A Pattern Language for Service-based Platform Integration and Adaptation

Ioanna Lytra[1], Stefan Sobernig[2], Huy Tran[1], Uwe Zdun[1]

[1]Faculty of Computer Science
University of Vienna, Austria
Email: *firstname.lastname@univie.ac.at*

[2]Institute for IS and New Media
WU Vienna, Austria
Email: *stefan.sobernig@wu.ac.at*

Often software systems accommodate one or more software platforms on top of which various applications are developed and executed. Different application areas, such as enterprise resource planning, mobile devices, telecommunications, and so on, require different and specialized platforms. Many of them offer their services using standardized interface technologies to support integration with the applications built on top of them and with other platforms. The diversity of platform technologies and interfaces, however, renders the integration of multiple platforms challenging. In this paper, we discuss design alternatives for tailoring heterogeneous service platforms by studying high-level and low-level architectural design decisions for integrating and for adapting platforms. We survey and organize existing patterns and design decisions in the literature as a pattern language. With this pattern language, we address the various decision categories and interconnections for the service-based integration and the adaptation of applications developed based on software platforms. We apply this pattern language in an industry case study.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

Additional Key Words and Phrases: service-based platform integration; pattern language; design patterns

1. INTRODUCTION

A software platform is a unified foundation on top of which applications can be developed and executed. Software platforms are an important organizational strategy to achieve software reuse in software development at a large scale. Platform-driven software reuse reaches out beyond systematic code reuse (e.g., through component orientation) and involves reusing use cases, software tests, documented design decisions (e.g., design documents), and development procedures (e.g., continuous application integration, release management, testing approaches; see [Ghanam et al. 2012]). As a software artifact, a software platform abstracts from details inside and underneath the platform and thereby eases application development and maintenance.

Many software systems that are developed today are based on one or more software platforms. For instance, an online shopping portal will rely on an enterprise resource planning (ERP) platform such as SAP R/3 for receiving and processing purchase orders and a warehouse system for managing large inventories of goods. Other examples of software platforms include platforms for steel plant control, telecommunication platforms such as Mobicents[1], data storage platforms such as Amazon's S3[2], social network platforms such as the Facebook Platform[3], and mobile plat-

[1]`http://www.mobicents.org`
[2]`http://aws.amazon.com/s3`
[3]`https://developers.facebook.com`

forms such as Google's Android[4] or Apple's iOS[5]. This short and non-exhaustive list of platform examples illustrates that *different application areas require different and specialized platforms*. Often it is the case that heterogeneous platforms need to be integrated to provide unified services to be used by domain-specific applications. For example, inside a warehouse usually a warehouse management system, an enterprise resource planning, a yard management system and various communication systems need to provide services in an integrated way for building warehouse operator and control applications.

As in many other areas, one of the current trends of software platforms is to offer their services using standardized interface technologies such as Web Services [Christensen et al. 2001]. With this trend comes the growing requirement for the *integration over these specialized service platforms*. If the current trend towards offering platforms through service-based interfaces continues, we will see a large ecosystem of heterogeneous service-based platforms emerging in the near future. However, the increasing specialization in the applications domains of software platforms, as well as the diversity of technologies for platform development and of platform interfaces leads to open challenges regarding the integration of multiple platforms.

In the context of platform integration, the heterogeneity of service platforms with respect to their functional and non-functional properties often leads to several alternative ways of successfully tailoring, adapting, and integrating platforms. In other words, software architects and developers are usually confronted with numerous design decisions at different levels of abstraction and at different levels of granularity when designing a platform integration solution.

This paper aims at addressing challenges regarding offering platforms as services and integrating multiple heterogeneous platforms. In particular, we will address how to offer the functions of a (legacy) platform using a service-based interface and how to tailor as well as to combine multiple platforms into a unified service platform. For this, we introduce a pattern language targeting high-level and low-level design decisions for integrating and adapting platforms using services. This pattern language is addressed to software architects and designers as well as software developers that deal with the problem of designing unified service platforms over heterogeneous platforms –often owned by third-party vendors– to be used as the basis for developing new applications.

So far, a substantial amount of patterns have been described covering many aspects of service-based integration and adaptation. However, those patterns have been presented with a different focus, such as general software design [Gamma et al. 1994], software architecture [Buschmann et al. 2000; Avgeriou and Zdun 2005], distributed system design [Buschmann et al. 2007a], enterprise application architecture [Fowler 2003], messaging [Hohpe and Woolf 2004], remoting middleware [Völter et al. 2005], service-oriented systems [Hentrich and Zdun 2009], service design [Daigneau 2012] and process-driven SOA [Hentrich and Zdun 2012]. The contribution of this paper is to survey and to organize the existing patterns and design decisions in a comprehensive pattern language for the service-based integration and adaptation of platforms. With this, this paper primarily addresses software architects who face the challenge to design, to realize, and to deploy a service-based integration architecture for software platforms. As a secondary audience, developers of client applications, which are built from the integrated platforms, can consult this pattern language to evaluate the impact of the underlying integration architecture (and the design decisions embodied therein) on the observed non-functional properties (e.g., QoS properties such as execution timings, distributed exception state) observed for their applications.

The remainder of the paper is structured as follows. Section 2 provides the problem statement and the background on service-based platform integration. In Section 3, we illustrate an industry case study as a motivating example. The actual pattern language is then presented in detail in Section 4. To demonstrate its applicability, the pattern language is applied to our motivating example in Section 5. We discuss the related work in Section 6 and conclude in Section 7.

[4] http://www.android.com
[5] http://www.apple.com/ios

2. SERVICE-BASED PLATFORM INTEGRATION: BACKGROUND AND PROBLEM STATEMENT

We consider a *software platform* as a collection of software sub-systems, like communication middleware and databases, and interfaces which together form a reusable infrastructure for developing a set of related software applications. The functional and non-functional properties of a service platform and its interfaces vary with the requirements of the application area in which the platform is deployed. To build a concrete application by reusing software artifacts in a platform, the platform lays out a customization and configuration process on top of its interfaces [Ghanam et al. 2012].

In a *service-based* software platform, software platforms expose their interfaces in terms of services which provide the programming models for developing platform-based applications. Platform customization and configuration usually involves adaptations of service interfaces (e.g., interface aggregation) and/or service implementations (e.g., service specialization and substitution) as well as forms of service composition (e.g., batched service execution, service chaining, or process-driven service orchestration). Platforms are then integrated by applications via their exported *platform services*.

When looking at multiple service-based platforms, the exported services and their interfaces are heterogeneous in terms of the middleware technologies, the transport protocols, the programming languages, and the programming models (e.g., remote procedure calls, document-centric services) used. In addition, platform service interfaces change over time and platform services are substituted for others (e.g., service specialization, service substitution [Ruokonen et al. 2008]). Applications using platforms must cope with the heterogeneity of the platforms they integrate, as well as with interface changes. At the same time, (groups of) applications exhibit different requirements on the same set of platform services; and may change in these requirements over time. For instance, applications might require functionally tailored interfaces (e.g., operation subsets, aggregated operations and aggregated operation data) and different interface capabilities for separate application groups based on their QoS requirements or on different authorization levels.

If this platform heterogeneity and the characteristic integration requirements were fully anticipated (for example, by analyzing the platform domains using a domain engineering approach), software platforms would be developed in terms of software product lines [Ghanam et al. 2012] and platform integration would turn into an issue of developing *multi software product lines* [Rosenmüller and Siegmund 2010]. In this paper, however, we are interested in integration scenarios involving software platforms which were not necessarily designed as product lines and, most importantly, which were not designed to be integrated with one another (e.g., by using product line engineering techniques such as code generation or component weaving).

A strategy to deal with previously *unanticipated* platform integration is *service-based platform integration*. In such an integration strategy, applications should strive for programming towards stable, service-based interfaces that hide technology, protocol, language, and programming model dependencies as much as possible. Some platforms already offer a suitable service-based platform integration interface to their platform-driven applications, but in most cases such interfaces are not available (consider, e.g., legacy platforms). Service-based platform integration addresses situations like the one depicted on the left-hand side of Figure 1: The client applications developed on top of a software platform have direct dependencies to the services offered by the platform. Service-based platform integration changes this situation then to the one depicted on the right-hand side of the figure, with an intermediate abstraction layer hiding the details of the platform underneath. Thus, an application is developed on top of a service-based integration platform which integrates services from one or more platforms. With such an integration platform in place, client applications with changing requirements on the platform services, on the one hand, and platforms exposing changing platform services, on the other hand, can be effectively shielded from each other.

An intermediate platform has an additional benefit, illustrated in Figure 1: Applications from different domains can be programmed against different service-based integration platforms, each offering a domain-specific view on the platform abstractions. For instance, consider developing applications for Android. The integration platform could provide different views on the Android platform, with each view exposing selected Android API chunks for, e.g.,

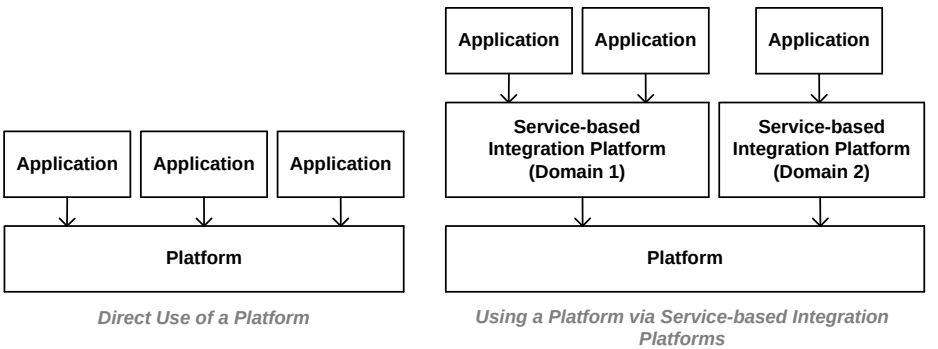

Fig. 1: Service-based Platform Integration

business applications, Web applications, 2D and 3D games, and so on. The integration platform could provide a stable view over different versions of the Android SDK.

Such a design also offers the advantage of rendering the underlying platforms exchangeable. For example, an abstraction layer can be provided in a way that similar script code can run both on Android and iOS. In this example, this is possible provided that a scripting engine running on both platforms is used.

In addition to the abstraction of platforms and their interfaces, this paper also considers the problem of integrating multiple platforms and their platform services into one and the same application. This problem is schematically illustrated in Figure 2. Consider an online shopping portal that relies on an enterprise resource planning (ERP) platform such as SAP R/3 for receiving and processing purchase orders and a warehouse system for managing large inventories of goods. In a service-based integration platform, we could select only those services of the two platforms that are relevant to the online shopping portal and present them in an integrated fashion. We could offer them through the different communication channels needed for the online shopping portal, but not offered by the legacy platforms. Internally, the service-based integration platform would perform all necessary tasks of integrating, adapting, and routing messages for the two backend platforms.

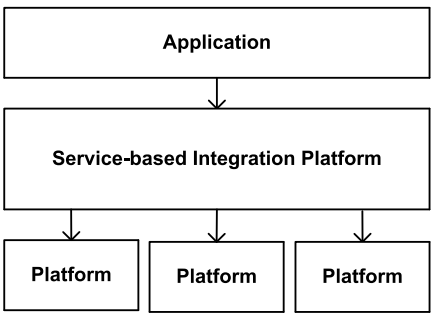

Fig. 2: Service-based Platform Integration for Multiple Platforms

3. MOTIVATING EXAMPLE

To illustrate the problem of service-based platform integration and adaptation, we present an example from a case study on industry automation performed in the context of the EU FP7 project INDENICA[6]. In this case study, there are three heterogeneous platforms: a Warehouse Management System (WMS), a Yard Management System (YMS), and a Remote Maintenance System (RMS). An operator application utilizes the services provided by these three platforms. The YMS manages the scheduling and the coordination procedures for trucks which are needed for the loading and unloading of the goods. The WMS handles the storage of the goods (or storage bins) into racks via conveyor systems. The RMS system is connected to the warehouse to monitor every incident occurring in the warehouse and the yard. An operator application uses the services exposed by the domain-specific integrating virtual platform (VSP). This intermediate platform integrates the services of the three backend platforms. All the interactions between the integrated platforms as well as between the operator application and the three platforms are performed through the VSP.

