above example illustrates a great advantage of our approach: a constraint template shall be defined once but can be efficiently reused and instantiated for several corresponding mappings from the ADDs to the Service Component view.

4 Summary and Conclusion

INDENICA faces the difficult problem of providing support for integrated domain-specific service-platforms. Even compared to the development of classical service platforms or service-based applications, this is extremely complex. This is due to the large range of decisions faced by requirements engineers, software architects and developers, who are confronted with many decisions stemming not only from specific non-functional and functional requirements of the applications but also from the variety of the underlying domain-specific service platforms and the possibilities in integrating them. .

In this document, we reported on the INDENICA decision support framework, a comprehensive, multi-step method which aims at addressing decision throughout the INDENICA-methodology.

The support framework addresses requirements engineering with the sub-parts of requirements capture and decision space modelling, requirements prioritization, and variability decision modelling and resolution. On the architectural level, we introduced a pattern language for service-platform integration and adaptation, provided a systematic approach to select specific design patterns and discussed how to integrate these architectural decisions in the view-based modelling framework described in [D3.1].

Our future endeavour that will be reported in the final version of this deliverable is, on the one hand, to complete the decision supporting framework for domain-specific platforms as well as virtual service platforms. On the other hand, we plan to accomplish an integration of the aforementioned approaches to form a unified decision framework and support parts of it by tools. We will evaluate and assess the decision framework with INDENICA case studies [D5.1].

5 References

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. A PatternLanguage: Towns, Buildings, Construction. Oxford University Press, New York, 1977.

[AZ05] P. Avgeriou and U. Zdun. Architectural patterns revisited – a pattern language. In Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), pages 1–39, Irsee, Germany, July 2005.

[BCK03] L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice, volume 2. Addison-Wesley Professional, 2003.

[BHS07a] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. Pattern-Oriented Software Architecture — A Pattern Language for Distributed Computing, volume 4 of Wiley Series in Software Design Patterns. John Wiley & Sons Ltd., New York, 2007.

[BMR+ 00] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, editors. Pattern-Oriented Software Architecture – A System of Patterns. John Wiley & Sons Ltd., Chichester, England, 2000.

[CBB+ 10] Paul Clements, Felix Bachmann, Lens Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. Documenting Software Architectures: Views and Beyond. Addison-Wesley, 2nd edition, 2010.

[Cop96] J. Coplien. Software Patterns. SIGS Mgt. Briefings. SIGS Books & Multimedia, 1996.

[D1.2.1] INDENICA Deliverable D1.2.1 Requirements Engineering Framework, Language and Tools for Service Platforms, 2011-09-30

[D2.1] INDENICA Deliverable D2.1, Open Variability Modelling Approach for Service Ecosystems. 2012-01-31

[D3.1] INDENICA Deliverable D3.1 View-based Design Time and Runtime Architecture for Tailoring VSPs. 2011-10-18

[D3.2]. INDENICA Deliverable D3.2 Architecture for Role-Based Governance of Virtual Service Platforms. 2012-02-18

[D5.1] INDENICA Deliverable D5.1 Description of Feasible Case Studies. 2011-07-31

[Dai12] Robert Daigneau. Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services. Addison-Wesley, 2012.

[DN+08] D. Dhungana, T. Neumayer, P. Grünbacher, R. Rabiser. *Supporting the Evolution of Product Line Architectures with Variability Model Fragments.* Working IEEE/IFIP Conference on Software Architecture, 327-330, 2008.

[JE09] I. John, M. Eisenbarth, A Decade of Scoping – A Survey, In Proceedings of the 13th International Conference on Software Product Lines (SPLC), pp. 31 – 40, 2009.

[FB09] Rik Farenhorst and Remco C. Boer. Knowledge Management in Software Architecture: State of the Art. pages 21–38, 2009.

[Fow03] Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 4th edition, 2003.

[GHJV94] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison Wesley Professional Computing Series. Addison Wesley, October 1994.

[HAZ07] Neil B. Harrison, Paris Avgeriou, and Uwe Zdun. Using Patterns to Capture Architectural Decisions. IEEE Softw., 24(4):38–45, July 2007.

[HC01] George T Heineman and William T Councill. Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley, 2001.

[HSH06] A. Helferich, K. Schmid, G. Herzwurm. *Product Management for Software Product Lines: An Unsolved Problem?* Communications of the ACM, Vol. 49, pp. 66-67, 2006.

[Hub12] Arnauld Hubaux. Feature-based Configuration: Collborative, Dependable, and Controlled. University of Namur, Belgium, 2012.

[HW04] Gregor Hohpe and Bobby Woolf. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2nd edition, 2004.

[HZ09] Carsten Hentrich and Uwe Zdun. A Pattern Language for Process Execution and Integration Design in Service-Oriented Architectures. In James Noble and Ralph Johnson, editors, Transactions on Pattern Languages of Programming I, volume 5770 of Lecture Notes in Computer Science, pages 136–191. Springer Berlin / Heidelberg, 2009.

[HZ12] Carsten Hentrich and Uwe Zdun. Process-Driven SOA: Patterns for Aligning Business and IT. Infosys Press, 2012.

[Int00] International Standard CEI/IEC. Programmable Controllers Part 7: Fuzzy Control Programming. Technical Report 61131-7, IEC-International Electrotechnical Commission, 2000.

[JB05] Anton Jansen and Jan Bosch. Software Architecture as a Set of Architectural Design Decisions. In The 5th Working IEEE/IFIP Conference on Software Architecture (WICSA ’05), pages 109–120. IEEE Comp. Soc., 2005.

[KB98] Wojtek Kozaczynski and Grady Booch. Guest Editors’ Introduction: Component-Based Software Engineering. IEEE Software, 15(5):34–36, 1998.

[KJ04] Michael Kircher and Prashant Jain. Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management. Wiley, June 2004.

[KLvV06] Philippe Kruchten, Patricia Lago, and Hans van Vliet. Building Up and Reasoning About Architectural Knowledge. Quality of Software Architectures, pages 43–58, 2006.

[LK07] Larix Lee and Philippe Kruchten. Capturing Software Architectural Design Decisions. In 2007 Canadian Conf. on Electrical and Computer Engineering, pages 686–689. IEEE, 2007.

[vLRS07] F. J van der Linden, E. Rommes, K. Schmid. Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering, Springer, 2007.

[MA99] E. H. Mamdani and S. Assilian. An Experiment in Linguistic Synthesis with a Fuzzy Logic Controller. International Journal of Man-Machine Studies, 51(2):135–147, 1999.

[MT00] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, 26(1):70–93, 2000.

[Po09] Klaus Pohl: Requirements Engineering, Fundamentals, Principles and Techniques. Springer Verlag, Berlin und Heidelberg, 2009.

[RD07] R. Rabiser and D. Dhungana. Integrated Support for Product Configuration and Requirements Engineering in Product Derivation. In Proc. of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'07), 2007.

[RGD07] R. Rabiser, P. Grünbacher, and D. Dhungana. Supporting Product Derivation by Adapting and Augmenting Variability Models. In Proc. of the 11th International Software Product Line Conference (SPLC 2007), 2007.

[Ros04] T.J. Ross. Fuzzy logic with engineering applications. John Wiley, 2004.

[RS10] M. Rosenmüller and N. Siegmund. Automating the configuration of multi software product lines. In Proc. of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VAMOS'10), pp 123–130, 2010.

[Sch10] K. Schmid, Variability Modelling for Distributed Development: A Comparison with established practice. Proceedings of the 14th International Conference on Software Product Line Engineering (SPLC'10), pp. 155-165, 2010.

[SLK09] Mojtaba Shahin, Peng Liang, and Mohammad Reza Khayyambashi. Architectural design decision: Existing models and tools. In IEEE/IFIP Conf. on Software Architecture/European Conf. on Software Architecture, pages 293–296. IEEE, 2009.

