Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Foundational MDA Patterns for Service-Oriented Computing

Colin Atkinson[a] Philipp Bostan[a] Dirk Draheim[b]

a. Software Engineering Group, University of Mannheim, Germany

b. Data Center, University of Innsbruck, Austria

Abstract As the foundation of EDI, B2C and B2B, distributed computing is a key enabler for today's enterprises and will become even more important with the advent of cloud computing on the one hand and an ever more agile work organization on the other hand. Whilst the rapid evolution of distributed computing technologies in the last three decades has delivered a rich set of platforms and paradigms for building robust enterprise systems, it has also left a legacy of unresolved problems including fundamental inconsistencies between the concepts of the two leading distributed computing paradigms, i.e., distributed object computing and service-oriented architecture. Equally important, there is a growing need to lower the complexities and barriers involved in developing client applications, which range from large scale business applications and business processes to laptop programs and small *apps* on mobile devices. In this article, we present a unified conceptual framework for service-oriented computing based on the proven MDA (Model Driven Architecture) terminology stack. With the conceptual framework we consolidate, and give semantics to, core concepts of service-oriented computing and provide a set of foundational model transformation patterns that map between the presented concepts and further clarify practical service-oriented computing scenarios. Finally, we show how the developed framework perfectly fits to the OSM (Orthographical Software Modeling) approach.

Keywords distributed computing, model-driven development, service-oriented architectures

1 Introduction

Distributed computing technologies are today key enablers for enterprise computing. Distributed computing is the basis by which IT (information technology) is able to go beyond the confines of single data centers and support general ICT (information and communication technology). Crucial ICT paradigms like EDI (electronic data interchange) [Emm93], B2C (business to customer) and B2B (business to business)

Colin Atkinson, Philipp Bostan, Dirk Draheim. *Foundational MDA Patterns for Service-Oriented Computing*. In Journal of Object Technology, vol. 14, no. 1, 2015, pages 1–30. doi:10.5381/jot.2015.14.1.a2

that changed the way we do business are all in the realm of distributed computing. And again, the emerging technological paradigms of ubiquitous computing [Wei91], cloud computing as well as smart manufacturing [Coa11] heavily rely on distributed computing technology. Similarly, BYOD (bring your own device) is not merely a buzzword. Superficially, it is about bringing gadgets to your staff; however, what we actually currently observe in enterprises is a fundamental transformation of work organization resulting into ever more agile work models and work forces. In the long run, we cannot ignore the demands of generation Y. All this makes us sure that distributed computing is becoming ever more important and pervasive.

Historically, distributed computing technologies became viable, scalable solutions no earlier than in the 1980s. Basically, we have seen four major waves of distributed computing technologies, i.e., client/server computing, distributed object computing, service-oriented architecture and cloud computing. The first wave, i.e., client/server computing, emphasized the separation of GUI, application logic and data persistence concerns. Key technologies of the client/server computing wave were transaction monitors [GR92] and message-oriented middleware (MOM) [Ber96]. The second wave, i.e., distributed object computing, established the use of object-oriented abstractions in the construction of distributed systems. Key technologies supporting this approach include CORBA (Common Object Request Broker Architecture) [Gro06] and J2EE (Java 2 Enterprise Edition) [KtET00]. The third and still current wave, i.e., SOA (Service-Oriented Architecture), focuses on the Internet as the underlying distribution platform. Key SOA technologies are therefore Web Services and the recent REST (Representational State Transfer) services paradigm [FT02].

The fourth major wave, i.e., cloud computing, is just starting to achieve its full potential. Although cloud computing already creates concrete, new end-user experiences with online storage services by several vendors, it has not take off as a widespread enterprise computing paradigm. Nevertheless, organization-level cloud computing is already reality, as proven by the UK government cloud G-Cloud. Other recent initiatives like the Deutsche Börse Cloud Exchange promise a yet unforeseen potential of cloud computing as an enterprise computing paradigm. The Deutsche Börse Cloud Exchange is a stock exchange market place for IT services that has been launched in June 2013, technologically, it is an IT services broker plus provisioning services. Cloud computing introduces the elastic and site-transparent distribution of computing platforms and applications over outsourced data center infrastructures. Important IT vendors have positioned themselves strategically with concrete products: the Google Cloud, the Microsoft Cloud, Amazon Web Services. Also, the on-demand data centers recently built by companies like IBM and HP must be considered as a form of cloud computing.

The rapid evolution of distributed computing technologies has no doubt provided a rich set of platforms and paradigms for building robust enterprise systems. Unfortunately, this rapid technological evolution has also left us with a large number of unresolved problems. The first is a trail of confusion and inconsistency in the concepts used to build distributed systems. Even within individual paradigms there are inconsistent interpretations of some of the most basic concepts, such as whether services are, or should be, stateless. Furthermore, between the different paradigms there is little consensus about the core ingredients of distributed systems; for example, what are the differences and relationships between components, services and objects.

Yet another problem is that the evolution of the different distributed system technologies has been overwhelmingly driven by the server-side. As a result, developers of

regular client applications, business processes or mobile applications such as Android and iPhone "apps" typically have to access server-side assets via low-level, platform-specific abstractions optimized for solving server-side problems rather than via abstractions that fit their needs. It is crucial to understand that this problem, which we would like to call a service consumability problem, forms a real hurdle for scalable, next-generation distributed software development. The importance of addressing this problem has grown as client applications have become more visible and as they have started to play a major role in the perceived usability of service-based systems. Furthermore, in large enterprise system landscapes [AHVE07] services are often clients of each other. The experience has shown that the most successful enterprises have paid particularly attention to the infrastructure features needed to support flexible service usage [Haa05, Dra10]. To date, however, driven by legacy system integration challenges, SOA best practices have primarily focused on only one aspect of usability, i.e., the straightforward and rapid integration of diverse technologies and platforms at the server-side. Unfortunately, this ease-of integration at the server side has come at the price of lower flexibility and ease-of-use of distributed software assets at the client side in client application development.

To address the problems of service consumability and to reduce the artificial complexity involved in building client applications there is a general need for a unified conceptual model of service-oriented computing. Such conceptual model should support the needs of client developers as well as service providers. Our premise is that such a conceptual model should be independent of, and implementable on top of, the several different service-oriented computed paradigms discussed above. Such a conceptual model should also be compatible with the different implementation technologies and modeling languages used to realize client applications today, such as high-level programming languages or business process modeling languages. Just as high-level programming languages are optimized for human programmers rather than realization platforms, we believe that the features of such a conceptual model should be determined by what is best for client developers rather than by the idiosyncrasies of individual implementation platforms. Using the terminology of the model-driven development we refer to this as a client-oriented platform independent model (CPIM). Although the approach focuses on traditional client/server systems where an object plays only one role, i.e., either client or server, the approach is also applicable in more general situations in which servers are also clients of other servers. In short, any object acting as a client benefits from the client-oriented view of their servers afforded by the approach.

In this article, we present a concrete set of metamodels for the various views and abstraction levels involved in a service-oriented computing landscape and position them within a single unified framework. There are two key difference between our framework and other more general conceptual models for distributed computing like RM-ODP [LMTV11], Service Component Architecture (SCA) [Edw14]or the CORBA Component Model (CCM). First, our framework focuses on service-oriented computing, and aims to accommodate the mainstream service-oriented computing platforms and paradigms as special cases of a more general, but nevertheless still service-oriented, modeling approach. Second, our framework emphasizes the needs of client-developers rather than the traditional server-side concerns that dominate distributed computing platforms, i.e., persistency, integration, interoperability, transactions, robustness, etc.. For client developers, these concerns in general manifest themselves as non-functional properties that are part of service-level agreements, i.e.,

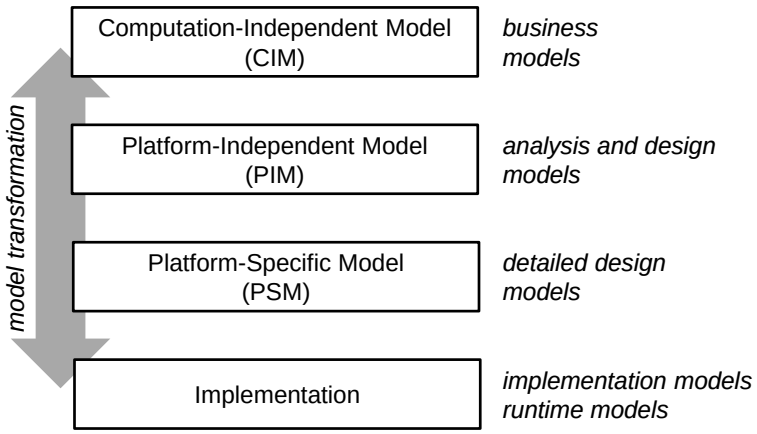

Figure 1 – The abstraction levels of the OMG MDA terminology.

performance, reliability etc.

Section 2 presents the overall structure of the framework. In Sect. 3, we present the core metamodel. In Sect. 4, we present the metamodels for the platform-independent views of a service-oriented system. In Sect. 5 we describe five key PIM-level realization patterns that define bi-directional transformations of client-side abstractions into server-side abstractions. In Sect. 6 we explain the relevance of the introduced foundational patterns to the OSM (Orthographic Software Modeling) approach. Section 7 serves as a proof of concept by showing how the model can be applied to particular realization platforms, i.e., Java for the client-side and Web Services for the server-side. In Sect. 8 we present a domain-specific example scenario from the banking domain. We discuss related work in Sect. 9. Then, finally, we give a conclusion and final remarks in Sect. 10.

2 Structure of the Conceptual Model

The basic goal behind the unified conceptual model is to provide a framework in which the concerns and viewpoints relevant for building client applications using distributed services can be expressed and related to one another [BAD11]. The two basic concerns that are used to structure the framework are the level of abstraction at which a system is represented and the roles from which the components of a system are viewed.