Figure 3 illustrates a single scenario covered by our case study, presented as a sequence diagram. This scenario addresses unloading storage bins onto the racks in the warehouse. When a loaded truck arrives at the yard, the operator gets notified (*truckArrived*) and requests a free dock (*getFreeDock*) for the truck. After receiving a free dock (a *dockID*), the operator communicates with the personal (*initiateVoiceCall* and *endCall*) and coordinates the redirection of the truck to the assigned dock (*moveTruckToDock*) for its unloading. When the operator gets notified that the truck is ready for unloading (*truckReady*), she sends a command to the personal to start unloading (*startUnloading*). For each product a storage unit is registered (*registerStorageUnit*) and a suitable bin location is reserved (*searchAndReserveSuitableBinLocation*). The product is stored in the reserved bin location (*transportStorageToReservedBinLocation*). Upon finishing the unloading, a notification (*unloadingFinished*) is sent to the operator. Finally, the operator gets notified when the truck leaves the dock (*truckLeft*).

To enable the operator application to use the services of the three platforms through an integration platform, many architectural decisions regarding the adaptation and the integration of the heterogeneous interfaces as well as the routing of the information between the operator application and the platforms must be made. In this paper, we study the design alternatives that must be considered when software architects and developers are confronted with the platform integration problem. In the next section, we describe the pattern language for service-based platform integration and adaptation in detail.

4. PATTERN LANGUAGE FOR SERVICE-BASED PLATFORM INTEGRATION AND ADAPTATION

4.1 Pattern Language Overview

In this section, we describe a pattern language for service-based platform integration and adaptation. This pattern language documents interconnected design decisions by drawing from existing pattern material, such as patterns for general software design [Gamma et al. 1994], software architecture [Buschmann et al. 2000; Avgeriou and Zdun 2005], distributed system design [Buschmann et al. 2007a], enterprise application architecture [Fowler 2003], messaging [Hohpe and Woolf 2004], remoting middleware [Völter et al. 2005], service-oriented systems [Hentrich and Zdun 2009], service design [Daigneau 2012], and process-driven SOA [Hentrich and Zdun 2012].

We introduce an overview of the main categories of design decisions documented in our pattern language in Figure 4. The direction of the arrows implies follow-on decisions. Arrows in both directions imply that the decisions can be made in parallel. In our pattern language, we consider the following architectural decision categories: *Integration and Adaptation*, *Interface Design*, *Communication Style* and *Communication Flow*.

The *Integration and Adaptation* category collects design decisions regarding the integration of platform services into a service-based integration platform and their interface and protocol adaptation, if required. The *Interface Design* category mainly covers design decisions regarding the design of the exported interface(s) of the service-based integration platform. Decisions in the categories *Integration and Adaptation* can be performed in parallel to decisions in

[6] http://www.indenica.eu

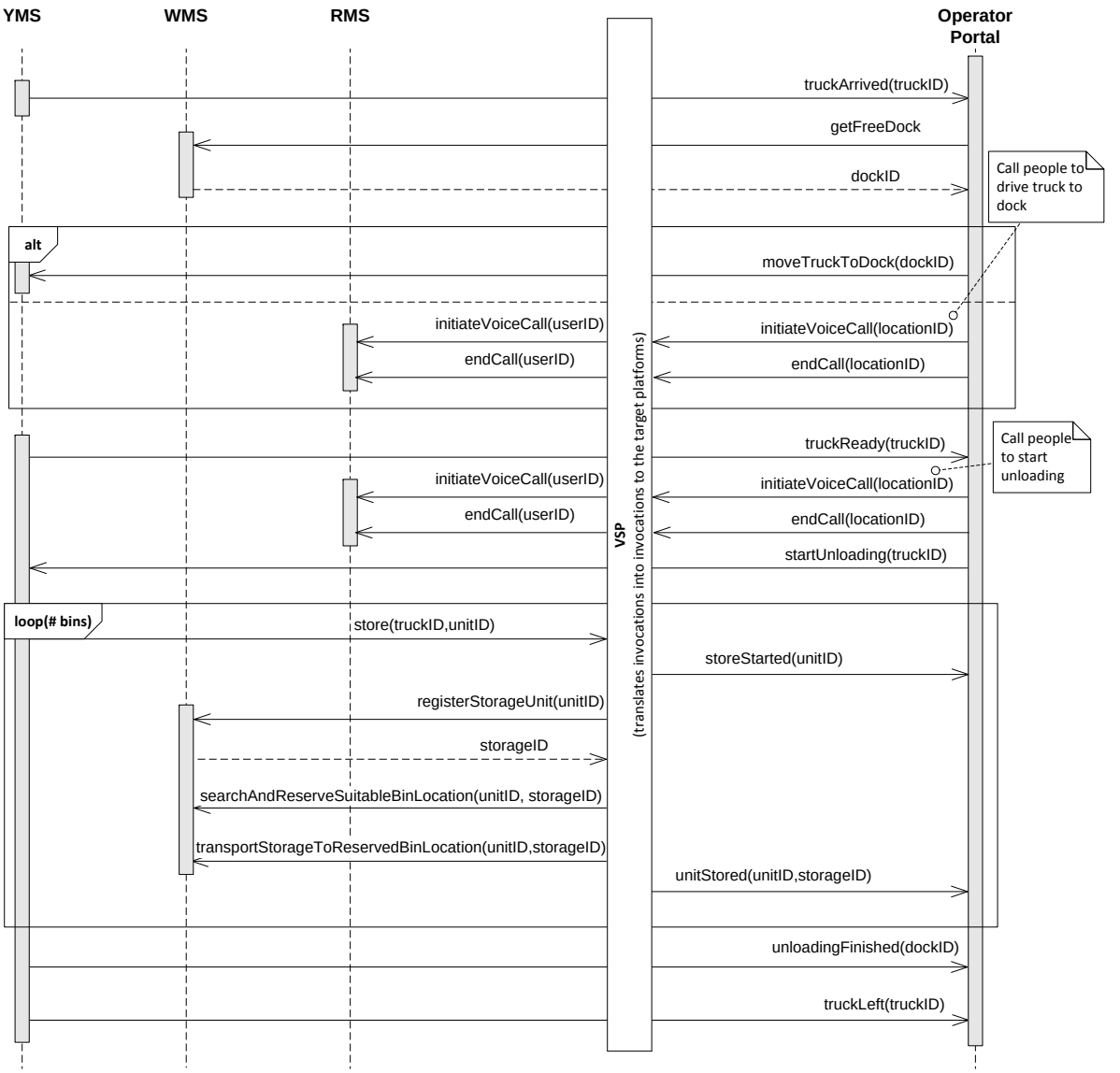

Fig. 3: Integration Scenario in a Warehouse

the *Interface Design* category. These categories mainly concern developing components and interfaces for connecting applications, platforms, and the service-based integration platform.

The *Communication Style* category comprises design decisions that must be taken for each distributed component connection. These decisions relate to options for connecting two components (e.g., blocking and non-blocking component interaction styles). These decisions reside at a lower level of abstraction than the decisions of the two previous categories.

The *Communication Flow* category describes additional decisions that must be considered in case the service-based integration platform introduces more complex communication flows than simple forwarding from an exported interface

to imported interfaces. For instance, such decisions relate to handling the aggregation or the splitting of the messages on their way through a service-based integration platform.

The patterns in each category are documented in the form *Problem – Solution – Decision Drivers*. These pattern sketches are based on the information extracted from the aforementioned design pattern material. The decision drivers reported have been selected based on the authors' assessment of their relevance for service-based platform integration.

In the following sections, we discuss the four pattern-language categories in more detail and provide visualizations of the characteristic decision flows in each category. The decision flows document relevant patterns and their interconnections, as well as the follow-on pattern category or categories to be considered. The decision flows and the related pattern sketches will help software architects and developers to structure their own decision-making processes by highlighting characteristic decision steps and by presenting pattern alternatives, pattern variants and pattern compounds. With this, we aim at offering a guideline on how to lay out a concrete decision-making process rather than presenting a ready-made "recipe" for selecting patterns in service-based platform integration.

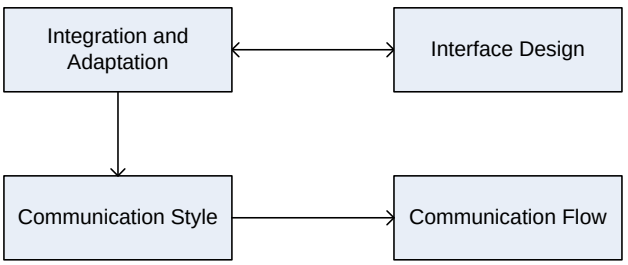

Fig. 4: Overview of Pattern language for Platform Integration

4.2 Integration and Adaptation

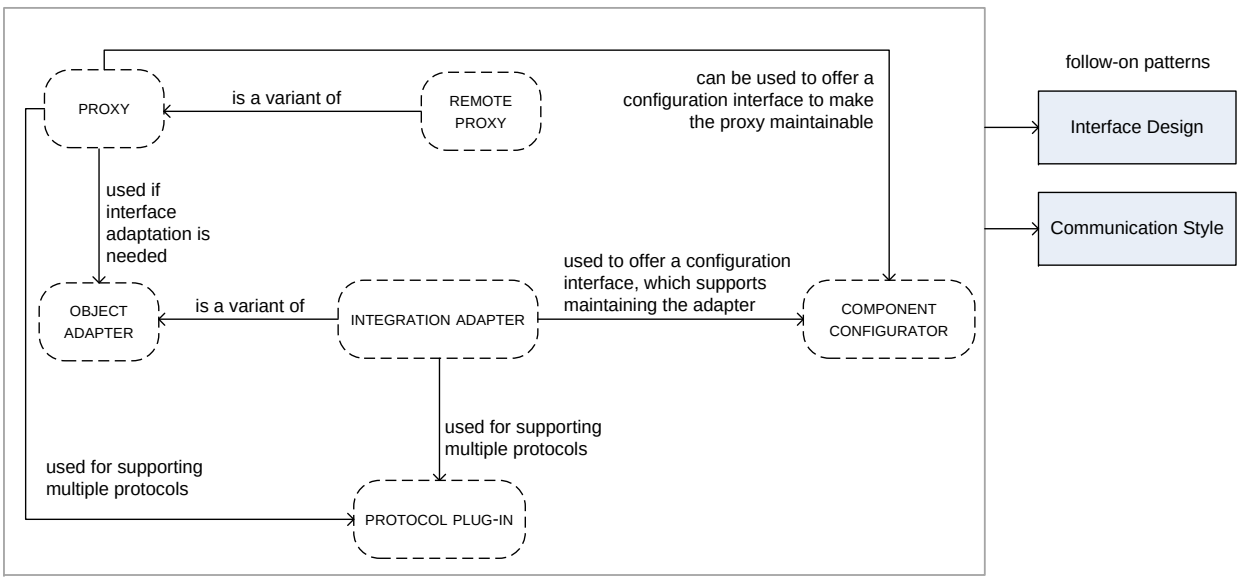

Fig. 5: Platform Integration and Adaptation Patterns

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 as motivated in Section 2. 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. Such situations are discussed in this section.

For the case that the interfaces offered by the platform are compatible to each other and that the extra functionality needed does not change the invocation flow, the PROXY pattern [Gamma et al. 1994] can be used to indirect the service invocations, to perform additional tasks on the invocation data, to select and to access the actual platform services. If extra functionality such as logging, monitoring or access control is needed and this does not change the invocation flow, the functionality can be handled using a PROXY between the platform services and the application.