[SSRB00a] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. Pattern-Oriented Software Architecture, chapter Extension Interface, pages 141–174. John Wiley & Sons Ltd.Wiley, Chichester, England, 2000.

[SSRB00b] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. Pattern-Oriented Software Architecture, chapter Interceptor, pages 109–141. John Wiley & Sons Ltd.Wiley, Chichester, England, 2000.

[SaVa01] T.L. Saaty, L.G.Vargas: Models, Methods and Concepts of the analytic Hierarchic Process. Kluwer Academic, 2001.

[Szy02] Clemens Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Boston, MA, USA, 2nd edition, 2002.

[TA05] J. Tyree and A. Akerman. Architecture Decisions: Demystifying Architecture. IEEE Softw., 22(2):19–27, 2005.

[VKZ05] Markus Völter, Michael Kircher, and Uwe Zdun. Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware. Software Design Patterns. John Wiley & Sons Ltd., Chichester, England, 2005.

[UlRuGa10] Muhammad Irfan Ullah, Günther Ruhe, Vahid Garousi: Decision support for moving from a single product to a product portfolio in evoving software systems. The Jpurnal of Systens and Software 83 (2010).

[Vla09] Axel van Lamsweerde. Requirements Engineering: From System Goals to UML Models to Software Specifications. John Wiley, 2009.

[Vog01] Oliver Vogel. Service Abstraction Layer. In Proceedings of EuroPLoP 2001, Irsee, Germany, 2001.

[Zad65] L A Zadeh. Fuzzy sets. Information and Control, 8(3):338–353, 1965.

[Zdu07] Uwe Zdun. Systematic pattern selection using pattern language grammars and design space analysis. Software Practice & Experience, 37:983–1016, July 2007.

[ZGK+ 07] Olaf Zimmermann, Thomas Gschwind, Jochen Kuester, Frank Leymann, and Nelly Schuster. Reusable architectural decision models for enterprise application development. In Proc. of QoSA 2007, pages 15–32. Springer-Verlag, 2007.

[ZZGL08] Olaf Zimmermann, Uwe Zdun, Thomas Gschwind, and Frank Leymann. Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method. In 7th IEEE/IFIP Conf. on Software Architecture, pages 157–166. IEEE, 2008.

6 Appendix

6.1 Detailed Description of the Pattern Language for Service-Based Platform Integration and Adaptation

In Section 3.1, we briefly describe the pattern language that software architects and developers can use to build up architectural decision trees and design solutions for service-based platform integration and adaptation. In this appendix, we elaborate the four major aspects of the pattern language and illustrate its usage via an excerpt of the INDENICA case studies.

6.1.1 Integration and Adaptation

The simplest case of integrating a platform into an application is to directly invoke the platform services from the application code. However, often we would like to avoid direct invocations in order to support abstraction or stable interfaces. In addition, often simple direct invocations are not enough, as the integration logic should introduce extra functionality, such as logging, monitoring, indirecting, or adapting the platform access. Provided that the interfaces offered by the platform are compatible to each other and the extra functionality needed does not change the invocation flow, the PROXY pattern [GHJV94] can be used to perform platform integration. Examples of extra functionalities that can be handled using a PROXY are logging functions, monitoring functions, or access control.

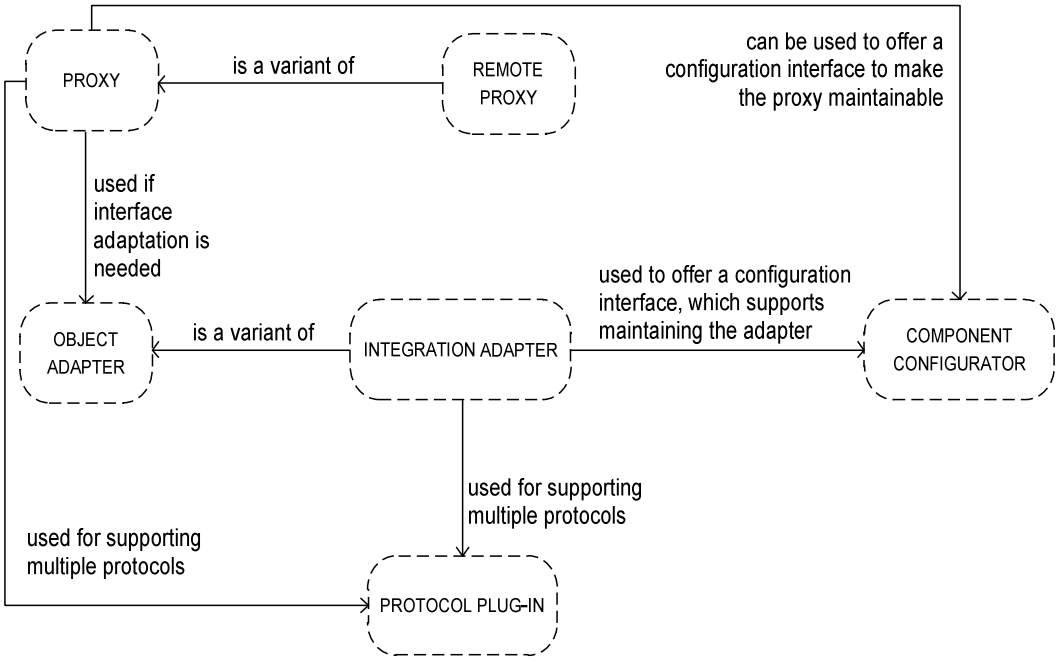


Figure 13: Platform Integration and Adaptation Patterns

Figure 14 illustrates direct invocations vs. proxy-based platform integration. In this simple, schematic example, the PROXIES introduce extra functionality for monitoring the invocation flow from the application to the platform.

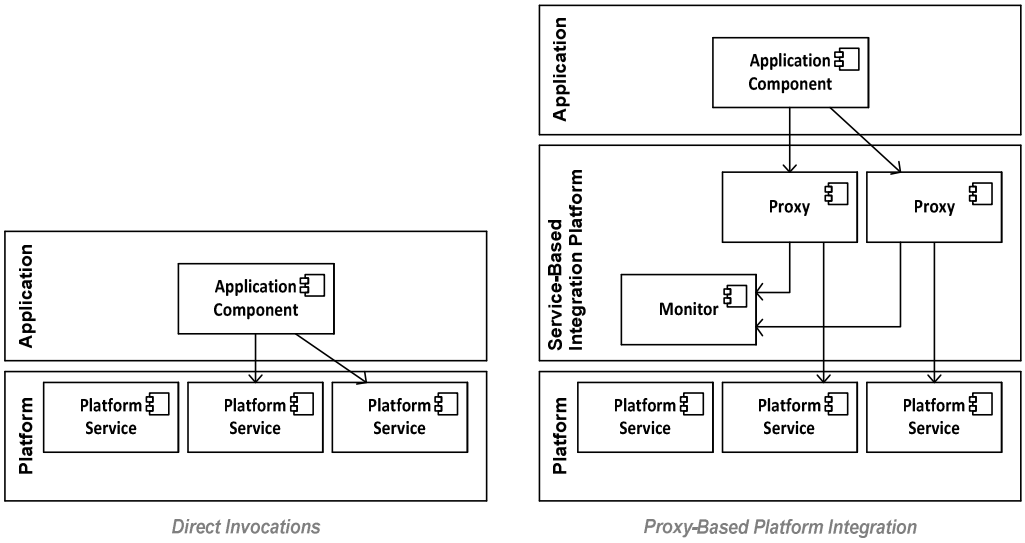


Figure 14: Direct Invocations vs. Proxy-Based Platform Integration

In many cases, applications and platforms are residing in different process or machine contexts. Hence, invocations must cross the process or machine boundary. In such cases, we can apply the remote variant of the PROXY pattern, the REMOTE PROXY [SSRB00a, BHS07a]. In the platform integration context, the REMOTE PROXY resides in the service-based integration platform and connects application and platform. The schematic illustration on the right hand side of Figure 14 also applies to REMOTE PROXIES, but the arrows depict remote invocations instead of local invocations.