2.1 Abstraction Levels and Roles

Figure 1 shows the levels of abstraction recognized in the model-driven architecture (MDA) approach popularized by the OMG [Sol03]. The most abstract level at the top is the Computation-Independent Model (CIM) in which models of the business processes to be automated or the environment to be enhanced are described independently of the envisaged IT-based solution. On the next level, a Platform-Independent Model (PIM) describes the key ingredients and the behavior of the envisaged system independently of the idiosyncrasies of specific platforms. It is the responsibility of a Platform-Specific Model (PSM) to consider these idiosyncrasies. A PSM describes the

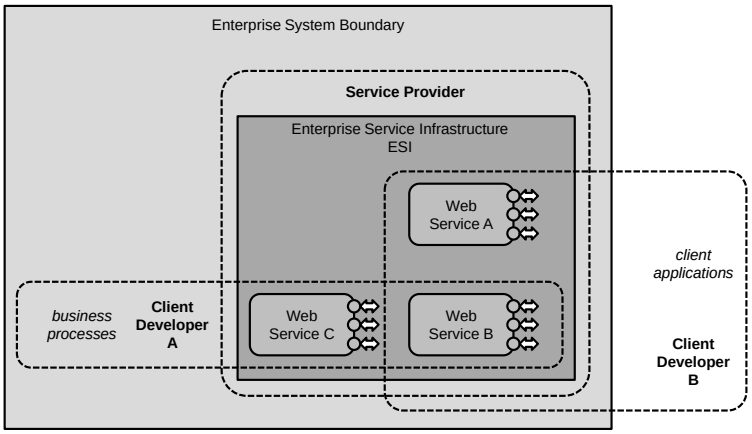

Figure 2 – An example distributed application development scenario.

system in terms of the concepts supported by a specific platform, but not necessarily in a way that is directly executable. The lowest level in the MDA is the implementation which can be executed without further mapping into more concrete forms. Our approach adopts the basic MDA abstraction levels and the associated terminology. However, we focus only on the two central levels since this is where the key separation of concerns and identification of common abstractions takes place. The CIM is not relevant since the client/server distinction does not appear until the PSM level. To the extent that they appear at all in CIMs, computer-based systems always appear as unified black boxes. The implementation level is not relevant either, since it is already well populated with a large range of implementation technologies and it is not the intention of this article to introduce more.

As the different technologies of distributed computing have evolved from basic client/server approaches to service-oriented architectures the differences between the concerns of client developers and the concerns of service providers have grown. Moreover, these are set to increase even further as distributed computing evolves further into cloud computing. The two fundamental roles of concern are therefore the *client developer* and the *service provider*. Figure 2 shows an enterprise service infrastructure (ESI), implemented and maintained by one service provider and used by two different client developers. A fundamental premise of our approach is that there is a clear boundary to the enterprise system for each stakeholder. The different stakeholders do not necessarily need to agree on the boundary, but they must each have a clear notion of the boundary of the system. In Fig. 2, the services in the ESI are implemented and maintained by one service provider but are used by two distinct client developers. One of them, client developer A, actually belongs to the organization owning the ESI, and thus is represented as being inside the enterprise system boundary, while the other, client developer B, is completely outside the enterprise boundary and uses the services as a customer.

2.2 Abstraction Levels vs. Client-and Service-Oriented Views

The overall structure of the unified conceptual model is derived by regarding the abstraction level dichotomy (PIM/PSM) and the dichotomy of the presented roles as

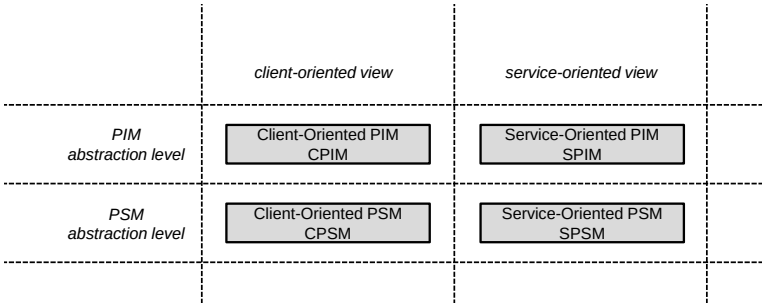

Figure 3 – Two-dimensional model space of the unified conceptual framework.

distinct, orthogonal dimensions. The basic goal is to address the concerns of both of the roles at the two abstraction levels as illustrated in Fig. 3 resulting in four different models that will be explained in the following.

The role of a service-oriented PIM (SPIM) is to provide a platform-independent view of a service landscape from the point of view of a service provider. There is therefore no notion of, or support for, clients in the SPIM. The abstractions used in an SPIM are service-oriented, but transcend particular realization technologies. If there are numerous service providers supporting different parts of a single overall landscape, they will have SPIMs tailored to their own particular view of the landscape. The role of a client-oriented PIM (CPIM) is to provide a platform-independent view of a service landscape tailored to the needs of a particular client developer. A CPIM therefore includes the notion of, and support for, clients. However, it also includes abstractions of server-side software entities that a client type wishes to use and access. As with an SPIM, a CPIM transcends particular realization technologies and represents remote software entities independently of their realization.

A service-oriented PSM (SPSM) provides a platform-specific representation of the service landscape from the point of view of a service provider. In terms of the traditional MDA paradigm it extends and refines the SPIM using platform-specific abstractions. A client-oriented PSM (CPSM) has the same relationship to a CPIM as an SPSM to an SPIM, representing a refinement of a platform-independent model that adds detail through platform-specific abstractions. Ideally, this platform-specific detail will only relate to the client-side because the server-side abstractions should be accessible exclusively through PIM level abstractions.

The two-dimensional modeling framework as presented in Fig. 3 provides four different views each tailored to the respective role and level of abstraction involved. However, not all views are of interest to all the roles. Each stakeholder has a particular constellation of views that reflect his particular concerns depending on the role that he plays, as illustrated in the example in Fig. 4, which is related to the example scenario in Fig. 2. Figure 4 shows four kinds of models used by the different kinds of stakeholders. These models are not independent, of course, but are related to one another in carefully defined ways. In fact, we believe that one of the main contributions of the unified conceptual framework is to capture the commonalities and differences between these models. A service provider is usually only interested in the server-side abstractions, and since in our example depicted in Fig. 2 there is only one service provider responsible for the whole ESI, all services are included in this service provider's SPIM and SPSM views. The SPSM views of the service provider

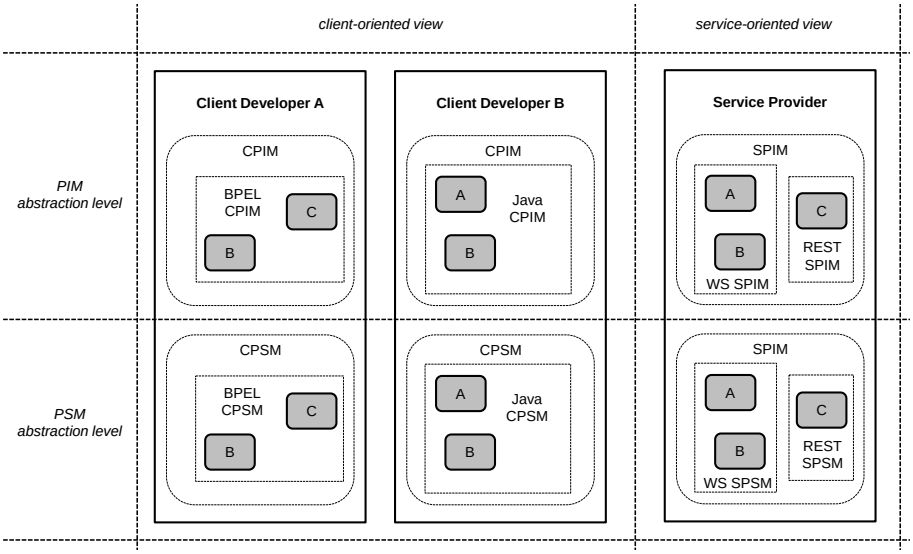

Figure 4 – An example of concrete view types.

are a Web Services technology view and a REST technology view in this example. Client developer A on the other hand, is only interested in his own client-oriented view of the ESI. This is reflected in his CPIM and CPSM models which only contain abstractions of the services he uses - services B and C, which belong to a BPEL (Business Process Execution Language) view in this case. Similarly, client developer B needs client-oriented views tailored to his concerns. His CPIM and CPSM models therefore only contain abstractions of services A and B, which belong to a Java view in our example.

3 The Core Metamodel

The Core Metamodel (CM) provides the abstractions that span the main roles and abstractions levels in service-oriented computing identified in Sect. 2. All of the other metamodels inherit directly or indirectly from the CM. At its heart, therefore, are the basic ingredients of 3GL programming languages (third generation programming languages), i.e., data types, processes and objects. The history of software system descriptions, in general, and the the history of programming languages in particular [Par85], can in fact be characterized in terms of the evolution in the way that these ingredients have been interwoven. In the early days of computing up to the late 70's and 80's, programs and systems were primarily described in a function-oriented way. This was based on the principle of strictly separating the *functions* in a program from the *data types* that they manipulated, and using the relationships between the former to define the architecture of the system. From the late 70's, the wisdom of this separation started to fall into question and the notion of object-oriented programming emerged based on the idea that functions and data should be tightly bound together and encapsulated. Since objects are essentially data-centric, this meant that data played the dominant role in defining the architecture of a program.

The Core Metamodel consists of ten abstractions arranged as classes within a

single hierarchy – see Fig. 5. The class Entity is an abstract class that serves as the root of the inheritance hierarchy. Instances of subclasses of the class Entity may therefore represent any kind of entity within a computing system. A Data Type is essentially a set of values whose use is controlled by a number of rules. The instances of a Data Type are values that do not have inherent identity of their own. Data Type is an abstract class that has two subclasses – Primitive and Compound Data Type. Primitive Data Types are used to represent the classic data types such as Integer, Character, String, etc. Compound Data Type are data types whose values are composed of combinations of Primitive and/or Compound Data Types, similar to *records* or *structs* as used in programming languages like Pascal or C.