PROXY [Gamma et al. 1994]
Problem There are situations in which a client does not or can not access a platform service directly, but wants still to interact with it. A surrogate or placeholder for an object to control the access to the service is needed.
Solution A PROXY acts as the intermediary between the client and the target. The PROXY has the same interface as the target. The PROXY holds a reference to the target and can forward requests to the target.
Decision Drivers A PROXY is used whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. It introduces a level of indirection when accessing an object. It also supports creation of objects on demand. In the context of platform integration, it can be used to introduce extra functionality or control, but it does not change the interface of the invoked service or the invocation flow.

Direct invocations vs. proxy-based platform integration are illustrated in Figure 6. When using direct invocations the platform services are called directly from the application, thus the application is tightly coupled with the platform interfaces. In the second schematic example, the PROXIES introduce extra functionality for monitoring the invocation flow from the application to the platform. From the viewpoint of the application, they essentially introduce a new platform abstraction, in this paper called the *service-based integration platform*.

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 [Schmidt et al. 2000a; Buschmann et al. 2007a]. 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 6 also applies to REMOTE PROXIES, but the arrows depict remote invocations instead of local invocations.

REMOTE PROXY [Schmidt et al. 2000a; Buschmann et al. 2007a]
Problem As in the PROXY pattern, we need to access an object through a placeholder for another object to control access to it. In addition, the object and its client are residing in different process or machine contexts.
Solution A REMOTE PROXY is a PROXY that connects two objects in different process or machine contexts. Usually, communication middleware is used to cross the process or machine boundary.
Decision Drivers The REMOTE PROXY has the same decision drivers as the PROXY pattern plus the need for integration of distributed applications and platforms.

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.

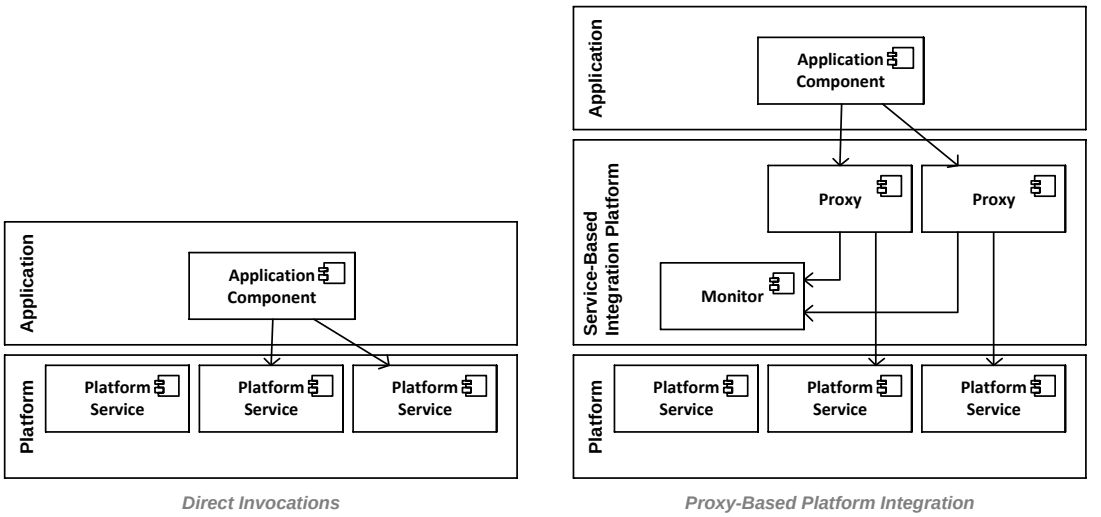

Fig. 6: Direct Invocations vs. Proxy-based Platform Integration

For these reasons, an ADAPTER [Gamma et al. 1994] 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 [Gamma et al. 1994; Buschmann et al. 2007a] and the INTEGRATION ADAPTER [Hentrich and Zdun 2012], can be used to connect the interfaces and to perform the appropriate transformations.

OBJECT ADAPTER [Gamma et al. 1994; Buschmann et al. 2007a]
Problem A class or component offers an interface, but the interface does not match the one that is needed by a client. We need to resolve the interface incompatibility.
Solution An OBJECT ADAPTER converts the interface of a class or component into another interface clients expect. The ADAPTER lets classes or components work together that could not otherwise because of incompatible interfaces.
Decision Drivers Interface adaptation lets us incorporate our classes or components into existing systems that might expect different interfaces. The amount of work for creating an ADAPTER depends on the similarity between the adaptee and target interfaces.

From a high-level perspective, OBJECT ADAPTERS usually have a similar structure as the PROXY example depicted in Figure 6. 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 [Hentrich and Zdun 2012] pattern.

An important part of the INTEGRATION ADAPTER pattern is its use of the COMPONENT CONFIGURATOR pattern [Schmidt et al. 2000c] to stop, suspend, and start the adapter component during the process lifetime of the integration platform. This pattern can also be used to make other integration solutions, like the PROXY based solutions discussed before, configurable at runtime. This form of runtime adaptability complements other configuration techniques available at the deployment time (e.g., deployment descriptors) and at the runtime of the integration platform (e.g., invocation interceptors for the middleware framework).

INTEGRATION ADAPTER [Hentrich and Zdun 2012]
Problem Heterogeneous systems need to be connected and we need to shield the client from the impact of system and system interface changes. The calling and called interfaces might change over time and maintenance activities should not cause invocations or messages to get lost.
Solution The INTEGRATION ADAPTER contains two connectors: one for the client system's import interface and one for the target system's export interface. It plays the role of the translator between the heterogeneous systems and for their different interfaces, protocols, technologies and synchronization mechanisms. The adapter can be made configurable at runtime by using the COMPONENT CONFIGURATOR pattern, so that the adapter can be modified without affecting the requests to the adapter. A COMPONENT CONFIGURATOR offers a configuration interface for stopping, suspending and starting adapters. When new versions of the adapter must be deployed the adapter is stopped. When new versions of the target system are deployed or the adapter is configured at runtime the adapter is suspended. After the maintenance activities the adapter can process all requests that have arrived in the meantime.
Decision Drivers The INTEGRATION ADAPTER provides flexible integration for applications from external vendors. Generic adapters can be offered to support interconnectivity via common standards (e.g., Web Services). A drawback of INTEGRATION ADAPTERS is that if many adapters from different systems exist, they need to be managed in a centralized and controlled way.

We illustrate in Figure 7 a potential INTEGRATION ADAPTER design. The INTEGRATION ADAPTER implements a configurable component interface to realize the COMPONENT CONFIGURATOR pattern. To avoid losing messages 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 (see the discussion of MESSAGING in Section 4.4). The integrated platform is connected via a synchronous connector. The adapter also performs the translation from asynchronous calls to synchronous calls (see the discussion of CORRELATION IDENTIFIER in Section 4.5).

COMPONENT CONFIGURATOR [Schmidt et al. 2000c]
Problem The application must provide a mechanism to configure components at any time of the application lifecycle. The components should be initiated, suspended, resumed, terminated or exchanged dynamically at runtime without having any impact on the rest of the application.
Solution The component interfaces are decoupled from their implementation and used from the application to dynamically control the components. Concrete components implement these interfaces in an application-specific manner.
Decision Drivers COMPONENT CONFIGURATOR offers a common interface for the administration of components (initialize, suspend, resume and terminate). The implementation of the components is decoupled from their configuration, thus increasing modularity and reuse. Configuration and reconfiguration of components can be performed dynamically. This pattern increases also the range of configuration alternatives. However, it has the liability of a lack of determinism, since the behavior of an application is not determined until its components are configured at runtime and a potentially lowered reliability, since the dynamically configured components can affect the execution of other components. Also, the dynamic linking adds extra levels of indirection to invocations.

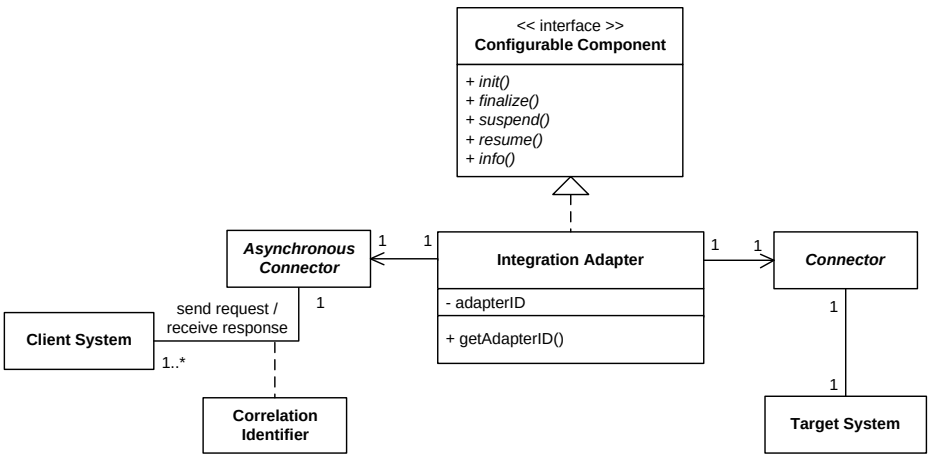

Fig. 7: Integration Adapter: Example Design

When the service-based integration platform must bridge between different communication protocols, PROTOCOL PLUG-INS [Völter et al. 2005] can be used to realize translation between the different protocols.

PROTOCOL PLUG-IN [Völter et al. 2005]
Problem A distributed application must support multiple communication protocols at the same time. The protocols used for the scope of, e.g., single requests or different clients should be configurable at runtime.
Solution PROTOCOL PLUG-INS contain implementation details at the communication-protocol layer and provide common interfaces for different communication protocols. For the configuration of their parameters, the PROTOCOL PLUG-INS offer either an API or a configuration file.
Decision Drivers PROTOCOL PLUG-INS abstract from communication protocol details and allow flexible support for several communication protocols. They can also allow configuration and optimization of the communication protocols used.

4.3 Interface Design

When developing a service-based integration platform, we need to expose interfaces to the application. Through these interfaces the applications built on top of the integration platform will be able to invoke the remote platform services. Common interfaces are also needed for monitoring, adaptation etc. In the simplest case, we can simply expose the PROXIES and ADAPTERS, as discussed in the previous section. However, often we are faced with additional interface design requirements, such as the unification of or the abstraction from interfaces, supporting different protocols or channels, optimizing 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 is more appropriate to first craft an appropriate service-based interface design for each of the platforms, and then to 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 [Fowler 2003] which hold all the data to

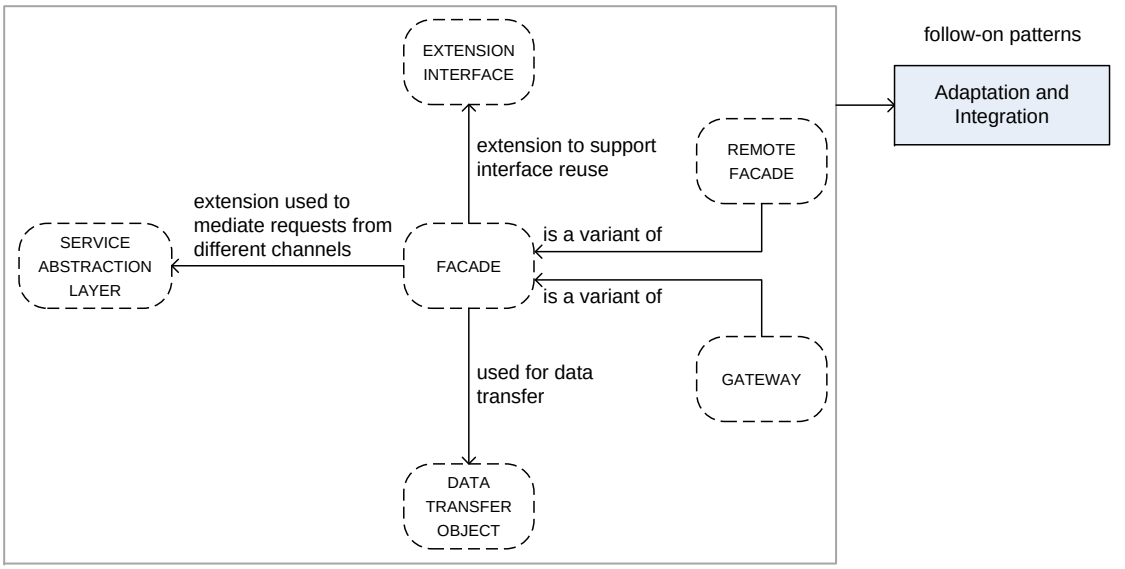

Fig. 8: Interface Design

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.