In addition to simple integration, service-based platform integration requires coping with the diversity of the interfaces that these platforms expose. Calling a remote interface directly or through a PROXY is not always possible, for instance, because the interfaces offered by a platform may not offer exactly what the calling application expects. Using the original interface might be possible, but we need to take into account that usually the applications are tightly coupled with their interfaces and implementations. Changing the interfaces of a platform is a possible solution. But, firstly, an interface change is tedious and error-prone, and, secondly, most often it is not possible at all because many platforms that need to be integrated are provided by third parties. In addition, platforms are typically used by many applications and it is usually not possible to offer a different interface for each of them. For these reasons, an ADAPTER [GHJV94] can be inserted between the caller and the remote interface that converts the provided interface into the interface that the caller expects and vice versa.

The adapter also transforms the data returned by the adaptee into the data structures expected by the caller. For distributed systems two variants of the ADAPTER pattern, the OBJECT ADAPTER ([GHJV94, BHS07a]) and the INTEGRATION ADAPTER ([HZ12]), can be used to connect the interfaces and to perform the appropriate transformations. From a high-level perspective, OBJECT ADAPTERS usually have a similar structure as the PROXY example depicted in Figure 14. The

ADAPTERS would simply replace the PROXIES and introduce the additional interface adaptation behavior.

Very often new versions of platforms come with new versions of interfaces. This can be hidden from the applications using the interfaces by exchanging the OBJECT ADAPTER. However, the more complex the mapping between the interfaces is, the more expensive is the mapping in terms of performance and development effort.

A general problem of components like OBJECT ADAPTERS in platform integration scenarios is that invocations reaching the ADAPTER while it is being maintained (i.e., stopped and redeployed) would get lost. In many cases, this is highly undesirable. This problem is addressed by an extension of the ADAPTER pattern, the INTEGRATION ADAPTER pattern [HZ12].

An important part of the INTEGRATION ADAPTER pattern is its use of the COMPONENT CONFIGURATOR pattern [KJ04] to stop, suspend, and start the adapter component. This pattern can also be used to make other integration solutions, like the PROXY based solutions discussed before, configurable.

Figure 15 shows a potential INTEGRATION ADAPTER design. The INTEGRATION ADAPTER implements a configurable component interface to realize the COMPONENT CONFIGURATOR pattern. To avoid losing message while the adapter is being maintained, the INTEGRATION ADAPTER has an asynchronous messaging interface to the client, which queues up messages until the maintenance actions are performed. The integrated platform is connected via a synchronous connector. The adapter also performs the translation from asynchronous calls to synchronous calls.

When the service-based integration platform must bridge between different communication protocols, PROTOCOL PLUG-INS [VKZ05] can be used to realize translation between the different protocols.

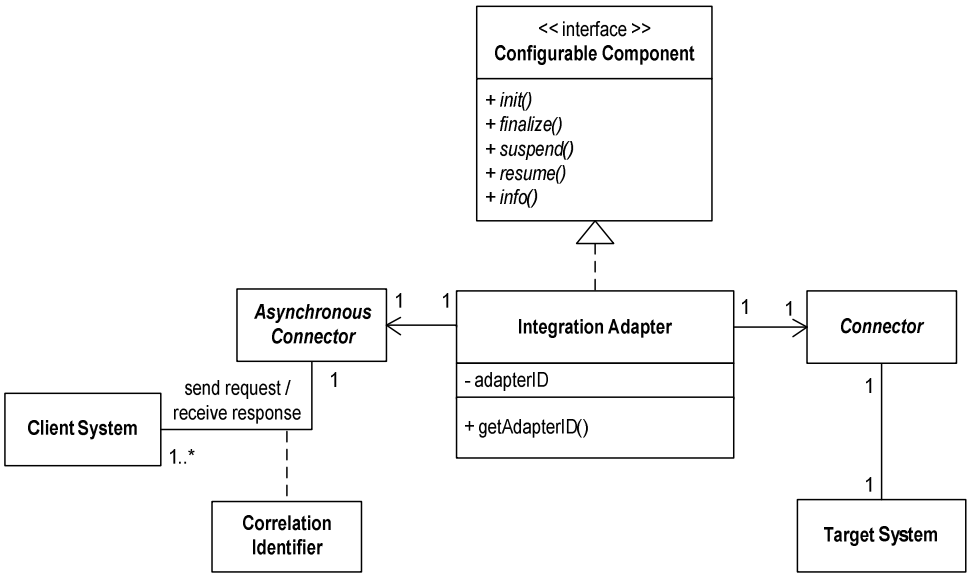


Figure 15: Integration Adapter: example design

6.1.2 Interface Design

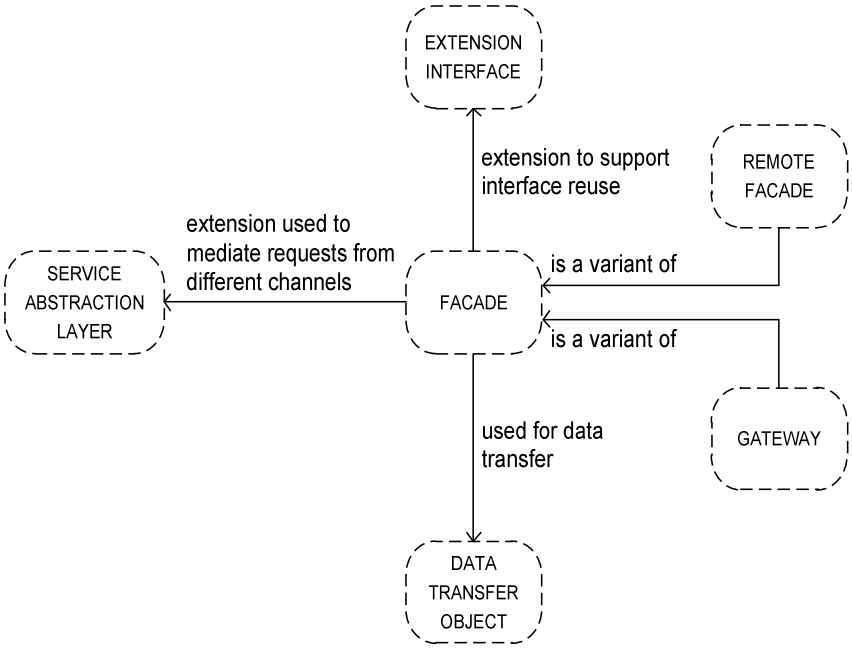


Figure 16: Interface Design

When developing a service-based integration platform, we need to expose interfaces to the application. In the simplest case, we can simply expose the PROXIES and ADAPTERS, as discussed in the previous section. However, often we have additional interface design requirements, such as unification or abstraction of interfaces, supporting different protocols or channels, optimizations of invocation flows, avoiding redundancies in interfaces, or supporting multiple interface versions. We might have the same requirements for one or more of the platforms to be integrated. For instance, many legacy applications do not expose an appropriate service-based interface. Sometimes it makes sense to first make an appropriate service-based interface design for each of the platforms, and then develop a service-based integration platform that offers a unified interface.

When designing interfaces for platforms or integration platforms, the design of the data transfer might be an important concern. Transferring data over the network between two distributed applications can be very expensive when the number of calls increases. Therefore, we can use DATA TRANSFER OBJECTS [Fow03] which hold all the data to be sent. A DATA TRANSFER OBJECT transfers the needed information within a single call. DATA TRANSFER OBJECTS may wrap primitive data types (e.g., integers, strings) or other DATA TRANSFER OBJECTS.