A Process represents a basic functional abstraction that includes a set of steps arranged in some well-defined order to achieve some effect or to reach some goal. It encompasses programming level abstractions such as subroutines, functions, procedures and methods, as well as high-level notions of processes such as workflows and business processes. Since these involve the sequential execution of sub-steps to achieve a goal, processes have an associated notion of procedural or *algorithmic state* [AB09] that represents its current progress through the designated sequence of steps. Processes can have input and output parameters which can be objects or data types and can also use other processes. The class Process is partitioned by the subclasses Free Process and Allocated Process. A Free Process is a process that is not allocated to any particular object. It therefore represents a purely functional abstraction, akin to a function, procedure or subroutine in older programming languages. An Allocated Process is a process that belongs to an object. The existence of an instance of an Allocated Process is therefore tied to the existence of the object that it belongs to and it cannot be used independently of that object. Allocated Processes have direct access to the cohesive state of the object they belong to. They therefore correspond to methods in object-oriented programming languages.

An Object represents the basic object abstraction familiar from object-oriented programs and object-based computing in general. The basic characteristic of an object is that it encapsulates one or more attributes of a Data Type behind one or more Allocated Processes, i.e., methods or operations. Like objects from object-oriented programming they have their own unique identity and can be duplicated. Objects therefore unify the two core ingredients of function-oriented computing – processes and data types. The key additional idea for service-oriented computing is that objects come in two basic kinds – Ephemeral Objects and Architectural Objects. Because they encapsulate data values, objects have an associated notion of *cohesive state* [AB09], which captures the effects of the operations that have been applied to the object. Although Processes only possess algorithmic state in our approach, Allocated Processes can also participate in maintaining the cohesive state of the Object they belong to in the sense that they update the data storing that state in a consistent way. However, they do not have to. The difference corresponds to the distinction between inspector and read-only operations in object-oriented programming. Free Processes play no role in maintaining cohesive state because they do not belong to an Object.

Architectural Objects are stable objects whose lifetime normally coincides with the lifetime of the system as a whole, because they are the *components* of the system. The only exception is when architectural reconfigurations are performed. Architectural Objects are generally large, behavior-rich objects and are usually few in number. In fact, most of the time there is only one instance of them in a system. The notions of components, services and distributed objects found in contemporary

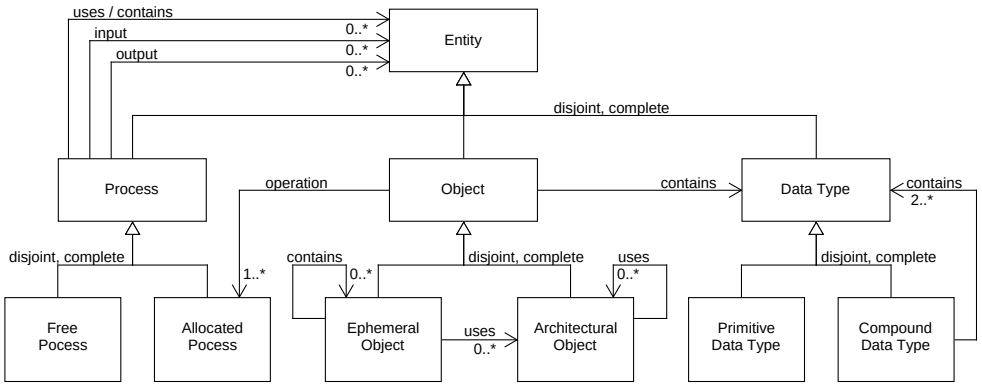

Figure 5 – The Core Metamodel.

distributed computing technologies are encompassed by the notion of architectural objects. Ephemeral Objects are objects which have a temporary lifetime with respect to the lifetime of the complete system. In other words, they are frequently generated and deleted during the lifetime of a system and essentially correspond to data in information systems.

The advent of the object abstraction in the late 70s led to the object-oriented revolution that still dominates many programming languages today. In contrast, the concepts and best practices of service-oriented architectures can be understood as a move away from object-orientation back to a more traditional function-oriented style of organizing data and processes. However, confusion surrounding SOA and its relationship to other paradigms demonstrates that SOA's conceptual model is not optimal for all aspects of distributed computing. The model proposed in this article can be viewed as an attempt to introduce a more sophisticated, unified approach that can accommodate both function-oriented and object-oriented viewpoints depending on what is optimal for the different roles involved.

4 Platform Independent Metamodels

4.1 Service-Oriented PIM Metamodel

The SPIM Metamodel extends the Core Metamodel with general, platform-independent concepts for the server-side perspective of the client/server divide. The basic extensions are the introduction of further subclasses of the classes Ephemeral Object and Architectural Object. As shown in Fig. 6, the Ephemeral Object class is divided into two subclasses, Data Object and Behaviour Rich Ephemeral Object. The first of these represents a type that is purely a wrapper for sets of attributes. In other words, Data Objects essentially wrap up and encapsulate the data contained in a Compound Data Type by shielding the attributes from direct access via setter and getter operations. They are therefore very similar to DTOs of the kind supported in J2EE. By definition, Data Objects therefore only contain create, read, update and delete (CRUD) operations and provide no "rich" behaviour beyond the getter and setter operations. This is what basically distinguishes a Data Object from a Behaviour Rich Ephemeral Object. The latter kind of object type provides extra functionality in the form of

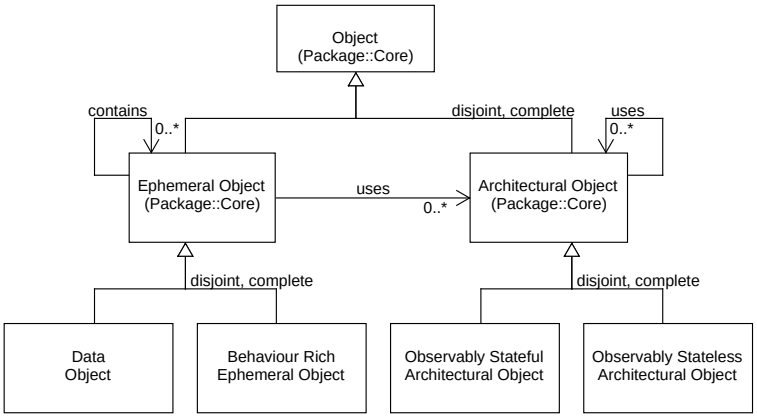

Figure 6 – SPIM – Service-Oriented PIM Metamodel.

methods that calculate new information that is not directly stored in the attributes. This means that these can no longer be regarded as mere Data Object types.

The Architectural Object class from the Core Metamodel is also divided into two subclasses – Observably Stateful Architectural Objects and Observably Stateless Architectural Objects. These capture the distinction between observably stateful and observably stateless services [AB09]. Observably Stateful Architectural Objects are objects that have a cohesive state from the point of view of a client. In concrete terms this means that the methods of the object exhibit different behaviour for the same set of explicit input parameters depending on the cohesive state of the object. In contrast, Observably Stateless Architectural Objects have no observable cohesive state.

4.2 Client-Oriented PIM Metamodel

The CPIM Metamodel extends the Core Metamodel with the general, platform-independent concepts for the client perspective of the client/server divide. As with the SPIM, the main extensions are applied to the Ephemeral Object and Architectural Object class of the Core Metamodel, but the Process class is also extended to include the notion of clients that are in general processes, so called Client Processes. The specializations of the Ephemeral Object and Architectural Object classes reflect the different client-oriented properties [AB09]. Basically, Ephemeral Objects are dynamic objects while Architectural Objects are static objects.

As shown in Fig. 7, the class Ephemeral Object is divided into two subclasses – Private Ephemeral Object and Shared Ephemeral Object. This reflects whether or not instances of a class can be shared by various client instances of a client type or whether they remain private to a given client type instance.

The two subclasses of Architectural Object – Observably Stateful Architectural Object and Observably Stateless Architectural Object – are both further divided into two subclasses depending on whether they are shared or private similarly to Ephemeral Objects. In fact, this is orthogonal to the property observably stateful/stateless, and could also have been modeled as a separate generalization set.

A further extension to the PIM is the addition of the Client Application specializa-

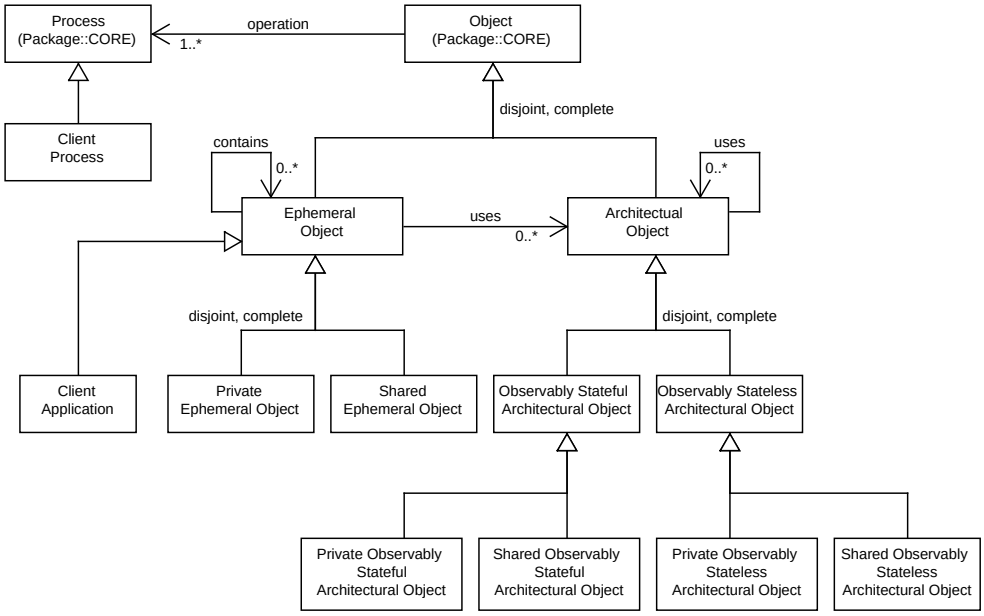

Figure 7 – CPIM – Client-Oriented PIM Metamodel.

tion of Ephemeral Object to represent clients that are applications in the traditional software engineering sense. These are a form of Ephemeral Object because they have an identity, a cohesive state and a lifetime that is often much shorter than the lifetime of the system that manipulates them.