DATA TRANSFER OBJECT [Fowler 2003]
Problem When a remote interface is designed akin to a local interface, often many invocations with small sets of data are sent. This can get very expensive in terms of bandwidth use and processing of calls. Also, it leads to cluttered and unstable interfaces.
Solution DATA TRANSFER OBJECTS can be used to group the data that needs to be sent in one object. Often multiple invocations transmitting only small sets of data can be replaced by a single invocation transmitting the DATA TRANSFER OBJECT.
Decision Drivers The use of DATA TRANSFER OBJECTS makes interfaces more stable, as for many changes only the DATA TRANSFER OBJECT needs to be changed and the interface can remain the same. DATA TRANSFER OBJECTS can lead to more efficient use of bandwidth and to a reduced overhead incurred by invocation processing. DATA TRANSFER OBJECTS need to be serialized before being sent. The choice of the serialization form is critical for the performance of the transmission. Using textual instead of binary data consumes more bandwidth and can potentially lead to a significant performance penalty. Using binary serialization can introduce fragility into the communication lines. In contrast, XML serialization can be more tolerant towards changes.

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 [Gamma et al. 1994] describes a general way to unify interfaces. A FACADE [Gamma et al. 1994] provides a coarse-grained interface on fine-grained components. In distributed systems, a REMOTE FACADE [Fowler 2003] 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 invoking on the remote interface remains efficient.

FACADE [Gamma et al. 1994]
Problem Clients require stable interfaces over multiple versions of a component and multiple, heterogeneous components. Clients might require client-specific views on a component exposed as a stable interface.
Solution A FACADE is an object or component that provides a unified and often simplified interface for a set of software components. The FACADE or FACADES of a system are usually not bypassed by clients.
Decision Drivers FACADE can be applied when multiple interfaces should be unified. Interface unification, however, comes at a price. Often only the common denominators of interfaces can be offered (e.g., in case multiple versions of an interface need to be unified in a FACADE), or unification of overlapping functionality can be difficult to reach. Introducing an additional FACADE means to introduce an indirection that on the one hand costs performance, but on the other hand provides an additional point of control in the message flow.

REMOTE FACADE [Fowler 2003]
Problem Interaction between objects is better understood when small objects have small methods, which leads to a fine-grained behavior. However, using fine-grained interactions when making calls between process or machine boundaries can be very expensive in terms of performance. Any object that is intended to be used as a remote object needs a coarse-grained interface to minimize the number of calls needed for a process.
Solution A REMOTE FACADE translates the coarse-grained methods onto the underlying fine-grained objects. Thus, it separates distinct responsibilities into different objects. A bulk accessor is used to replace a number of getters and setters of the underlying objects with one getter and setter.
Decision Drivers REMOTE FACADE provides access to a fine-grained object with a coarse-grained interface. Using a coarse-grained object model improves performance because of the reduced number of calls. It adds, however, additional programming effort, as the remote calls have to be translated into smaller internal calls.

A GATEWAY [Fowler 2003] is another variant of FACADE that represents an access point to an external system used by an application. The application thus becomes independent from the specific interfaces of the external system and also from its internal structure.

GATEWAY [Fowler 2003]
Problem Complex interfaces lead to complicated applications. When there is a need to call an external API that is difficult to understand and use, this complexity is spread through the whole system.
Solution A GATEWAY is a wrapper that translates a specialized and complicated API into a simpler API. All applications that need to call this API call instead the API offered by the GATEWAY.
Decision Drivers The introduction of a GATEWAY makes a system easier to test and any possible changes in resources flexible. When the source API changes only the GATEWAY component needs to be modified. When implementing a GATEWAY an issue that has to be considered is the handling of exceptions and return values from the source API.

When a platform needs to support consuming and providing remote objects through multiple channels, a SERVICE ABSTRACTION LAYER [Vogel 2001] 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 for sending service requests. We show in Figure 9 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 10, we support multiple remoting technologies through three different channels: a JMS, a SOAP, and a REST Interface. A FACADE unifies the different channels and exposes a common interface for the remote platform.

SERVICE ABSTRACTION LAYER [Vogel 2001]
Problem A system or platform must allow for providing and consuming remote objects through multiple channels, i.e., remoting technologies and transport protocols. This channel support should be independent from the core invocation handling for remote objects. New channels should be addable on demand.
Solution The SERVICE ABSTRACTION LAYER adds an extra layer which receives and mediates requests originating from different channels. Each channel contains a channel adapter which translates requests back and forth between the backend and frontend channel formats.
Decision Drivers SERVICE ABSTRACTION LAYER separates business from communication logic, thus clients become decoupled from the business services. Therefore, changes in the business logic do not affect the client implementations, as the clients use stable generic interfaces to interact with the remote system. The SERVICE ABSTRACTION LAYER increases, however, the level of indirection of requests. The introduction of this separate layer may reduce efficiency as all requests have to be processed at runtime.

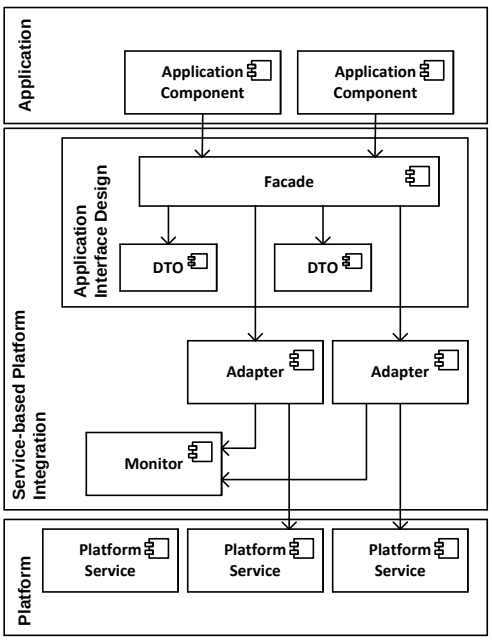

Fig. 9: Interface Design with Facade and Integration with Adapter

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 functionality can be grouped in separate EXTENSION INTERFACES [Schmidt et al. 2000b] and the common functionality can be included in a root interface. We illustrate in Figure 11 an example of an EXTENSION INTERFACE design. Clients access component functionality via interfaces. A component may provide multiple extension interfaces which implement the root interface functionality. Clients create new components and specify the initial extension interface using a factory associated with each component type.

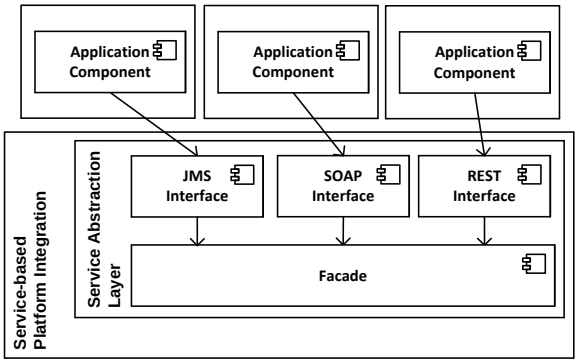

Fig. 10: Interface Design with Service Abstraction Layer

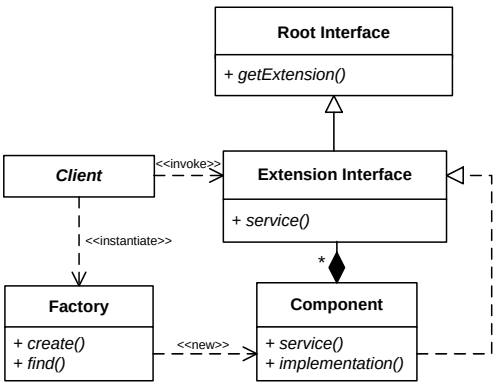

Fig. 11: An exemplary Extension Interface Design

EXTENSION INTERFACE [Schmidt et al. 2000b]

Problem The interface provided and exposed by a component is subject to adaptations and/or extensions due to changing requirements of client components. Similarly, an anonymous number of clients requires alternative, client-specific interfaces for component interfaces. Being limited to a single and monolithic component interface means that changes propagate into existing client components in an uncontrolled manner.

Solution Related functionality is grouped and exported via separate EXTENSION INTERFACES. This grouping results from domain-specific (e.g., functional views) and/or temporal bindings (e.g., interface versioning). Common and/or administrative functionality (e.g., for selecting a particular view or version) is exposed by a root interface, to be included by each single EXTENSION INTERFACE.

Decision Drivers The use of EXTENSION INTERFACES decreases the coupling between the clients and components. The clients depend only on the interface roles they actually use, which ensures that they do not break when signatures of services change or new services are added to the components. To extend the functionality of a component only new EXTENSION INTERFACES need to be added. EXTENSION INTERFACES can also be aggregated to offer a new functionality of a component that aggregates other components. EXTENSION INTERFACES, however, may cause additional indirection and runtime overhead, as they are introduced between the components and the clients. It can also lead to increased complexity of client programming, as the clients must decide which EXTENSION INTERFACES are suitable for their use case.

4.4 Communication Style

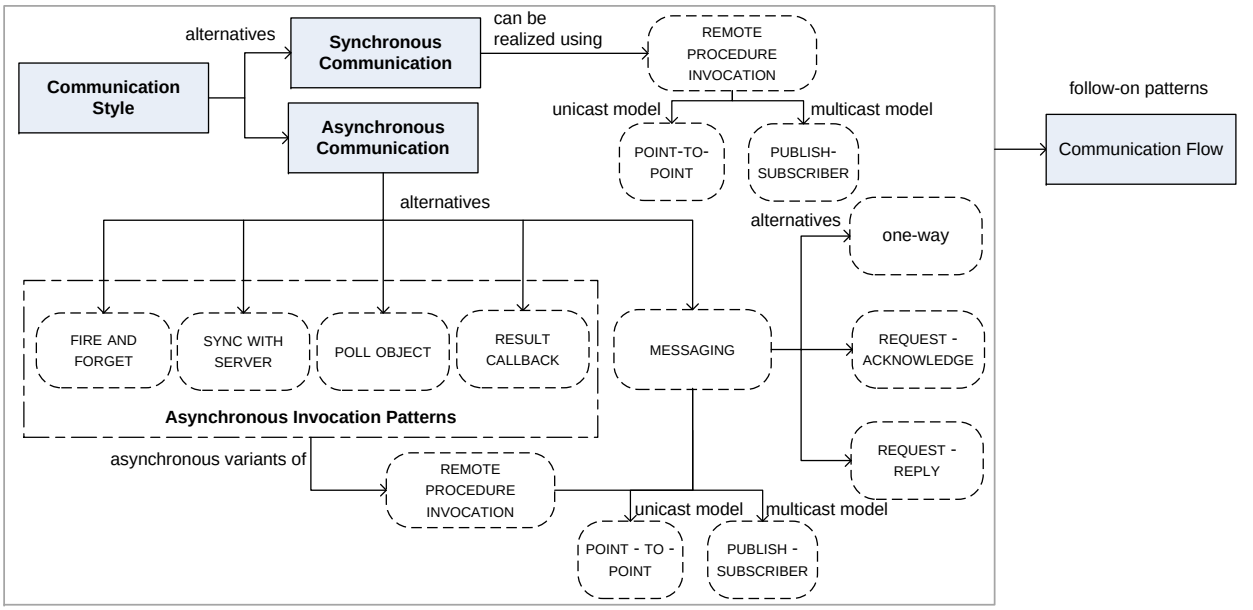

Fig. 12: Communication Style

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 taken at the component or service level, decisions on the communication style between the components must be tackled. 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 the service-based integration platform, connections between the 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 [Hohpe and Woolf 2004]. 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. 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 implementing 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 for invoking a remote service asynchronously. They describe asynchronous variants of the REMOTE PROCEDURE INVOCATION pattern.