From the viewpoint of the client of a platform, interface unification is often important. Platforms expose multiple interfaces, often in multiple versions. The interfaces exposed by the platforms are often not the interfaces required by the applications using the platforms. The FACADE pattern [GHJV94] describes a general way to unify interfaces. A FACADE [GHJV94] provides a coarse-grained interface on fine-grained components. In distributed systems, a REMOTE FACADE [Fow03] can be used to specify a single point of access for a group of components which provide

complex services in order to mediate client requests to the appropriate components. A REMOTE FACADE can also aggregate features of different components into new and/or higher-level services. It does not contain any domain logic and can use data from DATA TRANSFER OBJECTS. Using bulk accessors for the data ensures that using to the remote interface remains efficient.

A GATEWAY [Fow03] is another variant of FACADE that represents an access point to an external system used by an application. The application thus becomes independent of the specific interfaces of the external system and also of its internal structure. When a platform needs to support consuming and providing remote objects through multiple channels, a SERVICE ABSTRACTION LAYER [Vog01] can be used. It introduces an extra layer which contains all the necessary logic to receive and delegate requests originating from the different channels. To create a SERVICE ABSTRACTION LAYER a FACADE can be used to offer an interface for creating and sending service requests.

Figure 17 shows an example of interface design by implementing a FACADE which uses data from different DATA TRANSFER OBJECTS. The FACADE aggregates functionality from two application components and exposes an interface for integration with the remote platform. In this example, an ADAPTER inserts additional interface adaptation between the FACADE and the remote platform services. By providing a SERVICE ABSTRACTION LAYER, as illustrated in Figure 18, we support multiple remoting technologies through three different channels: a JMS, SOAP, and REST Interface. A FACADE unifies the different channels and exposes a common interface for the remote platform.

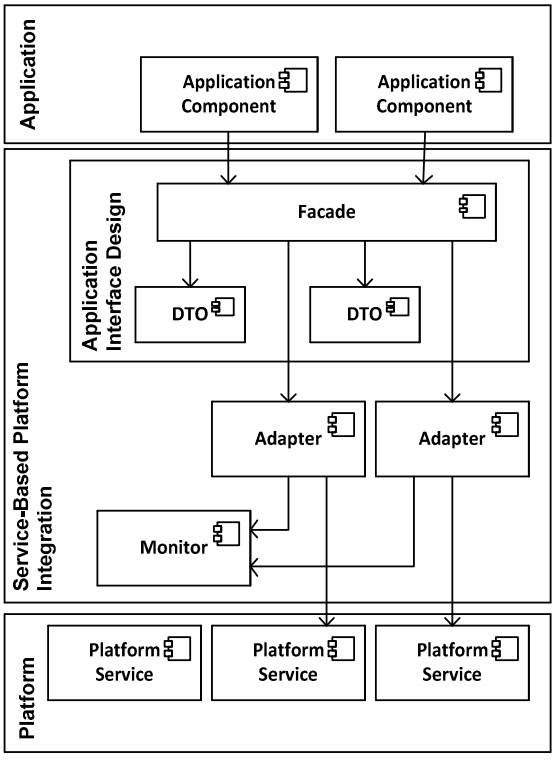


Figure 17: Interface Design with Facade and Integration with Adapter

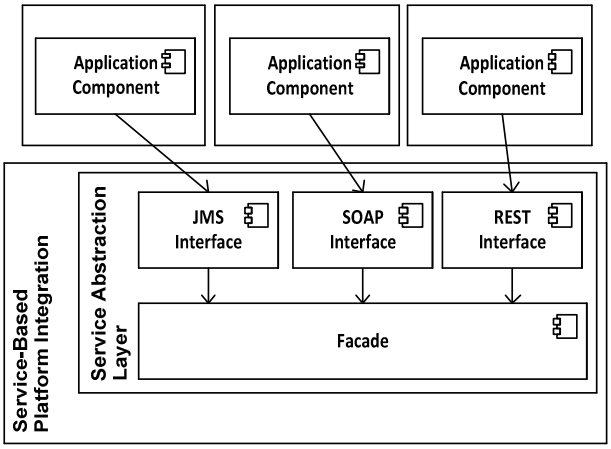


Figure 18: Interface Design with Service Abstraction Layer

Another issue related to the design of interfaces is that the interfaces provided by platform applications are subject to adaptations and/or extensions due to changing requirements. To support different client-specific interfaces, related functionalities can be grouped in separate EXTENSION INTERFACES [SSRB00b] and the common functionality can be included in a root interface.

6.1.3 Communication Style

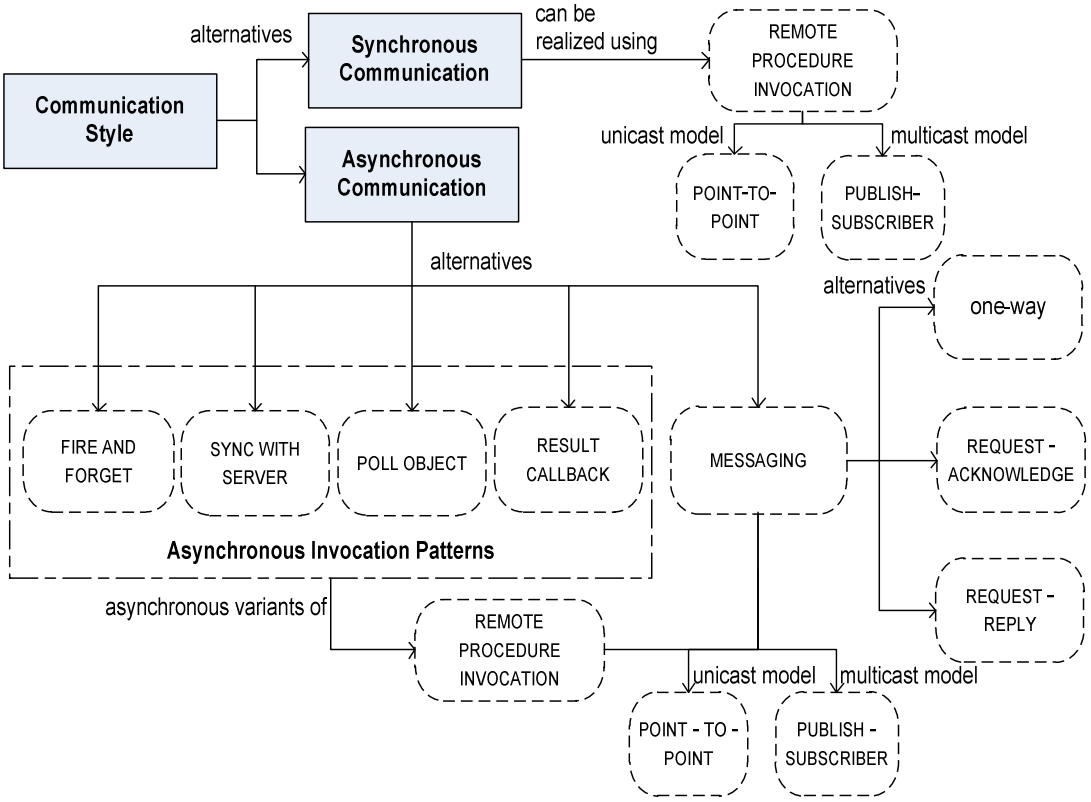


Figure 19: Communication Style

For each connection between two components in the platform integration solution, follow-on decisions about the communication style must be made. For instance, once the design decisions for integration and adaptation, as well as interface design, have been made at the component or service level, at a lower level of abstraction follow-on decisions for the communication style used by the connection between the components must be made. In this section we focus on the different options for connecting distributed components. That is, in the platform integration design space these design decisions are especially relevant for the connections between applications and service-based integration platform, connections between service-based integration platform and the platforms, distributed connections between the platforms, and connections among distributed components within the service-based integration platform.