5 PIM-Level Model Transformation Patterns

In this section we elaborate a series of core strategies used to switch between service-oriented views onto a system. We do so by defining transformations of entities of one view into entities of another view. The defined transformations map between one perspective of an abstraction to another perspective of an abstraction. The patterns are defined at the PIM level, however, they are intended to be exploited at the PSM level, i.e., they are intended to be applied to specializations of the affected abstraction as a concrete, practical realization step. The patterns are designed to be performed automatically. They are meant to be built into emerging technologies for writing client applications. Then, if a service is implemented in one way, the appropriate pattern can be automatically applied, as desired, to provide the client with an alternative view, and vice versa. The patterns are not necessarily applicable in isolation and they establish model transformation principles rather than complete solutions. The are intended to help reduce the levels of confusion and accidental complexity that exists in SOA development today. In particular, we hope that they will become part of the vocabulary of service-oriented computing. The transformations are reversible, i.e., they can be applied back and forth. However, the names of the pattern reflect one particular direction.

The five patterns that we describe are: the data type reification pattern, the

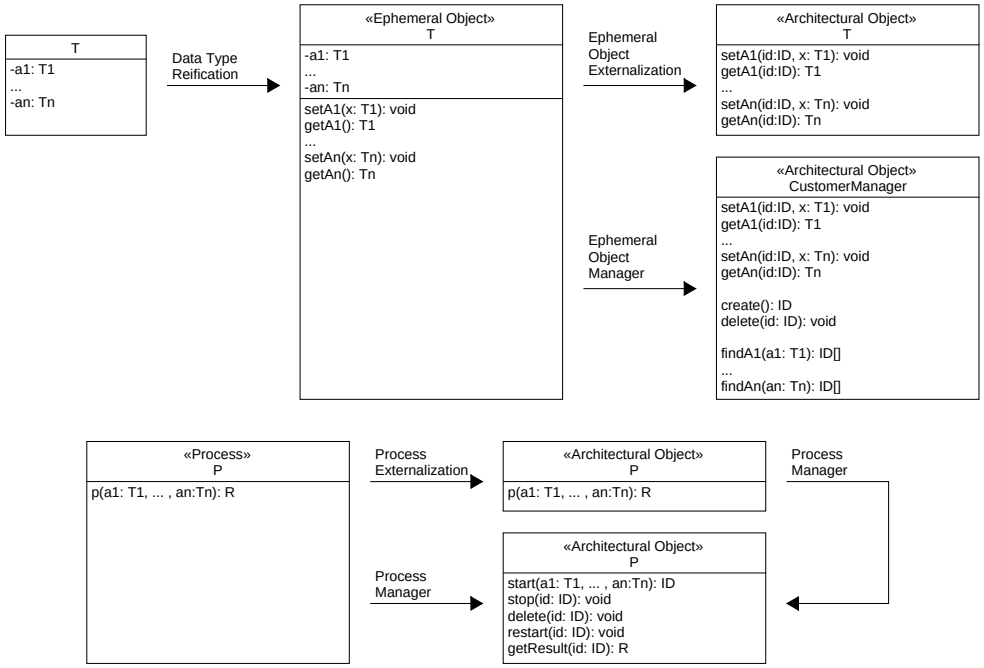

Figure 8 – Foundational MDA patterns.

ephemeral object externalization pattern, the ephemeral object manager pattern, the process externalization pattern and the process manager pattern. All the five patterns are summarized in Fig. 8 in an abstract manner. In Fig. 9 we have also given example application of instances of one of the patterns, i.e., the ephemeral object manager pattern, to a concrete expert domain class, i.e., a customer having a name and an address as attributes.

5.1 Data Type Reification

The data type reification pattern maps a Data Type to an Ephemeral Object. Essentially, it reifies a plain Data Type whose instances represent collections of values, into an Ephemeral Object type. The resulting Ephemeral Object can store the corresponding record values, however, as private attributes that can only be accessed by appropriate setter and getter operations. In non-technical terms, the pattern turns a plain record into a full-fledged object, i.e., it enriches the record by appropriate methods. For example, if we are working with Web Service technologies, the Data Type pattern allows for an object-oriented view of the structures that are communicated over the internet in document formats such as XML. Basically, the pattern corresponds to the notion of DTOs (Data Transfer Object) in J2EE [KtET00]. We have called the Data Type pattern a reification pattern, because it is an intuitive term commonly used in the generative programming language community [CE00]. In the strict sense of reflective programming terminology, the application of this pattern consists of a reification step and a reflection step. First, we inspect the data type, which amounts to a reification step. Next, we exploit the information that we

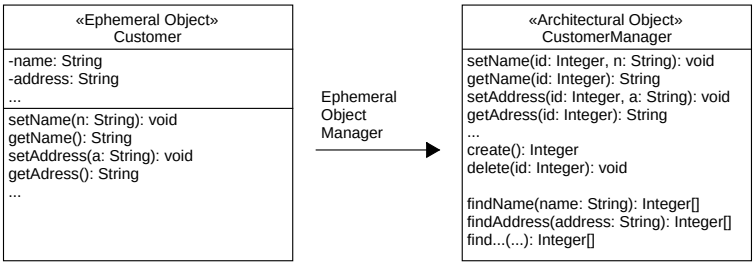

Figure 9 – Example application of foundational MDA patterns.

have gained about this data type to craft an Ephemeral Object, which amounts to a reflection step.

5.2 Ephemeral Object Externalization

The ephemeral object externalization patterns maps an Ephemeral Object type to an Architectural Object type – see Figs. 8 and 9. For a given Ephemeral Object type the pattern defines an Architectural Object type that supports the same operations, but the parameter list of each operation is extended with an additional object identifying parameter, i.e., the additional parameter holds the identifier of an instance of the Ephemeral Object class. This extra information is needed because a single instance of an Architectural Object type is responsible for storing the state of all instances of the Ephemeral Object type, and these need to be distinguishable. In a sense, therefore, this pattern delegates the responsibility for the cohesive state and functionality of an Ephemeral Object type to an Architectural Object. Figure 10 shows the basic way in which the externalization pattern is used to map between an ephemeral object type with behaviour and one architectural object that manages instances of the ephemeral object type and, furthermore, contains the operations that have been previously assigned to the ephemeral object type. We can further distinguish between two different variants of this pattern that we will refer to as light ephemeral object externalization and full ephemeral object externalization. For the latter kind, we consider the externalization of all operations of an Ephemeral Object, while for the former kind only the behaviour-rich operations are externalized to an Architectural Object and simple getter- and setter-operations are excluded.

5.3 Ephemeral Object Manager

The ephemeral object manager pattern is, essentially, an extension of the ephemeral object externalization pattern in Sect. 5.2. It also maps an Ephemeral Object type to an Architectural Object type. As well as mapping the operations of an Ephemeral Object type to the Architectural Object, with the extra parameters and/or return values, as illustrated in Fig. 8, it also adds creation and deletion operations to complete the CRUD functionality, as well as search operations to find instances of the Ephemeral Object based on one or more of its attributes. Again the ephemeral object manager pattern comes in full and light variants.

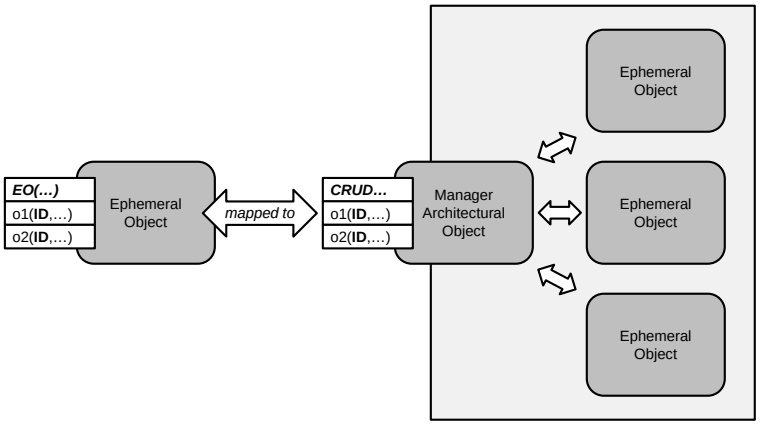

Figure 10 – Ephemeral object externalization.

5.4 Process Externalization

The process externalization pattern maps a Process type to an Architectural Object type – see Fig. 8. A single Process entity, with a given set of input parameters and/or result is mapped to an Architectural Object type with a corresponding re-entrant operation. Any client wishing to invoke the process therefore simply calls this operation on the singleton instance of this type. The inner working of the operation is completely invisible to the clients of the process. In a sense, therefore, this pattern delegates responsibility for the functionality and algorithmic state of multiple instances of a given Process type to a single instance of an Architectural Object type.