REMOTE PROCEDURE INVOCATION [Hohpe and Woolf 2004]
Problem Applications in different programming languages that run on different platforms need to share data and processes.
Solution Using REMOTE PROCEDURE INVOCATIONS means that each application offers a remote interface to interact with the other applications. Thus, one application can get or change data from another application by calling its remote interface.
Decision Drivers Applying the pattern results in tightly coupled applications. It is difficult to deal with application downtimes, such as system crashes or downtimes for maintenance, as incoming invocations will get lost during the downtimes. Hence, REMOTE PROCEDURE INVOCATION based systems might be more unreliable than, e.g., MESSAGING based systems. Synchronous REMOTE PROCEDURE INVOCATION may lead to slow and blocking applications.

In particular, when a result or application error needs to be delivered either a POLL OBJECT [Völter et al. 2005] or a RESULT CALLBACK [Völter et al. 2005] can be used. A FIRE AND FORGET interaction [Völter et al. 2005] does not return any result or acknowledgment 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 [Völter et al. 2005] can be used instead of FIRE AND FORGET.

FIRE AND FORGET [Völter et al. 2005]
Problem A client application wants to notify a remote object for an event. Neither a result is expected, nor does the delivery have to be guaranteed. A one-way exchange of a single MESSAGE is sufficient.
Solution A FIRE AND FORGET operation is performed by the communication middleware without acknowledging the processing or delivery status to the client. The thread of control is yielded back to the client immediately.
Decision Drivers The FIRE AND FORGET pattern provides non-blocking communication with unreliable transmission. That means that the client is not notified of errors in transmission or execution of the remote object. The remote object does not deliver any execution results to the client.

While FIRE AND FORGET offers one-way communication, SYNC WITH SERVER follows the communication style REQUEST-ACKNOWLEDGMENT [Hohpe and Woolf 2004]. 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.

SYNC WITH SERVER [Völter et al. 2005]
Problem A client application needs to ensure higher reliability of asynchronous invocations than FIRE AND FORGET, but does not require the transmission of a result.
Solution The client sends the invocation as in FIRE AND FORGET but waits for a reply from the server about the successful transmission of the invocation. The communication middleware blocks only until the notification of the successful reception of the invocation arrives and then continues the execution.
Decision Drivers SYNC WITH SERVER ensures successful transmission of requests and makes the remote invocations more reliable than FIRE AND FORGET. It introduces, however, additional latency, as the client must block until the notification of successful reception is received. Thus, there is a trade-off between higher reliability and worse performance in comparison with FIRE AND FORGET. Also the server application cannot inform the client for application errors as the execution of the remote invocation happens asynchronously.

RESULT CALLBACK and POLL OBJECT offer the REQUEST-REPLY [Hohpe and Woolf 2004] communication style. POLL OBJECT can be used in an imperative programming model.

POLL OBJECT [Völter et al. 2005]
Problem Remote invocations of a client must be processed asynchronously but the client needs the result to continue its computations.
Solution A POLL OBJECT receives the results from a remote invocation on behalf of the client. The client periodically queries the POLL OBJECT for the results. The client can continue with other tasks and when the results are available to the POLL OBJECT the client can fetch them the next time it queries the POLL OBJECT.
Decision Drivers The server side stays oblivious to the client side POLL OBJECTS. The pattern offers more reliable communication compared to FIRE AND FORGET, as the result is an implicit acknowledgment, but it cannot immediately inform the client about an incoming result.

RESULT CALLBACK [Völter et al. 2005]
Problem The client needs to be informed about the results of its asynchronously invoked operations once the results become available to the communication middleware.
Solution A callback-based interface for remote invocations is provided on the client which passes a callback object to the communication middleware upon a remote invocation. After the invocation, the client can continue with other tasks. When the call completes and the results become available a callback is invoked on the client to process the result.
Decision Drivers The pattern has the same basic decision drivers as POLL OBJECT. RESULT CALLBACK is preferred over POLL OBJECT when an immediate reaction on the incoming result is needed. While POLL OBJECT works well with the imperative programming model of today's OO application, RESULT CALLBACK requires an event-based programming model to handle the callbacks. The same or different callback objects can be used for different invocations of the same type.

In asynchronous remote invocations, ASYNCHRONOUS COMPLETION TOKENS [Schmidt et al. 2000a] are used to associate the callback with the original invocation. The pattern fulfills the same role as the CORRELATION IDENTIFIER pattern [Hohpe and Woolf 2004] discussed below. To ensure reliability of communication and increase decoupling of the integrating platforms, MESSAGING [Hohpe and Woolf 2004] provides the most convenient solution. The integrating applications exchange MESSAGES [Hohpe and Woolf 2004] via a MESSAGE CHANNEL [Hohpe and Woolf 2004] which can be either a POINT-TO-POINT CHANNEL [Hohpe and Woolf 2004] or a PUBLISH-SUBSCRIBE CHANNEL [Hohpe and Woolf 2004]. 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 exist multiple receivers-subscribers of the messages.

MESSAGING [Hohpe and Woolf 2004]
Problem Applications that are developed independently, are built in different languages, and run on different platforms need to share data and processes in a responsive way.
Solution MESSAGES are used to transfer packets of data frequently, immediately, reliably, and asynchronously using customized formats.
Decision Drivers MESSAGING offers high reliability, as sending a message does not require both systems to be up and running at the same time. Instead, message queues in the messaging system can temporarily store the messages when systems are down. Hence, systems become more decoupled from each other than in REMOTE PROCEDURE INVOCATION. However, complexity increases, as many decisions about the messaging system, such as the message formats, the message routing, the message transmission and the connection of the applications to the messaging system, etc. have to be made.

The communication using MESSAGES can be either one-way or two-way. In a one-way communication, the sender directs a message towards 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 acknowledgment of its request and then poll for the results (REQUEST-ACKNOWLEDGE-POLL [Daigneau 2012]) or get notified about the delivery of its request and receive the request results with a callback service (REQUEST-ACKNOWLEDGE-CALLBACK [Daigneau 2012]).

The PUBLISH-SUBSCRIBE CHANNEL is the version of the PUBLISH-SUBSCRIBER [Buschmann et al. 2007a] 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. Hence, PUBLISH-SUBSCRIBER is an alternative to POINT-TO-POINT connections, mainly discussed so far, that can be applied for all communication styles discussed in this section. 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 or in REQUEST-ACKNOWLEDGE style (as in the SYNC WITH SERVER pattern).

PUBLISH-SUBSCRIBER [Buschmann et al. 2007a]
Problem Data changes in one place, but many components depend on this data and have to be updated. That means, multiple application components need to be notified about changes in one component.
Solution One dedicated component takes the role of the publisher and the other components are its subscribers who subscribe in order to get notified for state changes of the publisher. An object can be a subscriber to many publishers and can also play the role of both the subscriber and publisher.
Decision Drivers Publishers are loosely coupled to subscribers. PUBLISH-SUBSCRIBER allows listening for events without disturbing the communication flow. Thus, it can also be used for debugging and logging purposes. However, using PUBLISH-SUBSCRIBER may introduce security issues, as any subscriber is able to look at the events generated by the publisher. Although PUBLISH-SUBSCRIBER offers high scalability, it does not guarantee the delivery of events to the subscribers.

REQUEST-REPLY [Hohpe and Woolf 2004]
Problem Two applications communicate through an exchange of MESSAGES. Each MESSAGE realizes a one-way conversation. The sending application requires a reply from the receiver of the initial MESSAGE.
Solution To realize a two-way conversation, pairs of request and reply MESSAGES are exchanged. Depending on the intended coupling between the sender and receiver, the reply MESSAGE is sent either via the request's back channel or, alternatively, via its own communication channel.
Decision Drivers The pattern provides a two-way message transmission on a two-way channel. The requestor is always notified about the successful or unsuccessful completion and/or of the result of its request. The two processes (request and reply) are decoupled. If the connection between requestor and receiver fails before the reply is sent, then the requestor must re-send its request unless messages are persistent.

REQUEST-ACKNOWLEDGE [Daigneau 2012]
Problem A client would like to notify a system about the fact that a request has arrived or about an interesting event after a request has arrived. Requests do not need to be processed right away, and the responses to the requests do not need to be delivered.
Solution When a service receives a request it forwards the request to a background process and then returns an acknowledgment containing a unique request identifier.
Decision Drivers REQUEST-ACKNOWLEDGE communication alleviates the problem of unavailable resources or request spikes, as unlike ONE-WAY communication it lets the client know that the requests have been received and will be processed. However, the client does not get informed about application errors that may happen during the process execution.

4.5 Communication Flow

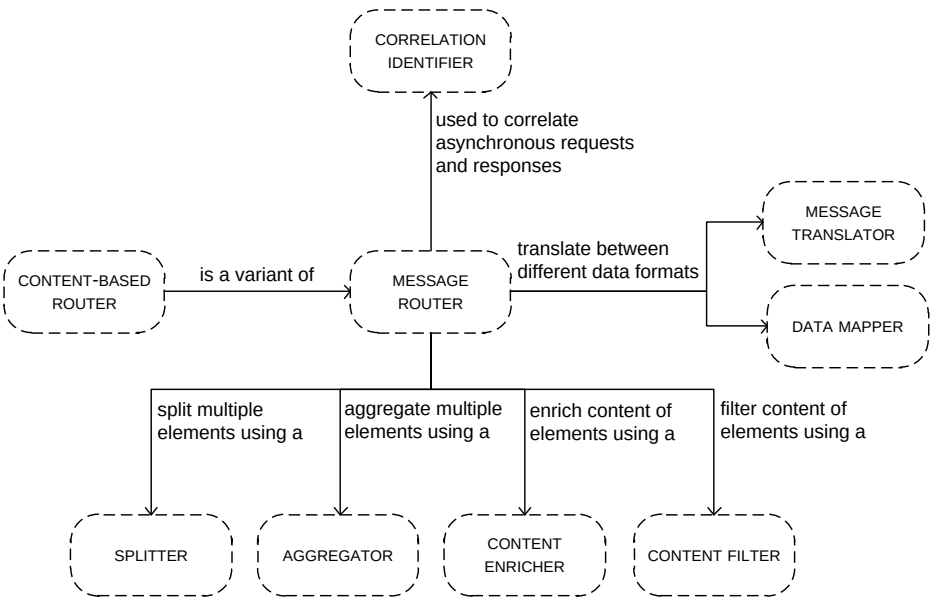

Fig. 13: Communication Flow

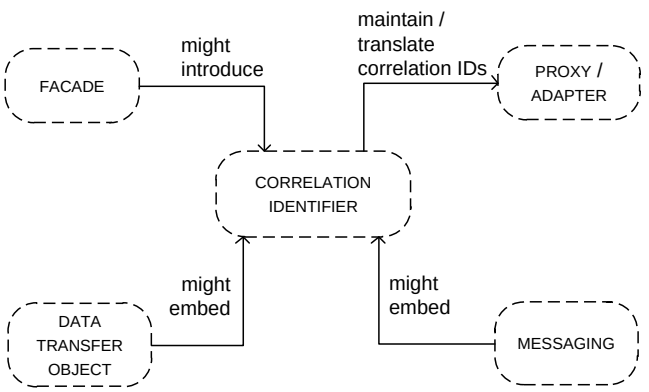

Fig. 14: Relationships between CORRELATION IDENTIFIER and other patterns

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, marshalling, 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 [Hohpe and Woolf 2004] 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 4.2).

MESSAGE ROUTER [Hohpe and Woolf 2004]
Problem A message channel can be used to exchange messages of varying structure and content, thus requiring different processing steps. To decouple the processing steps, without introducing dedicated message channels, the MESSAGES have to be filtered depending on a set of conditions.
Solution A MESSAGE ROUTER component inserts a special filter that consumes a MESSAGE from one incoming MESSAGE CHANNEL and republishes it to a different outgoing MESSAGE CHANNEL, after having evaluated certain filter criteria. MESSAGE ROUTERS distribute MESSAGES either to a fixed destination or redirect the messages depending on the message content (see CONTENT-BASED ROUTER).
Decision Drivers A MESSAGE ROUTER centralizes message filters and the actual filter functionality (routing rules) in a single component which becomes the single point of maintenance and failure. Message routing however adds to the processing overhead of the integration platform.