A basic option is to use synchronous invocations for the connection between two distributed components. Often synchronous invocations are realized following the REMOTE PROCEDURE INVOCATION pattern [HW04]. The remote application may respond either by sending a result value or a void result, unless an execution problem occurs and an exception is sent back. All communication follows the REQUEST- REPLY style [HW04]. In a platform integration solution, this synchronous invocations option will rarely be used because synchronous invocations can lead to slow and unreliable systems, as the communication of the calling application must block until it receives the result. Thus, in the following, we mainly focus on the asynchronous communication style and study the various options for realizing it.

Applications that communicate with each other using asynchronous communication do not need to block their execution, but they can continue with other tasks while they are waiting for the results of their invocations. The asynchronous invocation patterns offer many alternatives of invoking a remote service asynchronously. They describe asynchronous variants of the REMOTE PROCEDURE INVOCATION pattern. In particular, when a result or application error needs to be delivered either a POLL OBJECT [VKZ05] or RESULT CALLBACK [VKZ05] can be used. FIRE AND FORGET [VKZ05] does not return any result or acknowledgement to the application that invokes a remote object, but only offers best effort semantics. When a notification that the request arrived to the remote application is necessary, then SYNC WITH SERVER [VKZ05] can be used instead of FIRE AND FORGET. FIRE AND FORGET offers one-way communication. SYNC WITH SERVER provides communication of type REQUEST- ACKNOWLEDGMENT [HW04]. RESULT CALLBACK and POLL OBJECT offer the REQUEST-REPLY [HW04] communication style. POLL OBJECT can be used with the imperative programming style. In contrast to POLL OBJECT, RESULT CALLBACK requires an event-based programming style to consume the result. It has the benefit over POLL OBJECT to support immediate reaction upon the arrival of a result.

In asynchronous remote invocations, ASYNCHRONOUS COMPLETION TOKENS [SSRB00a] are used to associate the callback with the original invocation. The pattern fulfills the same role as the CORRELATION IDENTIFIER pattern [HW04] discussed below. To ensure reliability of communication and increase decoupling of the integrating platforms, MESSAGING [HW04] provides the most convenient solution.

The integrating applications exchange MESSAGES [HW04] via a MESSAGE CHANNEL [HW04] which can be either a POINT- TO - POINT CHANNEL [HW04] or a PUBLISH - SUBSCRIBE CHANNEL [HW04]. The difference between them is that in the first case we have only one receiver of the requests and in the second case the messages are broadcasted, as there are multiple receivers-subscribers of the messages. The PUBLISH - SUBSCRIBE CHANNEL is the version of the PUBLISH - SUBSCRIBER [BHS07a] pattern that applies for messaging. Apart from messaging the POINT- TO - POINT and PUBLISH - SUBSCRIBER styles can be also used in synchronous or asynchronous remote invocations for unicasting and multicasting respectively. The communication using MESSAGES can be either one-way or two-way. In a one-way communication the sender sends a message to a receiver using a one-way channel, without waiting for any notification or result of its request. A two-way communication requires a two-way channel to allow delivery of responses (void, result values, or exceptions). A REQUEST- REPLY communication can be implemented in different ways combining different asynchronous communication styles. For example, the client can first receive an acknowledgement of its request and then poll for the results (REQUEST-ACKNOWLEDGE - POLL [Dai12]) or get notified about the delivery of its request and receive the request results with a callback service (REQUEST- ACKNOWLEDGE - CALLBACK [Dai12]).

As in synchronous REMOTE PROCEDURE INVOCATIONS or in the asynchronous POLL OBJECT or RESULT CALLBACK patterns, messages are also often used to deliver messages in REQUEST- REPLY style. As in the SYNC WITH SERVER pattern, messages can be delivered in REQUEST- ACKNOWLEDGE style.

6.1.4 Communication Flow

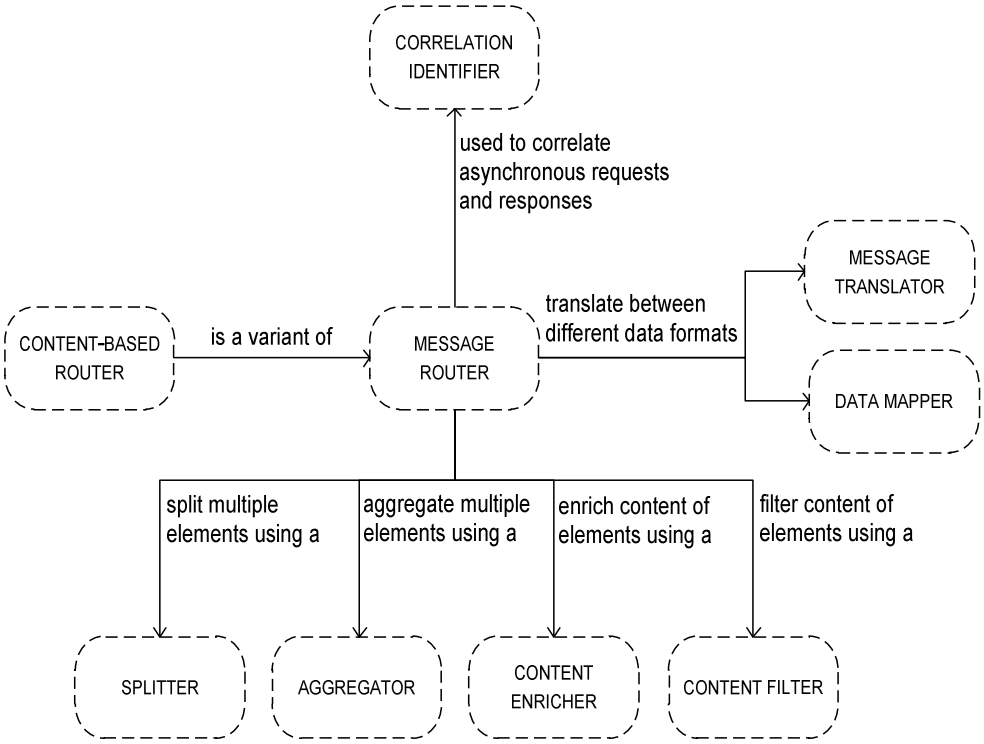


Figure 20: Communication Flow

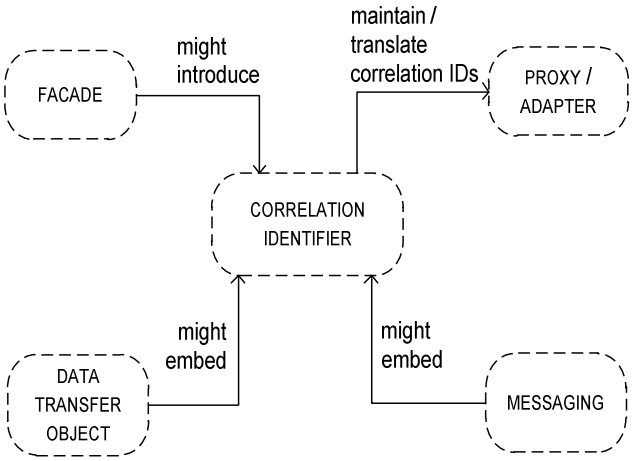


Figure 21: Relationships between CORRELATION IDENTIFIER and other patterns

Transferring distributed service invocation data from the client applications to the integrated platform services, mediated by the service-based integration platform, requires from the software designer to make design decisions related to the data transformations in the service-based integration platform. These decisions touch a variety of concerns, e.g., the routing of the invocations and their invocation data to the intended receivers, as well as all data transformations at different levels (e.g., data representation, marshaling, data transport).

The communication flow perspective considers the flow of requests and replies through the integration platform as a series of data transformations, performed by infrastructure components. The relevant data items are in-memory objects (e.g., DATA TRANSFER OBJECTS) and MESSAGES.