5.5 Process Manager

Like the process externalization pattern in Sect. 5.4, the process manager pattern also maps a Process type to an Architectural Object type. The difference is that, like the ephemeral object manager pattern, this pattern provides CRUD operations to create, manipulate and destroy process instances. As illustrated in Fig. 8, this requires individual process instances to be identified by a unique identifier (ID) that is passed to, or returned by, invocations of the CRUD operations. While the Ephemeral Object patterns handle the cohesive state of instances of the Ephemeral Objects, Process patterns handle the algorithmic state of Processes. The process manager pattern can also be interpreted as transforming an Architectectural Object that represents an externalized process – Fig. 8.

5.6 Technology Independency of the Patterns

The patterns presented in Sects 5.1 to 5.5 capture some of the strategic design decisions that have to be made to realize service-oriented systems, regardless of the specific technology that is used to implement them. They are therefore strategic platform-independent decisions. It is also important to note that clients are processes or objects, depending on their exact nature. Thus, when the appropriate manager pattern is applied, the algorithmic or cohesive state that is being maintained by the

architectural object is the state of the client – in other words, the state of the conversation between the client and an instance of an Architectural Object type. Such conversations are often called sessions and the corresponding identifiers are called session IDs. The model transformation patterns in this section are basic realization patterns which relate PIM level abstractions to other PIM level abstractions. We use the term "pattern" in the sense of the GRASP (General Responsibility Assignment Software Patterns) of Larman [Lar04] rather than the GoF (Gang of Four) design patterns [Gea95]. Like the GRASP patterns, the patterns that we have presented in this section are oriented towards supporting the transition from PIM level models to PSM level models. Like the GRASP patterns, they also represent mapping principles and options rather than descriptions of reusable design structures a la GoF patterns.

6 Integrating the MDA SOA Patterns into the OSM-SUM

The transformational patterns developed in Sect. 5 perfectly fit to, and integrate with, the OSM (Orthographic Software Modeling) vision aimed at overcoming essential frictions and tensions faced in today's software engineering tool landscapes. Today's software engineering projects suffer a high degree of artificial complexity due to the plethora of artifacts and technologies generating multiple redundancies and inconsistencies as a huge legacy problem. Although the problem is not visible in small agile projects, in the future we will need large and even very large projects to solve some of the open information system issues. The larger the projects, the larger this problem, and we experience a point beyond which projects do not scale in today's software engineering. Despite many initiatives in the last two decades, the dream of adequately supporting highly heterogeneous, diverse, distributed teams has not yet been realized.

Orthographic Software Modeling, see, e.g., [AS08, Atk14], which has its roots in the KorBa methodology [Atk02] is about completely rethinking the role of software artifacts, the way software artifacts are generated and maintained and, in particular, the relationship between different kinds of software. Orthographic Software Modeling is about the complete integration of all software artifacts, from scratch, based on the notion of the SUM (Single Underlying Model) – see Fig. 11. In the SUM all the different kinds of artifacts typically used in today's software engineering life cycle are integrated into one, single conglomerate of modeling elements. The range of artifacts we are talking about is exhaustive; it encompasses all kinds of platform independent models as well as all kinds of platform-specific models and beyond that, way more kinds of artifacts from all the realms of the software engineering life cycle: configuration management, version management, systematic test management, project management, and so on and so forth. The SUM is not a mere thought experiment, we see it as a real option that is up to now hindered merely by the legacy problem of our current software engineering artifact base coupled with a certain concrete-syntax oriented attitude that we are used to in practice in today's software engineering. Of course, the SUM does not come for free. It will take a lot of effort and discipline to build it. Furthermore it must be accompanied by a series of key success factors, that we have summarized, e.g., in [AD13]. We only name a few of these key success factors here: a notion of deep standardization, perfectly maintained meta-information, an innovative group moderation process, a focus shift towards management best practices and so forth.

Once the SUM becomes reality, we can hope for a substantial increase in scale

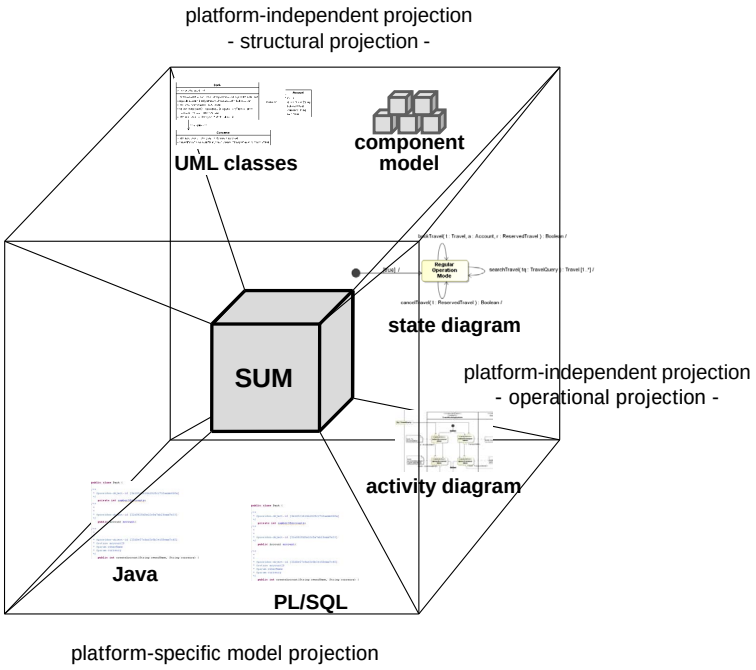

Figure 11 – Example cutout the SUM (Single Underlying Model).

and speed in software engineering projects. With the SUM the old dream of complete artifact traceability comes for free. The SUM is the enabler for Orthographic Software Modeling. With OSM, all the artifacts that we are used to become views onto the SUM. Therefore, OSM can be explained as a paradigm shift from model transformation to view transformation [ASB09, Atk14]. OSM is also a paradigm shift form a context-oriented perspective to an abstract syntax-oriented perspective of software engineering, which matches the data modeling approach of form-oriented analysis [DW04].

In Sect. 2.2 we have established a two-dimensional modeling space for the domain of service-oriented software systems engineering – see Fig. 3. With the OSM approach each of the four areas in Fig. 3, i.e., the Client-Oriented PIM, the Service-Oriented PIM, the Client-Oriented PSM and the Service-Oriented PSM all become views onto a single implementation base – the SUM. Similarly, each strand of each of the two discussed dimensions, i.e., the abstraction dimension and the view dimension can be considered OSM views onto the SUM. Here, we must not be confused that we have named only one of the dimension a view dimension, and the other one an abstraction level dimension – this is merely a property of the concrete modeling domain, i.e., the domain of service-oriented software systems engineering. With an appropriate tool views and sub-views onto the SUM can be maintained. For example, the CPIM and the SPIM can be considered sub views of the PIM abstraction-level view. Now, an appropriate tool could support drill-down, roll-up and dicing operations on the SUM and this way turning the SUM into a kind of model warehouse. Once we have grasped the level, views and models in Fig. 3 as OSM views, we can understand the transformational patterns in Sect. 5 as view transformations. This is where the

concepts elaborated in this article tie in with the essence of the OSM approach – the paradigm shift from model to view transformation.

7 Example SPSM Models

In this section we provide two examples of SPSM models for specific client- and server-side implementation technologies. We consider the two most widely known service-oriented system realization technologies – Web Services for the server-side and Java for the client side.

7.1 Web Service SPSM Metamodel

The first generation of Web Service standards, i.e., WSDL (Web Service Description Language) and SOAP (SOA Protocol), and several extensions of these represent one of the most well-known and widely used service realization technologies. The definition of the Web Service SPSM is given in Fig. 12. Not all of the concepts of the SPIM are directly supported, so the Web Service SPSM actually restricts the use of some of the concepts in the SPIM. The SPSM includes an indication of where the SPIM abstraction has a different name in the specific realization technology. This is achieved by placing the platform specific name in parentheses after the platform-independent name in Fig. 12. This renaming is most evident in the case of Data Types. Web Services basically support both kinds of Data Types as XML data types, i.e., simple XML data Types as Primitive Data Types and complex XML data types as Compound Data Types. The SPSM also indicates that Architectural Objects are referred to as services in Web Service technology. Both forms of Architectural Objects are also supported, but Observably Stateful Architectural Objects are further divided into two subclasses, Inherently Stateful Architectural Objects and Inherently Stateless Architectural Objects. This distinction reflects the fact that Architectural Objects (i.e. Web Services) that appear to be stateful to clients (i.e. that are observably stateful) may not in fact have any direct state of their own from an implementation point of view, but may delegate the storage of this state to a third party (e.g. a database) that is not visible to the client. Such types of Architectural Objects are referred to as Inherently Stateless Architectural Objects. In contrast, an instance of the type Observably Stateful Architectural Object may indeed encapsulate the state that it exposes to clients. As discussed in [AB09] the failure to distinguish between these two properties (observable versus inherent) statefulness is the root of much of the confusion surrounding this issue in contemporary service-oriented technologies. By providing an explicit model of the distinctions and relationship between these two concepts and allowing Architectural Objects to be characterized accordingly, the state-related behaviour of services can be much more clearly understood by client developers and service providers alike.

The restriction to the SPIM relates to the distinction between Allocated Process and Free Processes. The Web Service standard does not recognize the concept of free processes and requires all processes to be allocated to objects. The Web Service SPSM Metamodel in Fig. 12 therefore indicates that an Allocated Process corresponds to a Web Service operation.