In a service-based integration platform, routing is often performed by a CONTENT-BASED ROUTER [Hohpe and Woolf 2004]. 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 ROUTER [Hohpe and Woolf 2004]
Problem An integration solution deploys a MESSAGE ROUTER to have MESSAGES processed adequately. However, the message routing is not to be decided by external factors or by fixed routes, but rather by the messages' content.
Solution A CONTENT-BASED ROUTER has the capacity to examine the message content and distributes the message to a different channel based on its content (e.g., routing data in the message envelope, its structure, or message values).
Decision Drivers As a kind of MESSAGE ROUTER, the use of a CONTENT-BASED ROUTER allows for centrally managing message routing; without the need to modify either the client applications or platform services. The routing functions, however, may change frequently causing extra maintenance effort. The recipients also have no control over the routing process.

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 [Buschmann et al. 2000; Avgeriou and Zdun 2005] 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 [Hohpe and Woolf 2004] disassembles the compound messages into their constituents which are expected by the target platform(s). 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.

SPLITTER [Hohpe and Woolf 2004]
Problem Messages passing through an integration solution consist of multiple elements, each of which must be processed separately. The incoming message appears as a composite message.
Solution A SPLITTER component is incorporated into the integration platform to break up the composite message into a series of individual elements or element subsets. Each element subset is then published as a distinct message. Common elements of the initial message are maintained in the resulting messages (e.g., identification and sequencing tokens) in order to allow for re-integrating reply messages later on.
Decision Drivers The primary driver for adopting a SPLITTER is that target platform services, which are grouped by a dedicated FACADE, must receive the respective data subsets for which they are responsible when answering invocations dispatched onto the facade interface. A SPLITTER can not only break a message into its repetitive data chunks, but also a large message into individual messages to simplify the further processing. On the negative side, there is data overhead in the resulting messages (e.g., CORRELATION IDENTIFIERS, timestamps). Also, extra processing effort is needed for aggregating reply messages afterwards.

Conversely, an AGGREGATOR [Hohpe and Woolf 2004] 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 [Hohpe and Woolf 2004] is included in the processing chain of a message to extract and drop excessive data.

AGGREGATOR [Hohpe and Woolf 2004]
Problem We need to combine the data of individual, but related messages so that the aggregated data can be processed as a whole by the target system.
Solution An AGGREGATOR component observes the message stream, collects and stores individual messages based on filter criteria and identification tokens (CORRELATION IDENTIFIERS) until it has received a complete set of related messages. After having assembled a single message out of these parts, based on selected aggregation strategies, the resulting message is published for delivery to the target system.
Decision Drivers In order to be able to aggregate incoming messages to an AGGREGATOR the messages need to have a correlation that indicates which messages belong together. The aggregation decisions described by the aggregation algorithm bear the risk of introducing extra development effort and additional processing complexity (depending on the aggregation strategy). At the same time, message aggregation can realize an important optimization strategy in service-based platform integration: the batch processing of service invocations. For batching, multiple content-wise unrelated messages are packed into a single composite message to be delivered to a platform service.

CONTENT FILTER can be applied in the integrated platform to filter the messages that pass through it.

CONTENT FILTER [Hohpe and Woolf 2004]
Problem When sending messages from one system to another, it is common that situations occur in which the target system is not interested in all data included in the forwarded messages.
Solution A CONTENT FILTER component is provided to remove unneeded, obsolete, or protected data items from a message.
Decision Drivers Data filtering for messages is a critical adaptation mechanism to be supported by an integration platform. It can also simplify the structure of a message (e.g., convert a tree into a flat structure), or remove redundancy or ambiguity in a data structure. CONTENT FILTERS can act as pre-processors before handling messages using MESSAGE TRANSLATORS.

Requirements for additional data can result from domain model mismatches, different underlying middleware, or security requirements. In such cases, a CONTENT ENRICHER [Hohpe and Woolf 2004] 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.

CONTENT ENRICHER [Hohpe and Woolf 2004]
Problem When sending messages from one system to another, it is common that situations occur, in which the target system requires more data than included in the original message.
Solution A CONTENT ENRICHER is a specialized message transformer which accesses data sources external to the message processing system to add the missing data.
Decision Drivers A CONTENT ENRICHER may need to consult external resources to find the required data, based on references contained in a message. This may not only affect the message processing throughput negatively, but also the synchronization decoupling in the communication flow. External resources might require synchronous communication, which requires special treatment for the enricher component (e.g., introducing a special message consumer internal to the integration platform). If the external source is an integrated platform service, the CONTENT ENRICHER acts as a variant of AGGREGATOR.

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 [Fowler 2003] 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 [Völter et al. 2005] and PROTOCOL PLUG-IN [Völter et al. 2005] components as offered by the underlying middleware framework.

DATA MAPPER [Fowler 2003]
Problem Two or more components exchange data in terms of in-memory objects. However, the receiving interfaces require an incoming data format which is incompatible with the object structures exchanged. A format mismatch is the consequence, covering inconsistencies at the level of two incompatible data models, and in-memory representation styles.
Solution Incorporate an auxiliary transformation layer in the component architecture which hosts a group of mapper components. These DATA MAPPERS provide model and representation transformations for data objects. Certain model and representation strategies are so captured as dedicated modules and can be applied to data objects exchanged between different pairings of client applications and platform services.
Decision Drivers The processing of service data in terms of in-memory objects requires collocation of the two components for which the data is mapped. PROXIES can be used when process or machine boundaries need to be crossed. When crossing process and machine boundaries, MESSAGE TRANSLATORS take the role of format filters.

MESSAGE TRANSLATOR [Hohpe and Woolf 2004] 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 (see Figure 15). The MESSAGE TRANSLATOR can reside, for instance, in the integration platform and translate between the client application message formats and the message formats of the platform services.

MESSAGE TRANSLATOR [Hohpe and Woolf 2004]
Problem Two or more interacting applications operate on different message formats and there is a message format mismatch.
Solution A dedicated filter component is used by the MESSAGE ROUTER to transform an incoming message format into an outgoing message format. A single MESSAGE TRANSLATOR can cover any, or even all, levels of transformation (model, type, representation, and transport).
Decision Drivers A key driver for providing a MESSAGE TRANSLATOR component in the integration platform is to avoid enforcing a uniform message format throughout all (existing and future) client applications and platform services; if possible at all. A MESSAGE TRANSLATOR preserves the independence of the integrated clients and services in terms of message formats; the adoption of different standard formats or the modification of proprietary ones remain a localized event without propagating to the other participants. However, depending on the degree of heterogeneity within the distributed systems in terms of message formats to be mated, there is the risk of high complexity (i.e., combinatorial explosion of message format transformations). This is particularly valid for each MESSAGE TRANSLATOR realizing all transformation steps, potentially in redundancy to other translators. This risk can be reduced by limiting a single translator's responsibility to a single transformation step (e.g., marshalling) and combining them to message processing chains for a given integration scenario (e.g., a pair of client application and platform service).

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 [Hohpe and Woolf 2004] 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 as ASYNCHRONOUS COMPLETION TOKENS [Buschmann et al. 2007a].

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 (see Section 4.3).

In a MESSAGING infrastructure, the CORRELATION IDENTIFIER is extensively used to realize conversational interactions, i.e., for exchanging and processing messages such as in REQUEST-REPLY interactions and MESSAGE SEQUENCE interactions [Hohpe and Woolf 2004], to name but a few.

CORRELATION IDENTIFIER [Hohpe and Woolf 2004]
Problem Using asynchronous remote invocations or messages, the requesting component does not block, even if a reply is expected as part of an asynchronous REQUEST-REPLY conversation. However, upon incoming an asynchronous reply, the receiving components must align the incoming reply to the corresponding request.
Solution To correlate two messages, such as a request and a reply processed at different times, both messages embed a unique identity token. In the request message, the CORRELATION IDENTIFIER is referred to as the request ID. The reply message then includes a token, the CORRELATION IDENTIFIER, which matches or refers to the initial request ID.
Design Drivers For general MESSAGING and asynchronous REMOTE PROCEDURE INVOCATION scenarios, it is sufficient to generate and maintain unique IDs for the messages. In service-based platform integration, it is also required to preserve a reference to the client application having submitted the original request, along with the unique identifier for the message as such. This is needed to resume serving a service invocation for a particular client across the asynchronous frontend connector of an INTEGRATION ADAPTER. Assuming that the client applications should not and cannot be altered to emit an additional identifier token, to be used as the CORRELATION IDENTIFIER or as a part of it, the integration platform has to maintain a mapping table which aligns the requesting clients and the CORRELATION IDENTIFIERS issued.

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 that 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 15. Depending on the decisions taken on the communication styles (see Section 4.4), 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 15.

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.

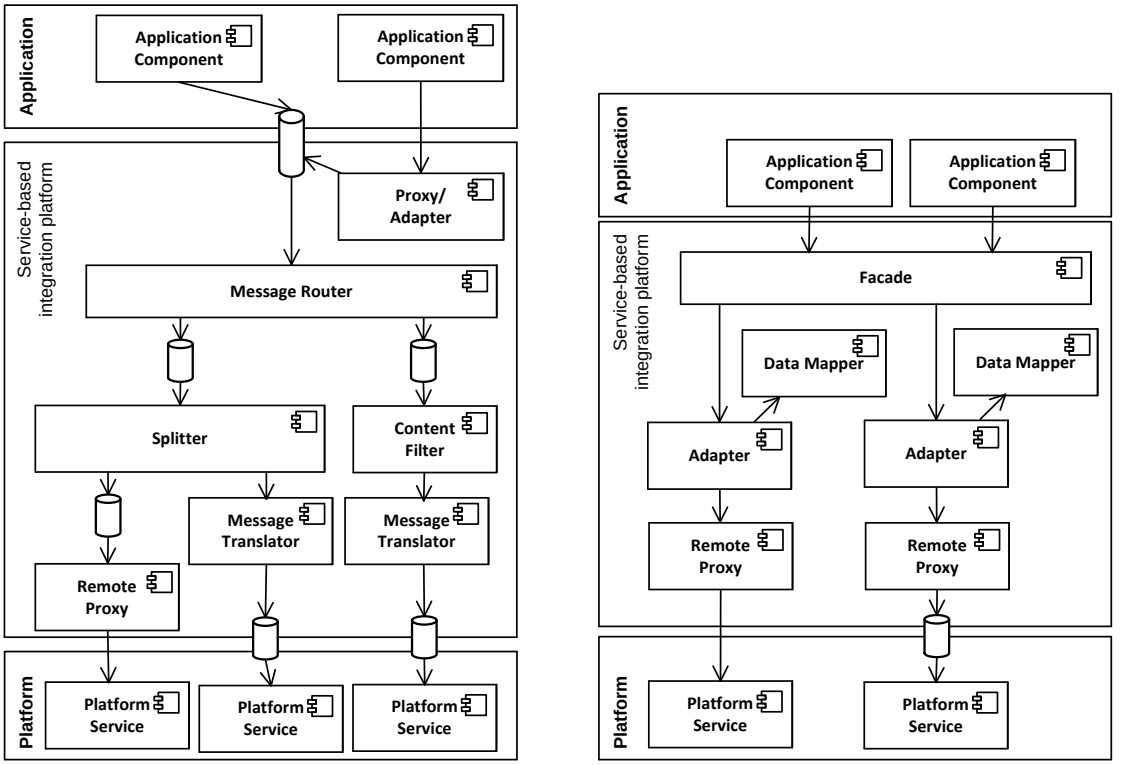

Fig. 15: Organizing Communication Flows in a Service-based Integration Platform

5. MOTIVATING EXAMPLE REVISITED

In this section, we apply our pattern language to the industry case study introduced in Section 3, with special emphasis on a single integration scenario as illustrated in Figure 3. We show, thus, how the patterns presented in this paper can be used individually or in combination for building a service-based platform integration solution. In order to make informed decisions, we consult the pattern descriptions. For selecting follow-on patterns, we review the pattern interconnections as documented for the pattern language throughout Section 4.