While many patterns described in this section have originally be described in the context of messaging, in variants they can also be applied in combination with the other (asynchronous) invocation patterns. If sophisticated message or invocation routing is required, a MESSAGE ROUTER [HW04] offers an appropriate solution. The MESSAGE ROUTER listens at the incoming, or frontend, message channels and redirects the messages intercepted towards the necessary processing chains and towards the actual backend receivers, i.e., the platform services. With such a central routing component, there is a single point of responsibility for administering the routing rules and to configure the processing chains needed for preparing the messages for the individual platform services. The MESSAGE ROUTER can be made configurable following the COMPONENT CONFIGURATOR pattern (see also Section 6.1.1).

In a service-based integration platform, routing is often performed by a CONTENT-BASED ROUTER [HW04]. As a variant of the MESSAGE ROUTER, this router accesses the message content, i.e., envelope and body elements, to evaluate the standing routing rules against the data extracted from the messages. This way, the routing conditions can be set and transmitted by the MESSAGES themselves (e.g., in their envelopes or by their type annotation), rather than by providing the routing-critical

data through an external source. Content-based routing is not applicable only for the exchange of MESSAGES representing service invocation data (e.g., implicit invocations on domain objects), but it can also be used to differentiate between invocation messages and messages carrying invocation-unrelated or opaque types of data. Imagine application scenarios, which involve setting up audio/video streaming data between client applications and platform services (i.e., here, streaming services). Such data requires alternative processing steps when being mediated by the integration platform; for example as part of an optimization which bypasses routing and processing steps applicable to handshake and invocation messages only. Besides acting as a matchmaker between messages and the available data transformation tasks, a MESSAGE ROUTER also allows for composing processing chains to be applied on selected messages. Message processing and filter components can be organized in a PIPES AND FILTERS [BMR+ 00, AZ05] style. Finally, the processing chains can be constructed in a way so that the delivery to the responsible platform service is performed by republishing the transformed message to a backend or outgoing channel.

The data sent across the network will not always be used by the data receiver, i.e., the platform services, as it is; whatever the dominating communication styles or the communication flow approach is (MESSAGING vs. explicit component invocations). For example, for exposing FACADE interfaces using DATA TRANSFER OBJECTS, the backend invocations must be decomposed into a series of invocations upon one or more platforms and their input and output parameter types. The MESSAGING equivalent to FACADES and DATA TRANSFER OBJECTS are compound messages, with each of the part messages addressing a distinct platform service.

A SPLITTER [HW04] disassembles the compound messages into their constituents which are expected by the target platforms. Sometimes multiple elements need to be collected and reassembled to be delivered to their final destination and to be accepted by the platform services as message endpoints. On the back channel, e.g., for asynchronous REQUEST- REPLY interactions, there is the need for re-assembling the resulting data elements into a composite reply message. This bears the risk of duplicates or an out-of-order reassembly. The SPLITTER can for instance split the messages in the integration platform that are sent to the different platform services. Conversely, an AGGREGATOR [HW04] merges individual messages or element subsets thereof into compound messages to be delivered to the platform services. The AGGREGATOR detects the related elements as well as their right order according to their CORRELATION IDENTIFIERS. On the reply channel, an AGGREGATOR might require a SPLITTER. The AGGREGATOR can for instance aggregate messages in the integration platform that are sent to the different platform services. Apart from this whole-part mismatch between senders and receivers at the level of messages, the data contained in the messages might simply be too excessive or incomplete to be (efficiently) processable by the receivers. There are many possible reasons for this problem. For example, the domain model of the target system might only correspond to a subset of the source domain model. Or certain auxiliary invocation data contained in a message might not be relevant; for instance, the data might only

be required for add-on services or constitute metadata relevant only for the underlying middleware technologies. Sometimes, security requirements demand the removal of message parts (e.g., identity tokens). In such cases, a CONTENT FILTER [HW04] is included in the processing chain of a message to extract and drop excessive data. CONTENT FILTER can be applied in the integrated platform to filter the messages that pass through it. Requirements for additional data can result from domain model mismatches, different underlying middleware, or security requirements. In such cases, a CONTENT ENRICHER [HW04] augments the message with the missing information by accessing external data sources or the message context. CONTENT ENRICHER can be used in the message processing of the integration platform. A frequent source of mismatch between client applications and platform services are incompatibilities between the data formats supported. Such format mismatches involve differences in data models, data types, data representation, and data transport techniques.

When using explicit component invocations and in-memory object representations of the invocation data, a DATA MAPPER [Fow03] can be used to deal with the unaligned or non-canonical data formats between integrating platforms. A DATA MAPPER transforms, e.g., the data from one object type to another. For dealing with marshalling and transport protocol mismatches, the DATA MAPPER can use the services offered by MARSHALLER [VKZ05] and PROTOCOL PLUG-IN [VKZ05] components as offered by the underlying middleware framework.

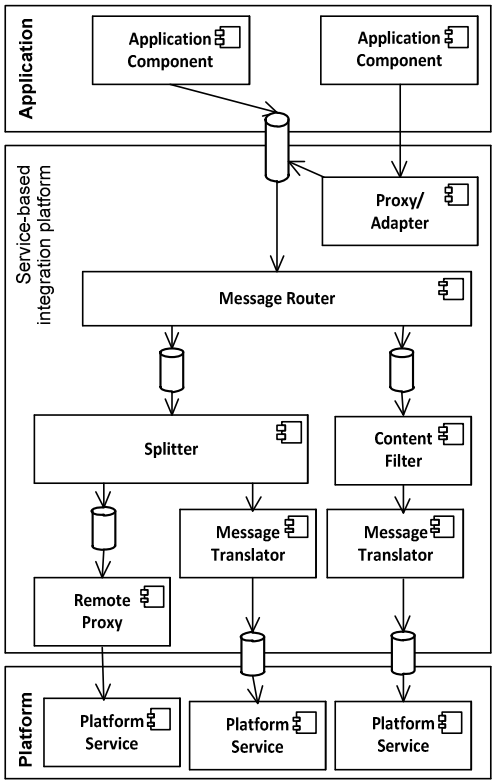
MESSAGE TRANSLATOR [HW04] can be incorporated into the processing chains of MESSAGES for transposing them from one data format into another. In the processing chains, the MESSAGE TRANSLATORS usually come last; as they operate on the already filtered messages.

The MESSAGE TRANSLATOR can for instance reside in the integration platform and translate between the client application message formats and the message formats of the platform services. A particular source of complexity in the communication flow design of a service-based integration platform is the repeated dis- and reassembly of data items; and bridging between process synchronization styles. Both, the content and the synchronization decoupling, require the identification of decoupled parts. Important examples are message parts of disassembled compound messages (see SPLITTER pattern) or non-blocking backend replies to blocking frontend requests. Also, the permanent interleaving of related messages in the integration platform requires a message tracking mechanism.

Adopting CORRELATION IDENTIFIERS [HW04] is an adequate design decision to address such tracking requirements. For asynchronous communication styles, where one has to (implicitly or explicitly) identify exactly a corresponding pair among multiple communication parties, these identifiers are also referred to ASYNCHRONOUS COMPLETION TOKENS [BHS07a]. As for designing the frontend interfaces, for instance, CORRELATION IDENTIFIERS can be employed and stored in the FACADE to track the resulting backend invocations at a per-request level. One option is to maintain the identifier in the service descriptions, such that every

communication with the service needs to refer to a specific CORRELATION IDENTIFIER. Alternatively, a FACADE could also store the CORRELATION IDENTIFIERS in the DATA TRANSFER OBJECTS, if available. In a MESSAGING infrastructure, the CORRELATION IDENTIFIER is extensively used to realize conversational interactions, i.e., for exchanging and processing messages such in REQUEST- REPLY interactions and MESSAGE SEQUENCE interactions [HW04], to name but a few.