This Web Services SPSM is not a single particular SPSM, rather, it stands for a certain service-oriented paradigm that became popular with concrete Web Services technology and revolves around the vague notion of statelessness that is now usu-

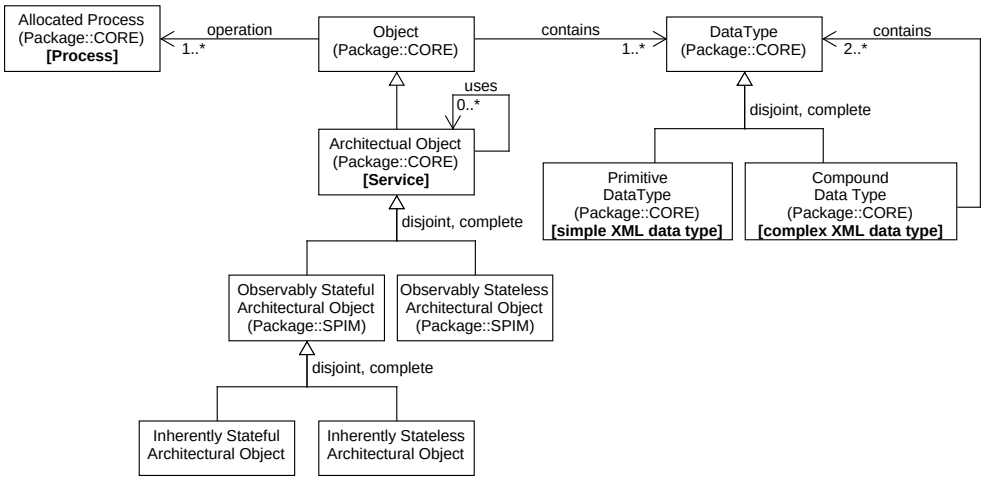

Figure 12 – Web Service SPSM metamodel.

ally associated with the name SOA (service-oriented architecture). Actually, there are concrete system architectures, both in industry and academia that were called service-based, service-oriented, service buses, or even service-oriented architectures, way before the SOA acronym gained widespread acknowledgement and usage. Even more importantly, these service-oriented architectures often made use of full-fledged object-oriented technology like CORBA and also exploited all the state-oriented features that are associated with object-oriented technology. This section aims to clarify the resulting terminological overload and identify the essential differences between full object technology and SOA technology in the narrower, latter sense.

Service-oriented architecture is not a single vision or paradigm. Depending on the particular community in which the term is used it can mean anything from an enterprise architecture to an inter-enterprise architecture, though the acronym SOA is often identified with the concrete web service technology stack and a certain mix of architectural principles that emerged in the early 2000s. Service-oriented architecture started as a three-tier hub-and-spoke architecture for enterprise applications [FT02, KtET00, Ber96]. After the B2C hype it then became the carrier for a lightweight B2B alternative for EDI [Emm93]. Another strand of SOA focusses on the massive and systematic reuse of software components, both enterprise-wide, as so-called SOA governance, and world-wide [ABHS07]. In the past there has been lot of confusion about the concept of statelessness of services. To avoid this confusion it is important to cleanly separate the conceptual level from the implementation level. At the implementation level the question of state refers only to the decision about where to maintain state, e.g., on the application server or in the database. From an architectural perspective the concept of statelessness actually represents a programming language feature for the presentation of services to potential clients.

There is a direct tradeoff between the feature-richness of the service implementation technology platform and its usability for client developers. The less that is assumed about programming language concepts and features, the simpler the mapping between the concepts of a client programming language and the concepts of the service interface specification. A mapping is more than just a conceptual correspon-

dence between features, it embraces the complete realization technology involved in bridging between the service interface and the client code. For example, the IDL compilation and generation of stubs in the CORBA world belong to such a mapping. In general, the less you assume about the client programming language concepts and features, the easier it is for a client to exploit the service. Nevertheless, it is important for the framework to provide clean, standardized language bindings to the IDL, since this is the key to providing client programmers with a natural, tailored environment. Techniques developed in frameworks such as CORBA [Gro06] and ODP [LMTV11] are highly relevant here such as implicit binding.

A service facade can realize complex concepts like the construction and manipulation of objects, whereas the service facade itself is not complex. A service facade can be thought of as a remote control unit with knobs that support operations on the complex artifacts behind. Though the implementing service code does not have to be written in an object-based language, the structure of the overall system is object-based because it results in the conceptual realization of objects. All of this is related to the complexity of the exploited type system and the exploitation of the Ephemeral Object Manager pattern. The Architectural Object in the pattern can be written in a programming language that is not an object-oriented language. In fact, it could even be written in a basic programming language that does not support complex types. Pure SOA is an ideal based on the specification of interfaces that can be used immediately in any existing programming environment. In general, SOA tries to find the sweet spot between immediate and lightweight usability of a service and robustness of its usage. In practice, software system designers make use of mediating type systems like the .NET Common Type System. However, a mediating type system is already a compromise. The described tradeoff is similar to the one described by John K. Ousterhoust in his seminal paper on Scripting [Ous98], i.e., the trade-off between the "weight" of a programming language and the maintainability of its code artifacts.

7.2 Java CPSM Metamodel

The first platform-specific model for the client-oriented perspective that we consider in this section is the Java Standard Edition (Java SE) platform and its derivatives like the Java Micro Edition (Java ME) and also the Android platform for mobile devices. Java is one of the most widely used programming languages for writing client applications both in terms of regular GUI driven programs that run on laptops and PCs and more recently in terms of application for the rapidly expanding Android phone market. We therefore first consider the basic Java language metamodel to elaborate how the different kinds of abstractions specified in the CPIM can be supported (i.e. which Java language constructs are used to represent the different kinds of software entities) in the CPSM for Java. The Java language metamodel as presented in Fig. 13 represents the basic concepts used to express a model or software program in the Java language.

As can be seen when comparing the CPIM metamodel as defined in Sect. 4 with the CPSM metamodel for Java as depicted in Fig. 14 there are no meaningful structural differences between the two metamodels. However, the metamodel depicted in Fig. 14 specifies which kinds of entities can be mapped to the Java language constructs. These mappings can be summarized as the following equivalent pairs of elements:

- JavaMethod ↔ Process
- JavaClass ↔ Object

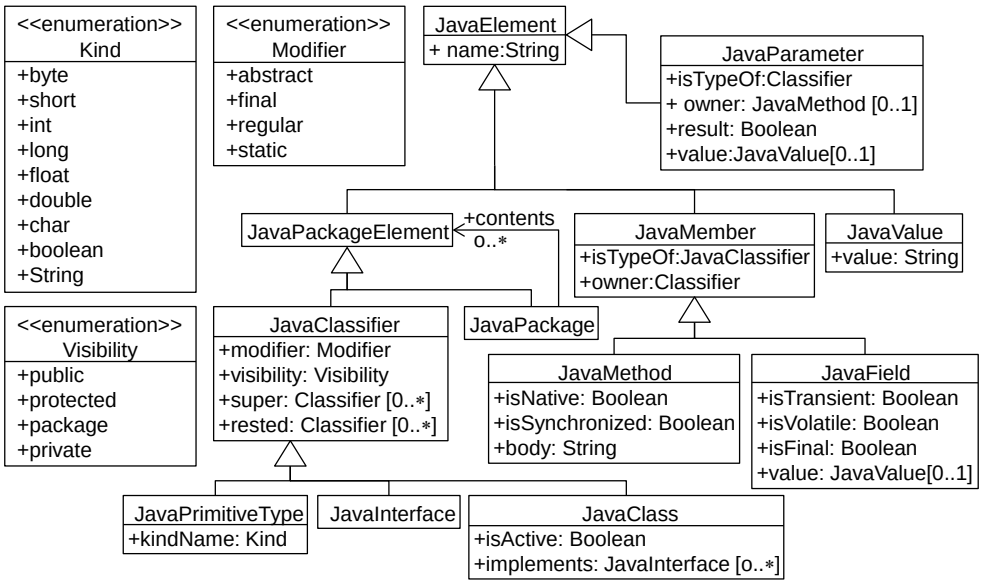

Figure 13 – Java language metamodel.

- JavaPrimitiveType ↔ Primitive Data Type
- regular JavaClass ↔ Ephemeral Object
- static JavaClass ↔ Architectural Object

As illustrated in Fig. 14, on the Java platform a Process entity can never stand-alone and always represents an entity that needs to be assigned to an Object entity with the role of an operation. In terms of the Java language constructs a Process entity is mapped to a Java Method element, while an Object entity type is always represented by the Java Class construct that may own one or more Java Method elements as its members.

On the next level, Ephemeral Objects can be represented in the Java language as regular Java Classes. We consider a regular class as an entity that can be instantiated dynamically and has at least one encapsulated entity and one Allocated Process to maintain and manipulate its internal state (i.e. they always represent stateful entities). In contrast, Architectural Object entities are characterized as either observably stateful or stateless as described in the CPIM in Sect. 4. In terms of the Java language constructs both kinds are represented by static Java Classes ready to be accessed at any time as long as the accessing client instance is running and alive. For a static Java Class, all its assigned Allocated Process entities (i.e. Java Methods) are also allocated with the Java modifier set to static.

As a mainstream object-oriented language, Java is able to support all of the concepts in the CPIM in a fairly direct way. The only two things from the CPIM that Java does not support are Compound Data Types and Free Processes since data composition and behavioral abstractions are achieved using objects. Ephemeral (i.e. dynamic) objects correspond directly to regular classes, while Architectural Objects correspond to static Java classes, since only one instance of them is required.