In Figure 16, we present an excerpt from the integration architecture containing the three backend platforms (i.e., the Yard Management System YMS, the Warehouse Management System WMS, and the Remote Maintenance System RMS), the Virtual Service Platform (VSP), and the operator application. We select appropriate patterns from the pattern categories *Integration and Adaptation* and *Interface Design*. The services introduced for the integration scenario are grouped into components; for example, the services *initiateVoiceCall* and *endCall* are enclosed by the component *CallHandling*. A FACADE component (*OperatorAppFacade*) provides a common application interface for

invoking the different platform services. The component *CommunicationFlowManager* embodies the communication flow between the operator application and the integrated platforms. In order to invoke the remote platform services, ADAPTER and PROXY components are introduced into an integration layer below the *CommunicationFlowManager* component. In case the access to the remote services does not require any interface adaptations, a PROXY component is used (e.g., *TruckManagementProxy*, *PositionReportingProxy*, etc.). Otherwise, an ADAPTER component is deployed to resolve interface incompatibilities, i.e., changes to the parameter structure (e.g., *CallHandlingAdapter* and *VideoHandlingAdapter*) or to operation names.

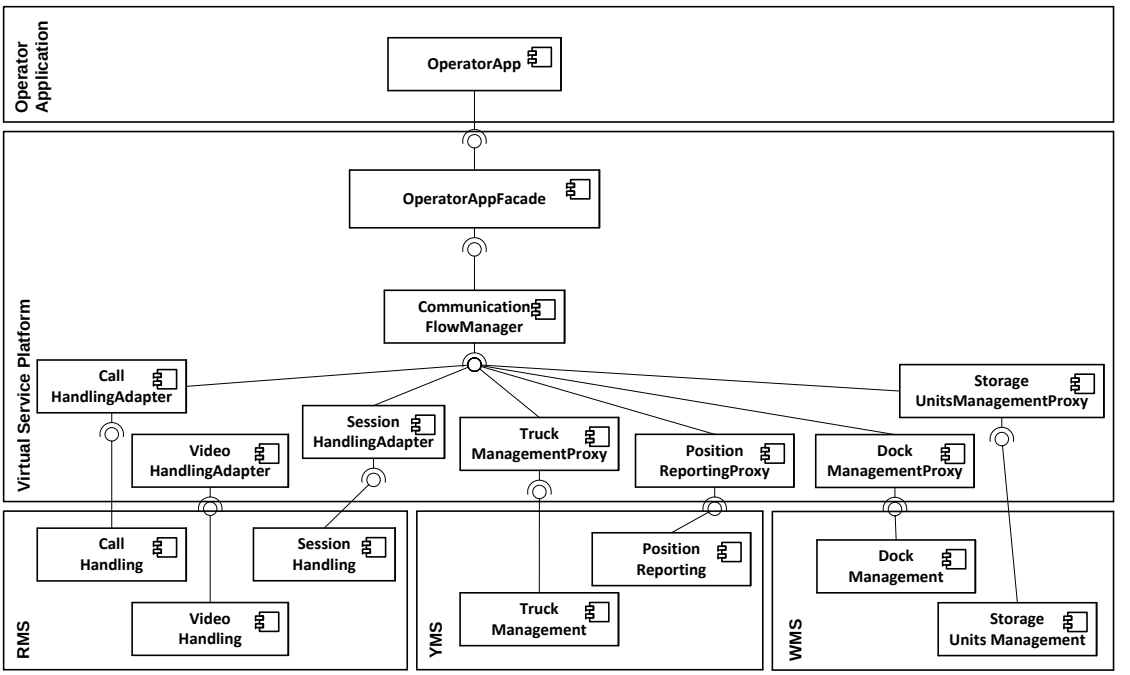

Fig. 16: Excerpt from the Integration Architecture

In Figure 17, we introduce two exemplary designs of the communication flows between the operator application and the three platforms as provided by the *CommunicationFlowManager* component in Figure 16. For this, we select appropriate patterns from the pattern category *Communication Flow*. In the first communication flow diagram, the platforms send notifications. These notifications are assigned CORRELATION IDENTIFIERS before they are enriched with platform details required by the operator (*WMSNotificationEnricher*, *YMSNotificationEnricher*, *RMSNotificationEnricher*). The atomic notifications are aggregated into one notification (in the *PlatformNotificationAggregator*) which is then delivered to the operator application. To receive the notifications, the operator is expected to subscribe to the appropriate notification channel (following the PUBLISH-SUBSCRIBER pattern). In the second communication flow diagram, the operator invokes the operation *moveTruckToDock* and the request receives a CORRELATION ID. Afterwards the request is logged using a PUBLISH-SUBSCRIBER interaction style. Finally, the request is added to the *TruckRequestsQueue* message queue at which the YMS is listening for incoming tasks.

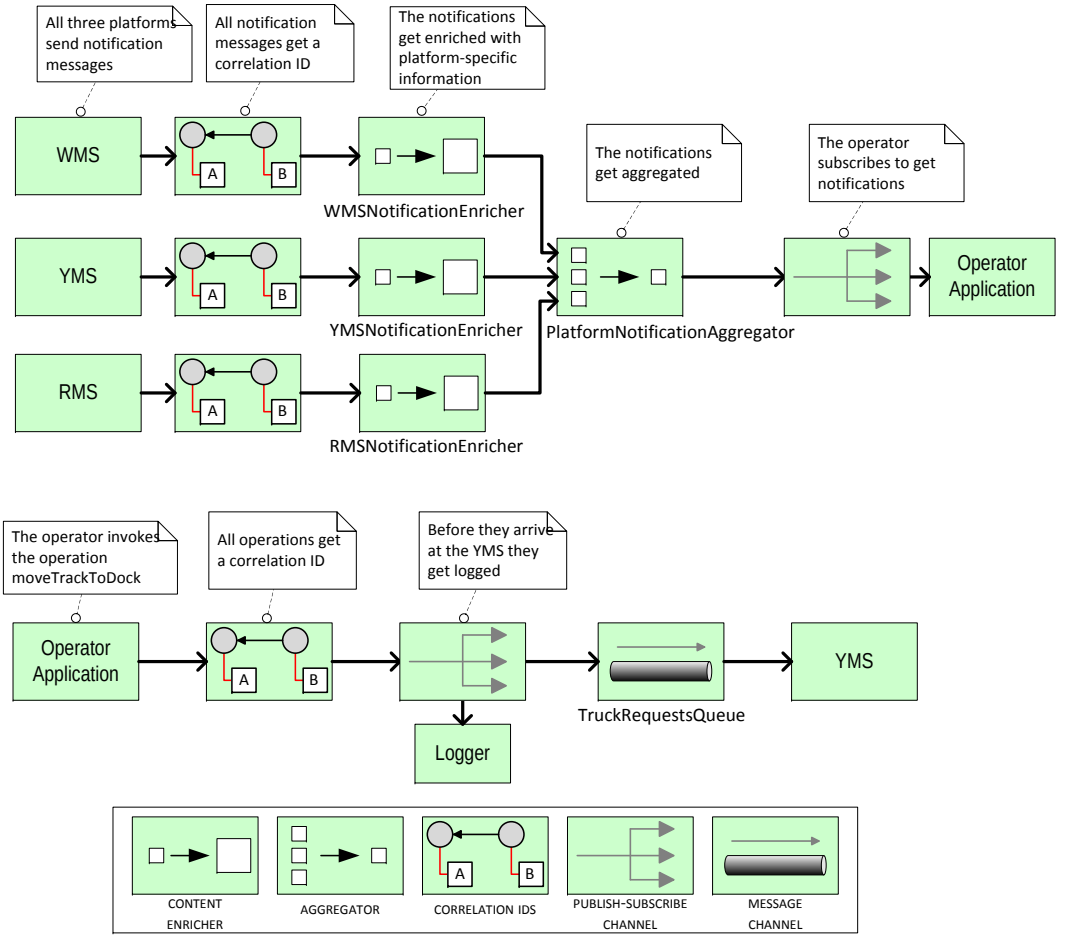

Fig. 17: Examples of Communication Flows

6. RELATED WORK

Our pattern language describes a family of service-oriented architectures [Hentrich and Zdun 2012] for platform-integration purposes. From this viewpoint, our pattern language complements architectural patterns emphasizing application, service, and business process integration in the large [Hohpe and Woolf 2004] by adding the architectural description of another integration style, service-based integration platforms, which has not been explicitly discussed in pattern language form before.

However, our work is closely related to many other pattern languages. The works most closely related are existing pattern languages that are integrated into ours. In particular, we used selected patterns from those other pattern languages in the specific context of service-based platform integration and adaptation. With this, we documented missing links and decision drivers (i.e., forces and consequences). For decisions relating to adapting and integrating platform services, patterns on SOA integration [Hentrich and Zdun 2012; Vogel 2001; Buschmann et al. 2007b] and software architecture integration [Avgeriou and Zdun 2005; Gamma et al. 1994] have been consulted. Regarding decisions on service and component interface design, we identified relevant patterns in two pattern languages on general software and service-oriented architectures [Buschmann et al. 2000; Avgeriou and Zdun 2005], as well as in the pattern

language on enterprise application integration [Fowler 2003]. For decisions regarding the communication style and communication flow in the platform integration solution, patterns on object remoting middleware [Völter et al. 2005] and messaging systems [Hohpe and Woolf 2004] have been integrated.

Zdun and Hentrich [Hentrich and Zdun 2009; 2012] describe a pattern language for process integration architectures. This pattern language can be combined with the patterns described in this paper, by using either a workflow or a business process engine (called macroflow engine in [Hentrich and Zdun 2009; 2012]). To the same effect, a microflow engine for handling the integration logic can be incorporated into the service-based integration platform. That is, the process-based integration patterns can be used to replace the communication style and communication flow patterns, described in our pattern language, to support the process-based integration style in a service-based integration platform.

Closely related are contributions about managing the variability in a SOA, in particular the pattern-based approaches by Khan et al. [Khan et al. 2011]. By managing variability, we mean the separated specification of differences (i.e., variation points and the binding options) between the variants to be instantiated from a SOA (see also SEPARATE DESCRIPTION OF VARIABILITY in [Voelter 2009]) and the corresponding implementation of the so-specified variants using different variability techniques (e.g., static and dynamic parametrization, injection). SOA variability [Ruokonen et al. 2008] must be dealt with or must even be realized in a service-based integration platform. Prominent examples for SOA variations stressed in our paper are forms of runtime re-configuration of platform internals (e.g., interface adapters) by applying the COMPONENT CONFIGURATOR pattern. Varying service interfaces (e.g., interface evolution over time, client-specific interfaces) and substituting services (e.g., for different client bases) are addressed by the various patterns touching service interface design (e.g., FACADE; see Section 4.3).

While there have been numerous contributions for describing and managing variability in SOAs (see more recently, e.g., [Chang and Kim 2007; Narendra and Ponnalagu 2010; ai Sun et al. 2010; Nguyen et al. 2011]), to the best of our knowledge, the approach by Khan et al. [Khan et al. 2011] is the only other pattern-based approach existing so far. Khan et al. [Khan et al. 2011] present a catalog of six patterns for describing recurring variability problems and variability solutions in SOAs. The patterns touch on service parametrization (e.g., distinguishing between invocation parametrization and extrinsic configuration through various configuration descriptors), on conditional routing of service invocations (e.g., based on business rules), on providing signature compatibility for services, on client-driven service adaptation (e.g., by injecting behavior into existing services through configuration interfaces), and on service cloning (to serve client applications independently from each other). While the pattern collection sketches the considerable range of variability in a SOA, it is severely limited. Some patterns relate to very specific SOA variants, in particular SaaS systems; others address general issues of SOA adaptation and SOA integration without referring to the extensive body of existing pattern works on these subject matters. For example, the PARAMETER and SERVICE WRAPPER patterns closely relate to patterns on component configuration and different variants of the ADAPTER pattern [Gamma et al. 1994]. Finally, the relationship among the six patterns and between the six patterns and related patterns are not described.