In some particular cases, one might need to integrate two or more software platforms that do not support compatible CORRELATION IDENTIFIER mechanisms. The reason can be either one of the platforms does not support CORRELATION IDENTIFIERS or both support CORRELATION IDENTIFIERS but their CORRELATION IDENTIFIERS are not simply interchangeable. In such cases, components, such as the PROXIES or ADAPTERS in this pattern language, are often introduced for mediating the communication and data exchange between these platforms, i.e., translate and temporarily store the CORRELATION IDENTIFIERS. This can be realized, e.g., by letting the mediators maintain an additional table to map the CORRELATION IDENTIFIERS from one communication partner to the CORRELATION IDENTIFIERS of the other communication partner, and vice versa. The design decisions become embodied in the way the service-based integration platform lays out the communication flow in terms of component interactions as depicted in Figure 22. Depending on the decisions taken on the communication styles, there are various possibilities to laying out the data transformation infrastructure in the service-based integration platform. For example, the integration platform can be built using basic MESSAGING principles. Alternatively, an explicit invocation style between transformer components can be applied. Both variants are sketched out as exemplary setups in Figure 22.



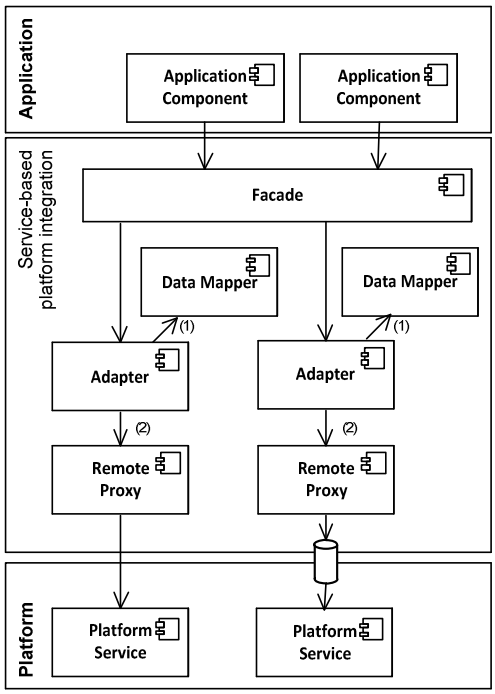


Figure 22: Organizing Communication Flows in a Service-Based Integration Platform

The initial drivers for opting for either approach are the communication styles supported by the components to integrate (i.e., the client applications and the platform services), as well as the decoupling strategies to be implemented by the integration platform. For example, while a straightforward OBJECT ADAPTER can be easily constructed using explicit invocations, an INTEGRATION ADAPTER with an asynchronous frontend connector, which attaches to the client applications can leverage an underlying MESSAGING infrastructure. Both approaches allow for minimizing, or ideally turning obsolete the need for modifying either the client applications and/or platform services to assist in the data transformations required. Client applications or platform services not enabled for MESSAGING can be integrated using bridging PROXY/ADAPTER components, which act as the sending or receiving message endpoints to a frontend and backend channel, respectively. This way, client applications and the platform services do not have to be manipulated even for overcoming such a mismatch in communication style.

6.1.5 Illustration of the pattern language in INDENICA Case Studies

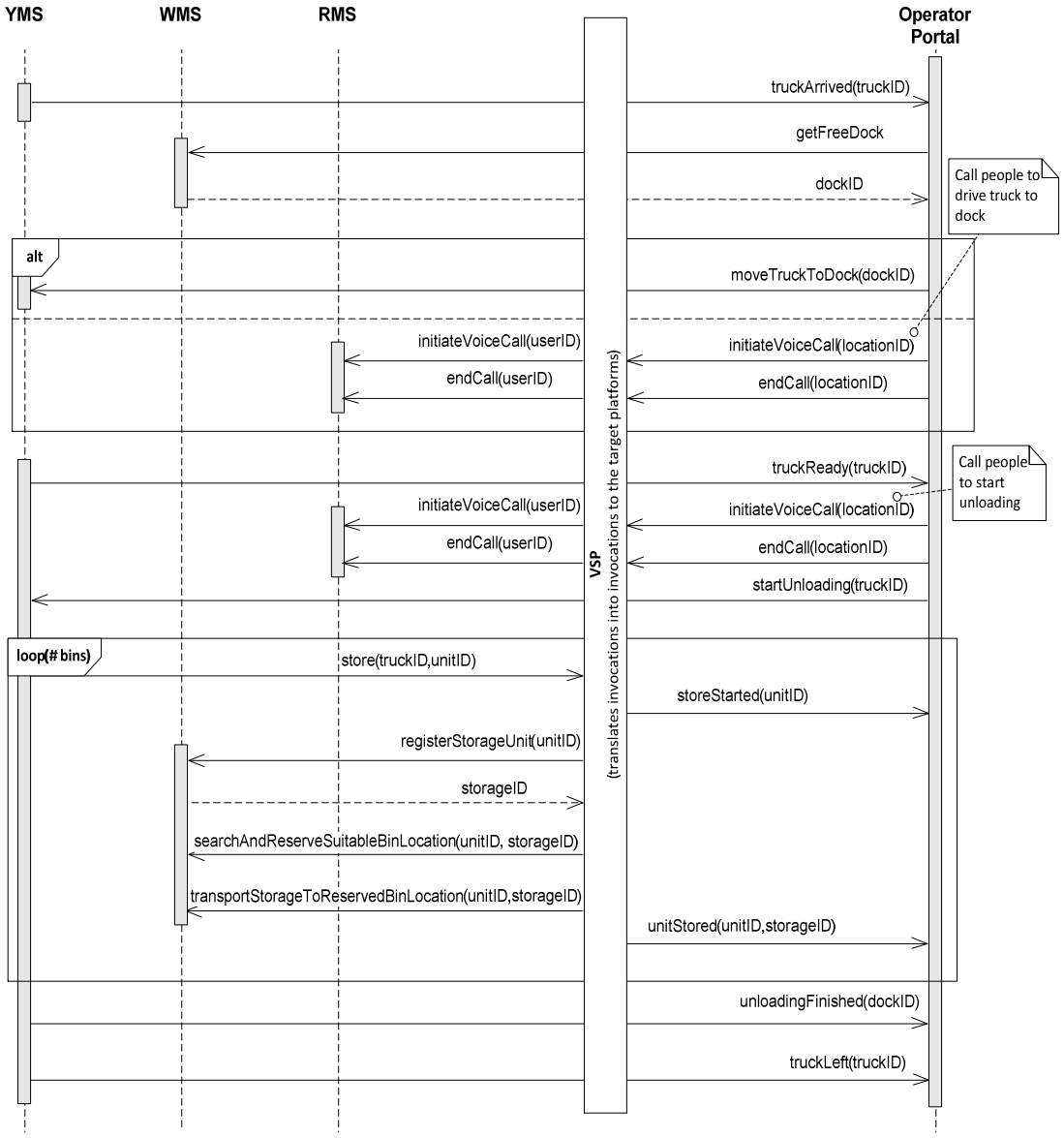


Figure 23: Integration Scenario in the Warehouse

We utilize an excerpt of INDENICA case studies [D5.1] to demonstrate our pattern language for integrating of three platforms: Warehouse Management System (WMS), Yard Management System (YMS), and Remote Maintenance System (RMS). We briefly show in Figure 23 the integration scenario of the case study for unloading storage bins to the racks in the warehouse in terms of a sequence diagram. Further details of the case studies can be found in [D5.1].

To enable the operator application to use the services of the three platforms through an integration platform, many architectural decisions regarding the adaptation and integration of the heterogeneous interfaces as well as the routing of the information between the operator application and the platforms need to be made.