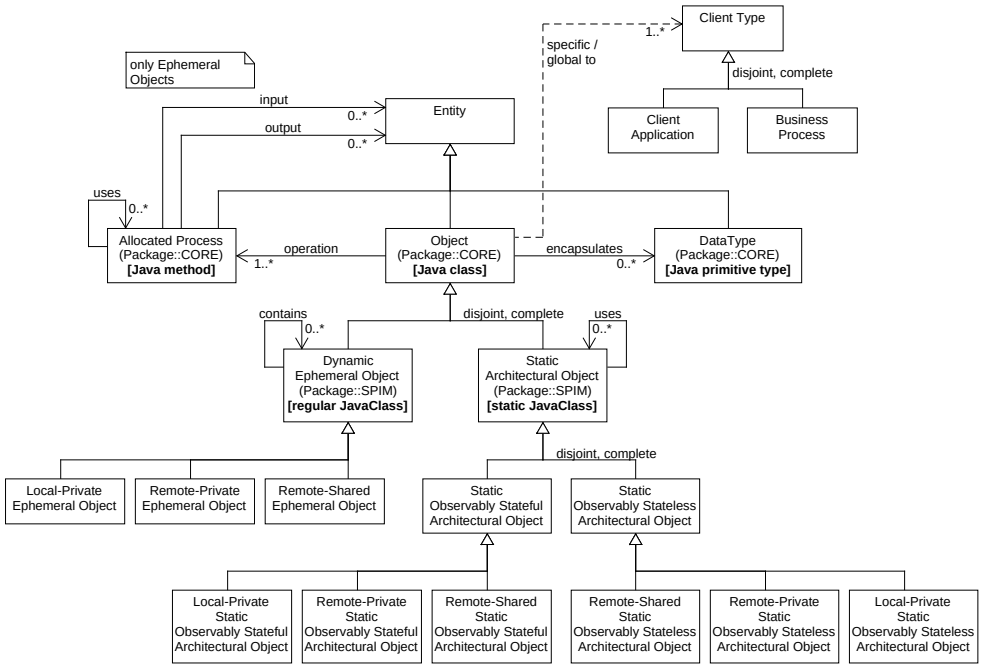

Figure 14 – Java language CPSM.

8 Example Scenario

Figure 15 illustrates a scenario involving two different client application types. The first client application (Salesman client) supports the salesman responsible for interacting with customers to create mortgage proposals that are electronically submitted for approval. A second client application (Financial Expert client) is used by the financial experts who are responsible for analyzing proposed mortgages and sending notifications to customers about the acceptance or rejection of mortgage proposals. As illustrated in the following models, this example scenario contains a typical mix of different kinds of objects and relationships.

8.1 PIM-level Models

In Fig. 16, a CPIM model which captures these types from the perspective of the Salesman client application type is presented. It consists of five classes that are of importance from the client type's perspective, three of them are of the type Ephemeral Object (EO) whose instances are created and deleted as the system operates, and two of them are of the type Architectural Object (AO) representing permanent instances. Furthermore, the stereotypes indicate that both Architectural Object types are observably stateful, but that one of them, the PriorityQueue type, is shared between multiple instances of different client types, while the other type, the Dictionary is private to an instance of the client type Salesman. Similarly, two of the Ephemeral Object types are shared, while one type is private.

Next we consider the CPIM for the Financial Expert client type, presented in Fig. 17. This indicates that there are only four classes of importance, three Ephemeral

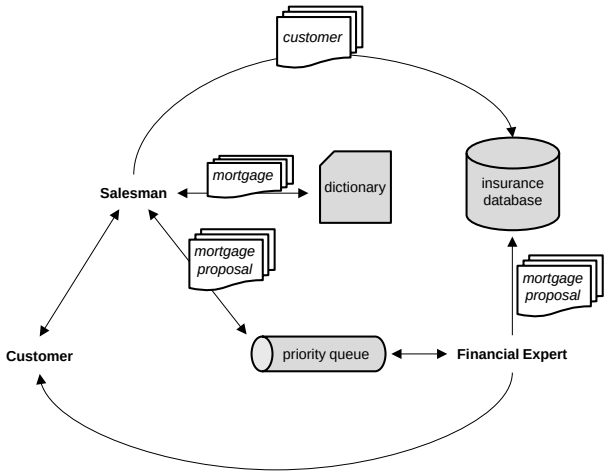

Figure 15 – Mortgage sales example.

Object types as well as one Architectural Object type. All but one of them, Notification, also appears in the Salesman client CPIM, with the same properties. The stereotype of this new class, Notification, therefore indicates that this is a private Ephemeral Object type.

After presenting the CPIM models for the Mortgage Sales scenario, we next introduce a SPIM model for the scenario from the perspective of a single service provider working for the same organization developing the two client applications. As shown in Fig. 18, any object that is either part of the Salesman or the Financial Expert client models is also in the Mortgage Sales SPIM model. As can be seen from the diagram, it includes several Data Ephemeral Object types, one Behaviour-Rich Ephemeral Object type as well as two Observably Stateful Architectural Object types.

Note that since clients usually only need to see a portion of a server's full capabilities, client PIMs usually include only a section of (i.e. a view of) a part of the server's overall PIM. Moreover, as in this case, different clients often have different views which only partially overlap.

8.2 PSM-level Models

After the introduction of the PIM-level models for the Mortgage Sales example scenario, this subsection presents a subset of the PSM-level models for the Mortgage Sales scenario using Java as a platform for the client-oriented PSM model and Web Services as the platform for the service-oriented PSM model. First, we consider the Salesman client Java CPSM as presented in Fig. 19. Note, that we omit the CPSM of the Financial Expert client for space reasons.

According to the Java CPSM metamodel that we have presented before, Fig. 19 contains the same entities as specified in the Salesman client CPIM, but replaced by the corresponding entities of the Java CPSM. Note that the properties "shared" and "private" are removed in the CPSM since the handling of these issues is deferred to the underlying infrastructure (which is aware of the PIM-level specifications) the applications interact with when executing. The most important issue here is that

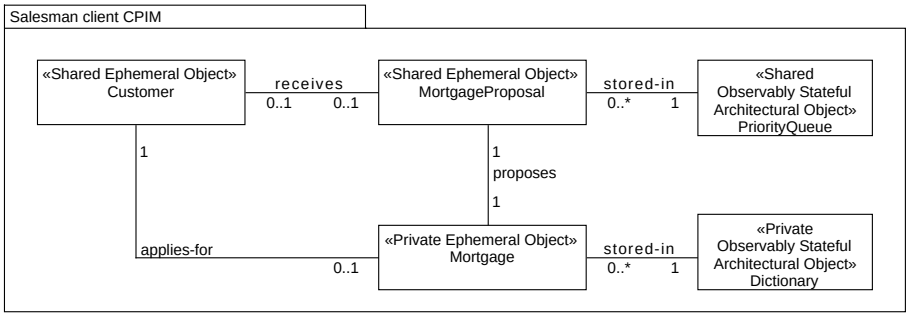

Figure 16 – Salesman client CPIM.

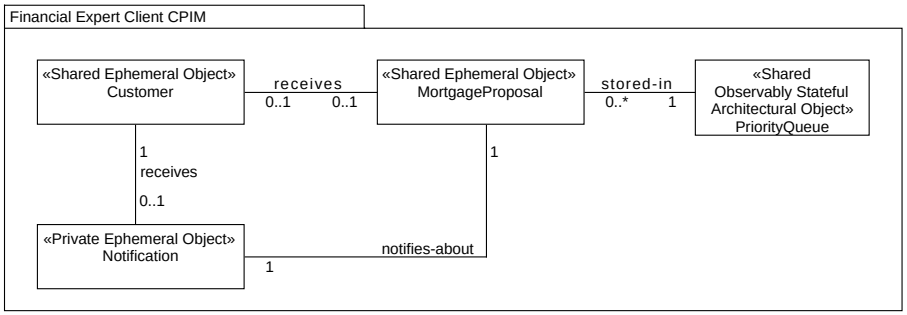

Figure 17 – Financial Expert client CPIM.

Ephemeral Object types are represented by regular Java classes and Architectural Object types are represented by static Java classes in the CPSM to convey their behaviour to the client developer.

Finally, to complete the Mortgage Sales example scenario and to cover the full spectrum of models proposed in this article, in Fig. 20 we present an SPSM based on the Web Service SPSM metamodel that we have introduced before. We therefore will also apply some of the patterns that have been proposed to map the presented PIM-level abstractions of the SPIM to PSM-level abstractions in the SPSM.

As Ephemeral Objects are not supported in the Web Services SPSM metamodel, we have to apply one of the proposed patterns of Sect. 5 to be able to create a model on the level of the Web Services SPSM. For reasons of space we pick out two examples at this point. The first is the Customer entity that has been mapped from an Ephemeral Object type to an Architectural Object type applying the Ephemeral Object Manager pattern. We have chosen to provide the PriorityQueue as an Inherently Stateless Architectural Object in this example. As a second example we pick out the PriorityQueue type that remains an Architectural Object as already specified in the SPIM. For the PriorityQueue we have chosen to provide it as an Inherently Stateful Architectural Object type whose instances maintain their state on their own, while the previous example, the Customer, defers state to an external participating resource like a database for example.

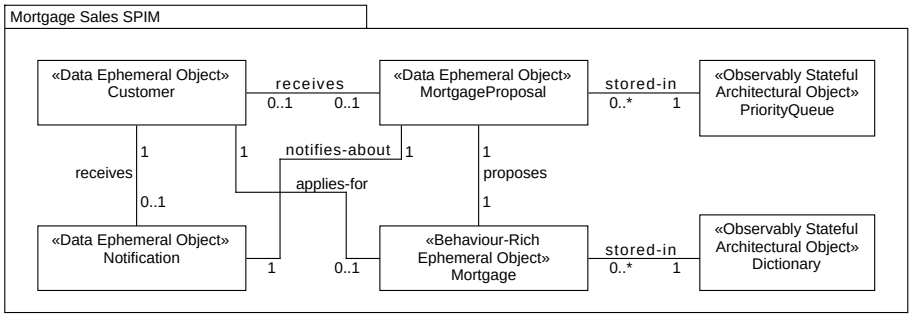

Figure 18 – Mortgage sales SPIM.

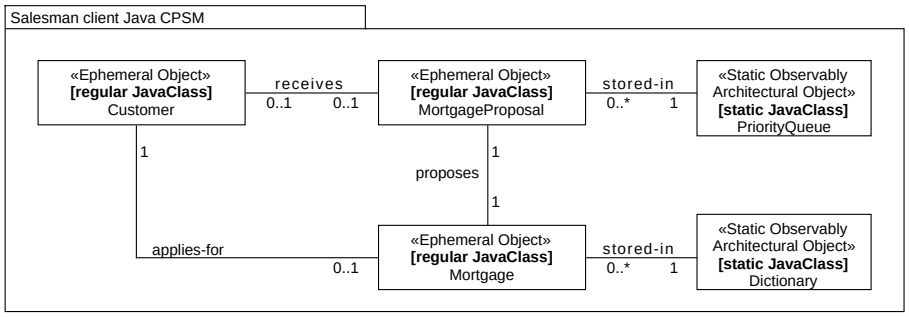

Figure 19 – Salesman client Java CPSM.