In recent years, software architecture is often seen as the principal design decisions governing a system [Taylor and van der Hoek 2007; Jansen and Bosch 2005]. An important idea is to document the design rationale of the architecture using means such as architectural design decisions (ADDs). ADD approaches propose prescriptive architecture design decisions meta-models for structuring, relating, and navigating the actual templates created for a given architecture [Tyree and Akerman 2005; Kruchten et al. 2006; de Boer et al. 2008; Zimmermann et al. 2009; Capilla et al. 2011]. As conceptual meta-models, ADDs are decomposed into compounds of decision descriptions, decision alternatives, decision groups, artifacts and activities related to individual decisions; and the relations between these building blocks.

By reusing and linking ADDs to patterns as design artifacts [Zimmermann et al. 2008; van Heesch and Avgeriou 2009], documenting architecture decisions can be substantially facilitated [Harrison et al. 2007]. The work by Zimmermann et al. has integrated various SOA pattern languages in a reusable ADD model. For Zimmermann et al. [Zimmermann et al. 2009] patterns take the role of architectural decision alternatives, i.e., they represent the solution space of an ADD. The authors further discriminate between four different levels of decisions (executive, conceptual etc.), and different kinds of patterns as decision output are proposed for each level: At the executive decision level,

process and requirement analysis patterns enter as decision options. Then, at the so-called conceptual decision level, high-level architectural patterns (e.g., BROKER vs. SHARED REPOSITORY) and critical technology choices follow. At the third and technological decision level, design and remoting patterns apply as decision alternatives [Völter et al. 2005]. For asset-level decisions, implementation-level patterns and concrete technology options apply.

Drawing upon their experiences using a structured Wiki as an ADD documentation tool and a SOA-centric industry case study, Capilla et al. [Capilla et al. 2011] consider architectural patterns as concrete decision alternatives. Regarding process-driven SOA patterns [Hentrich and Zdun 2012], patterns enter the concrete solution catalog maintained by the Wiki-based documentation tooling. In terms of their ADD meta-model, SOA patterns predominantly represent decision alternatives (`ADAlternatives`), but also decision issues and outcomes.

7. CONCLUDING REMARKS

A typical (service-based) application often relies on functions provided by different (service) platforms specialized for different domains. As a consequence, many applications are faced with the requirement for integration of services from one or even multiple heterogeneous platforms. However, platform integration is a rather challenging task as the software architects and developers are confronted with several design decisions at different levels of abstractions and different levels of granularity. There is a considerable amount of patterns targeting various aspects of service-based integration and adaptation [Gamma et al. 1994; Buschmann et al. 2000; Buschmann et al. 2007a; Fowler 2003; Hohpe and Woolf 2004; Völter et al. 2005; Hentrich and Zdun 2009; Daigneau 2012; Hentrich and Zdun 2012]. Unfortunately, these patterns, on the one hand, have been documented with a different focus and, on the other hand, walking through several patterns scattered in different literature in order to arrive at a design solution for service platform integration is tedious and time-consuming.

The major contribution of this paper is to revisit the existing patterns and design decisions regarding service-based integration and adaptation of platforms and organize them in a comprehensive pattern language such that software architects and developers can systematically reference and follow the pattern language to build up 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 *Integration and Adaptation*, *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.

While our pattern language covers the core design space of service-based platform integration, there are many open issues relevant for the design of platform integration solutions, but not covered yet in our pattern language. Our future endeavors will consider categories such as monitoring, QoS and SLAs for platform integration solutions, more sophisticated adaptation options, and further patterns at lower levels of abstraction and finer levels of granularity. In addition, follow-on tool support would also be useful to enable software architects in better devising and utilizing an adequate set of questions for developing and documenting a certain architectural design in service platform integration and adaptation. The pattern language described in this paper will provide the basis for such tools supporting architectural design.

8. ACKNOWLEDGEMENTS

Thanks are due to Michael Weiss as our EuroPLoP shepherd and to all EuroPLoP 2012 workshop participants who contributed to improving our paper with their comments. This work was partially supported by the European Union FP7 project INDENICA (`http://www.indenica.eu`), grant no. 257483.

REFERENCES

AI SUN, C., ROSSING, R., SINNEMA, M., BULANOV, P., AND AIELLO, M. 2010. Modeling and managing the variability of Web service-based systems. *Journal of Systems and Software 83,* 3, 502–516.

AVGERIOU, P. AND ZDUN, U. 2005. Architectural patterns revisited – a pattern language. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*. Irsee, Germany, 1–39.

BUSCHMANN, F., HENNEY, K., AND SCHMIDT, D. C. 2007a. *Pattern-Oriented Software Architecture — A Pattern Language for Distributed Computing*. Wiley Series in Software Design Patterns Series, vol. 4. John Wiley & Sons Ltd., New York.

BUSCHMANN, F., HENNEY, K., AND SCHMIDT, D. C. 2007b. *Pattern-Oriented Software Architecture – On Patterns and Pattern Languages*. Wiley Series on Software Design Patterns. John Wiley & Sons Ltd., Chichester, England.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M., Eds. 2000. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons Ltd., Chichester, England.

CAPILLA, R., ZIMMERMANN, O., ZDUN, U., AVGERIOU, P., AND KÜSTER, J. 2011. An enhanced architectural knowledge metamodel linking architectural design decisions to other artifacts in the software engineering lifecycle. In *Software Architecture*, I. Crnkovic, V. Gruhn, and M. Book, Eds. Lecture Notes in Computer Science Series, vol. 6903. Springer Berlin / Heidelberg, 303–318.

CHANG, S. H. AND KIM, S. D. 2007. A Variability Modeling Method for Adaptable Services in Service-Oriented Computing. *Proceedings of the 11th International Software Product Line Conference (SPLC'11)*, 261–268.

CHRISTENSEN, E., CURBERA, F., MEREDITH, G., AND WEERAWARANA, S. 2001. Web Services Description Language (WSDL) 1.1.

DAIGNEAU, R. 2012. *Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services*. Addison-Wesley.

DE BOER, R. C., FARENHORST, R., LAGO, P., VAN VLIET, H., CLERC, V., AND JANSEN, A. 2008. Architectural knowledge: Getting to the core. In *Post-Conference Proceedings of Third International Conference on Quality of the Software Architectures, Components, and Applications (QoSA 2007) Medford, MA, USA, July 11-23, 2007*. Lecture Notes in Computer Science Series, vol. 4880. Springer, 197–214.

FOWLER, M. 2003. *Patterns of Enterprise Application Architecture* 4th Ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

GAMMA, E., HELM, R., JOHNSON, R. E., AND VLISSIDES, J. 1994. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison Wesley.

GHANAM, Y., MAURER, F., AND ABRAHAMSSON, P. 2012. Making the leap to a software platform strategy: Issues and challenges. *Information and Software Technology 54,* 9, 968–984.

HARRISON, N., AVGERIOU, P., AND ZDUN, U. 2007. Using patterns to capture architectural decisions. *IEEE Software 24,* 4, 38–45.

HENTRICH, C. AND ZDUN, U. 2009. A Pattern Language for Process Execution and Integration Design in Service-Oriented Architectures. In *Transactions on Pattern Languages of Programming I*, J. Noble and R. Johnson, Eds. Lecture Notes in Computer Science Series, vol. 5770. Springer Berlin / Heidelberg, 136–191.

HENTRICH, C. AND ZDUN, U. 2012. *Process-Driven SOA: Patterns for Aligning Business and IT*. Infosys Press.

HOHPE, G. AND WOOLF, B. 2004. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* 2nd Ed. Addison-Wesley.

JANSEN, A. AND BOSCH, J. 2005. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEE/IFP Conference on Software Architecture, WICSA*.

KHAN, A., KÄSTNER, C., KÖPPEN, V., AND SAAKE, G. 2011. Service variability patterns. In *Proceedings of the ER 2011 Workshops on Advances in Conceptual Modeling, Recent Developments, and New Directions*. Number 6999 in Lecture Notes in Computer Science. 130–140.

KRUCHTEN, P., LAGO, P., AND VAN VLIET, H. 2006. Building up and reasoning about architectural knowledge. In *Quality of Software Architectures*, C. Hofmeister, I. Crnkovic, and R. Reussner, Eds. Lecture Notes in Computer Science Series, vol. 4214. Springer Berlin / Heidelberg, 43–58.

NARENDRA, N. C. AND PONNALAGU, K. 2010. Towards a Variability Model for SOA-Based Solutions. *Proceedings of the IEEE International Conference on Services Computing 2010*, 562–569.

NGUYEN, T., COLMAN, A., AND HAN, J. 2011. Modeling and Managing Variability in Process-Based Service Compositions. In *Proceedings of the 9th International Conference on Service-Oriented Computing (ICSOC'09)*, G. Kappel, Z. Maamar, and H. Motahari-Nezhad, Eds. Lecture Notes in Computer Science Series, vol. 7084. Springer-Verlag, 404–420.

ROSENMÜLLER, M. AND SIEGMUND, N. 2010. Automating the Configuration of Multi Software Product Lines. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems*. ICB-Research Report Series, vol. 37. Universität Duisburg-Essen, 123–130.

RUOKONEN, A., RAISANEN, V., SIIKARLA, M., KOSKIMIES, K., AND SYSTA, T. 2008. Variation needs in service-based systems. In *Proceedings of the 6th European Conference on Web Services*. IEEE Computer Society, 115–124.

SCHMIDT, D. C., STAL, M., ROHNERT, H., AND BUSCHMANN, F. 2000a. *Pattern-Oriented Software Architecture*. John Wiley & Sons Ltd.Wiley, Chichester, England, Chapter Extension Interface, 141–174.

SCHMIDT, D. C., STAL, M., ROHNERT, H., AND BUSCHMANN, F. 2000b. *Pattern-Oriented Software Architecture*. John Wiley & Sons Ltd.Wiley, Chichester, England, Chapter Interceptor, 109–141.

SCHMIDT, D. C., STAL, M., ROHNERT, H., AND BUSCHMANN, F. 2000c. *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*. Wiley Series in Software Design Patterns Series, vol. 2. John Wiley & Sons Ltd.Wiley, Chichester, England.

TAYLOR, R. N. AND VAN DER HOEK, A. 2007. Software design and architecture: The once and future focus of software engineering. *Future of Software Engineering (FOSE '07)*, 226–243.

TYREE, J. AND AKERMAN, A. 2005. Architecture decisions: Demystifying architecture. *IEEE Software 22*, 19–27.

VAN HEESCH, U. AND AVGERIOU, P. 2009. A pattern driven approach against architectural knowledge vaporization. In *Proceedings of 14th Annual European Conference on Pattern Languages of Programming (EuroPLoP 2009), Irsee, Germany, July 8-12, 2009*. CEUR Workshop Proceedings Series, vol. 566. CEUR-WS.org.

VOELTER, M. 2009. Variability patterns. In *Proceedings of the 14th European Conference on Pattern Languages of Programs (EuroPLoP 2009), July 8-12, Irsee, Germany*, M. Weiss, Ed. CEUR Workshop Proceedings Series, vol. 566. CEUR-WS.org.

VOGEL, O. 2001. Service Abstraction Layer. In *Proceedings of EuroPLoP 2001*. Irsee, Germany.

VÖLTER, M., KIRCHER, M., AND ZDUN, U. 2005. *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. Software Design Patterns. John Wiley & Sons Ltd., Chichester, England.

ZIMMERMANN, O., KOEHLER, J., LEYMANN, F., POLLEY, R., AND SCHUSTER, N. 2009. Managing architectural decision models with dependency relations, integrity constraints, and production rules. *Journal of Systems and Software 82,* 8, 1249–1267.

ZIMMERMANN, O., ZDUN, U., GSCHWIND, T., AND LEYMANN, F. 2008. Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method. In *Proceedings of the Seventh Working IEEE / IFIP Conference on Software Architecture (WICSA 2008), 18-22 February 2008, Vancouver, BC, Canada*. IEEE Computer Society, 157–166.