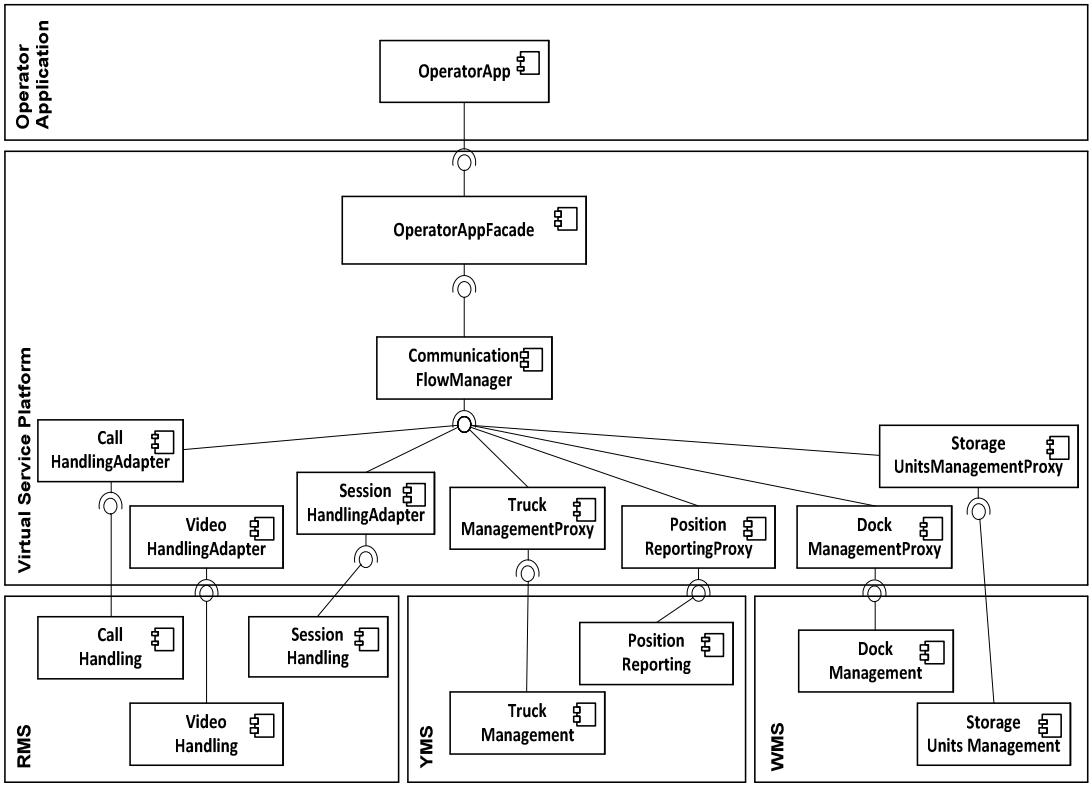


Figure 24: Excerpt of the Integration Architecture

Figure 23 shows how our pattern language is applied for the aforementioned integration scenario. The services introduced by the integration scenario are grouped into components, for example the services *initiateVoiceCall* and *endCall* are grouped into the component *CallHandling*. Figure 24 depicts an excerpt of the architecture of the integration between the three platforms through the VSP that allows the operator application interact with these heterogeneous platforms. A FACADE (*OperatorAppFacade*) provides a common application interface for invoking the different platform services. The component *CommunicationFlowManager* hides the communication flow details between the operator application and the integrating platforms. In order to invoke the remote platform services ADAPTERS and PROXIES are introduced in the integration layer between after the *CommunicationFlowManager* component. In case the access to the remote services does not require any interface adaptations a PROXY is used (e.g. *TruckManagementProxy*, *PositionReportingProxy*, etc.), otherwise an ADAPTER is used in order to resolve interface incompatibilities, i.e. parameter changes (e.g. *CallHandlingAdapter* and *VideoHandlingAdapter*).

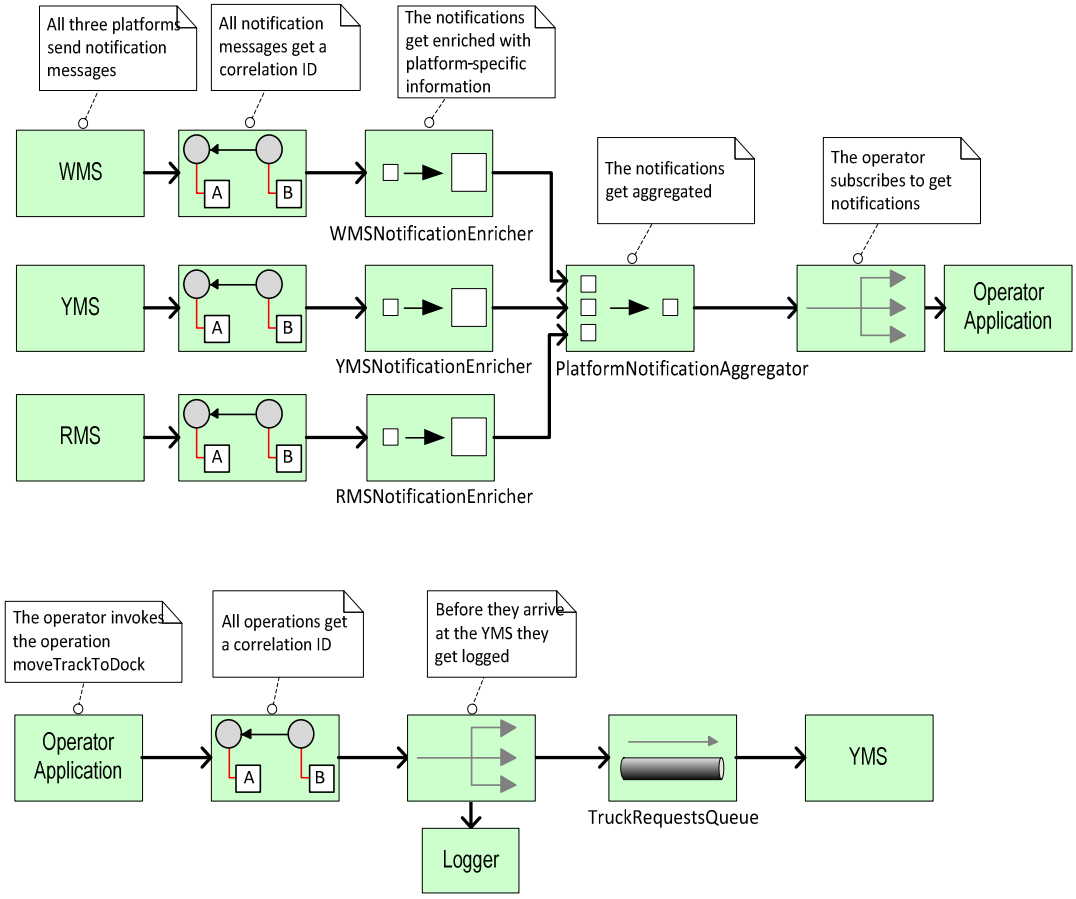


Figure 25: Examples of Communication Flows

Figure 25 demonstrates two examples of communication flow design between the operator application and the three platforms. In the first communication flow diagram the platforms send notifications, which get CORRELATION IDENTIFIERS before they get enriched (*WMSNotificationEnricher*, *YMSNotificationEnricher*, *RMSNotificationEnricher*) with platform details useful for the operator. The single notifications get aggregated into one notification (*PlatformNotificationAggregator*) that is delivered to the operator application which subscribes to the appropriate notification channel (PUBLISH - SUBSCRIBER pattern). In the second communication flow diagram the operator invokes the operation *moveTruckToDock* and the request gets a CORRELATION ID. Afterwards the request gets logged using the PUBLISH - SUBSCRIBER style before it is added to the *TruckRequestsQueue* queue on which the YMS is listening for requests.