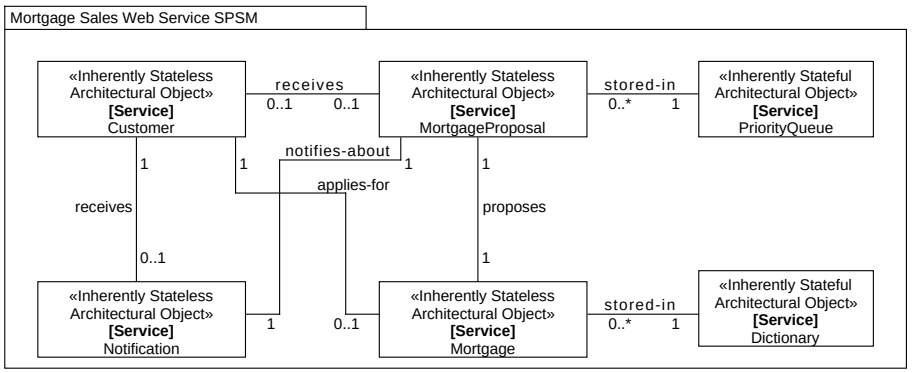

Figure 20 – Mortage Sales Web Service SPSM

9 Related Work

A dedicated language for modeling service-oriented architectures, the Service oriented architecture Modeling Language (SoaML) [Gro09] has also been standardized by the OMG. The Service-Oriented Modeling and Architecture (SOMA) published by IBM in 2004 [Ars04] was one of the first fully fledged methods targeted at service-oriented architectures. However, it is a very broad spectrum method that covers many more aspects of the development lifecycle than just the modeling and documentation of the services in a service-oriented architecture. The same goes for the Service-Oriented Modeling Framework (SOMF) developed by Michael Bell [Bel08]. It covers everything from the modeling of business goals and processes to enterprise architecture. Probably the method most focused on supporting the UML-based modeling of service-oriented architectures per se is the method of Piccinelli and Skene [PS05]. Their approach focuses on the modeling of Electronic Service Systems (ESSs). The approach uses a mixture of metamodels and profiles (the ESS profile) to support two views of services – a capability view showing how services are composed from capabilities and an implementation view showing how business assets are assigned to capability roles to make services concrete. However, both views are highly abstract and platform independent, and provide no support for modeling SOA realizations on a particular execution platform. Similarly, the method of Lopez-Sanz et.al. [LSACM08] aims to support the modeling of service-oriented software architectures using MDA principles. It does this within the context of the MIDAS model-driven methodological framework by defining a single metamodel containing typical concepts from service-oriented computing. However, like [PS05] it also focuses exclusively at the PIM level from a single perspective. In [MSTW12] a functional, symmetric view on pools of services is formally established. The interplay of client/server entities is specified by a variant of abstract state machines. It pursues an orthogonal, classical client/server architecture based on that formalization.

Rich internet applications [FRSF10] realize desktop experience on the basis of extended web client technology. The emergence of RIAs is technology-driven [FRSF10] and therefore the discussion of RIA development is necessarily technology-dependent. Disciplined approaches to RIA development are challenged by the concrete mix of technologies in this area. The approach in [ACL13] aims at a systematic RIA development by exploiting patterns from the model-driven architecture approach. It is a classical domain-specific language approach in that it creates a stable model of the exploited technologies. The approach [ZLC+12] tackles the concrete RIA data access problem and aims at mitigating the client/data friction. The realized framework provides support for the typical data issues in RIA development, e.g., proprietary local storage solutions. However, they must not be mixed with data-driven generative approaches [DLW05].

Although our approach shares some of the same goals as general purpose modeling and distributed computing frameworks such as RM-ODP [LMTV11], UFO [Gui05] and CORBA Component Model [Gro06] it is much more focused on service-oriented computing. In this regard it has more in common with COSMO [Qea07] and Service Component Architecture (SCA) [Edw14]. The big difference between these approaches and the approach presented in this article is that they do not explicitly distinguish between client-developer and service-provider views of service-oriented systems. The approach presented in this article is unique in (i) highlighting the four possible combinations of the PIM versus PSM perspectives and the service-client versus service-provider perspective as the four most important viewpoints from which to

visualize service oriented architectures, (ii) identifying a core set of concepts common to all these viewpoints, (iii) specializing these core concepts into four distinct metamodels optimized for the modeling of service-oriented architectures from the point of view of each distinct viewpoint.

Many of the concepts elaborated in this article may seem similar to the CORBA [Gro06] and J2EE [KtET00] frameworks, because these also tried to facilitate distributed computing in terms of object concepts. However, there are two main differences to our approach. The first is that we focus on client-side issues, and on reducing the artificial complexity experienced by client developers when accessing service infrastructures. Second, CORBA and J2EE both focus on the creation of distributed systems from a green field, from requirements to components, whereas we place equal, if not more, emphasis on the reuse of existing server-side assets, from components to requirements. One aspect of client-development that is not addressed in the approach is graphical user-interface design. Although it simplifies the development of graphical user interface by providing a simpler *model* in the sense of the MVC pattern, the approach presented in this article does not focus on the development of graphical user interfaces *per se*.

10 Conclusion

Given the importance of client side functionality in service-oriented computing, and the ongoing evolution of server-side realization technologies from distributed-object and service-oriented architectures to cloud computing, there is a growing need to lower the complexities and barriers involved in developing client applications. These range from large scale business applications and business processes to laptop programs and small "apps" on mobile devices. In this article we have presented a unified conceptual framework for describing service-oriented computing systems at two key levels of abstraction, i.e., a platform- independent and platform-specific level, and from the perspective of two key roles or stakeholders, i.e., the client-developer and the service-provider. This separation of concerns enables us to support distinct views of a system which are customized for the different stakeholders in a particular application optimized for their specific needs. In particular, client developers can be provided with an object-oriented viewpoint of the abstractions involved in a particular business process or application. This can relieve them of the burden of writing code or process specifications to interact with services at the level of abstraction that was optimized for maximum interoperability rather than for usability by clients. The main contribution of this approach lies not only in the shape and structure of the overall modeling framework but also in the separation of concerns and identification of common abstractions that is reflected in the contents of the various metamodels. On the basis of the achieved conceptual model we were able to define a set of six basic realization patterns that can be used to support the core client-oriented abstractions on top of all compliant server-side technologies. Finally, we have explained why the achieved unified conceptual model and the defined transformational pattern perfectly integrate with the Orthographic Software Modeling approach.

References

[AB09] C. Atkinson and P. Bostan. Towards a Client-Oriented Model of Types and States in Service-Oriented Development. In *EDOC 2009 – the 13th IEEE International EDOC Conference*, August 2009. doi:10.1109/EDOC.2009.16.

[ABHS07] C. Atkinson, P. Bostan, O. Hummel, and D. Stoll. ICWS 2007 – the 5th IEEE International Conference on Web Services. In *A Practical Approach to Web Service Discovery and Retrieval*, . IEEE Press, 2007. doi:10.1109/ICWS.2007.12.

[ACL13] J.L.H. Agustin, P. Carmona, and Lucio. An MDA Approach to Develop Web Components. *Advances in Intelligent Systems and Computing*, 206:511–522, 2013. doi:10.1007/978-3-642-36981-0_47.

[AD13] C. Atkinson and D. Draheim. Cloud Aided-Software Engineering – Evolving Viable Software Systems through a Web of Views. In Z. Mahmood and S. Saeed, editor, *Software Engineering Frameworks for the Cloud Computing Paradigm*. Springer, 2013. doi:10.1007/978-1-4471-5031-2_12.

[AHVE07] Hess A, B. Humm, M. Voß, and G. Engels. Structuring Software Cities A Multidimensional Approach. In *EDOC 2007 – the 11th Enterprise Distributed Object Computing Conference*, pages 122–129. IEEE, 2007. doi:10.1109/EDOC.2007.17.

[Ars04] A. Arsanjani. *Service-Oriented Modeling and Architecture*. IBM Online Article, November 2004.

[AS08] C. Atkinson and D. Stoll. Orthographic Modelling Environment. In *FASE'08 – the 11th International Conference on Fundamental Approaches to Software Engineering*, LNCS 4961. Springer, March 2008. doi:10.1007/978-3-540-78743-3_7.

[ASB09] C. Atkinson, D. Stoll, and P. Bostan. Supporting View-Based Development through Orthographic Software Modeling. In *ENASE'2009 – the 4th Intl. Conference on Evaluation on Novel Approaches to Software Engineering*. INSTICC Press, 2009.

[Atk02] Colin Atkinson. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.

[Atk14] Colin Atkinson. From Language Engineering to Viewpoint Engineering – Keynote. In B. Combemale and D.J. Pearce and Olivier Barais and J.J. Vinju, editor, *SLE'2014 – the 7th International Conference on Software Language Engineering*, LNCS 8706. Springer, September 2014.

[BAD11] P. Bostan, C. Atkinson, and D. Draheim. Towards a Unified Conceptual Framework for Service-Oriented Computing. In *2nd International Workshop on Models and Model-driven Methods for Service Engineering (3M4SE-2011)*, Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2011. doi:10.1109/EDOCW.2011.29.

[Bel08] M. Bell. *Introduction to Service-Oriented Modeling – Service-Oriented Modeling: Service Analysis, Design, and Architecture*. Wiley & Sons, 2008.

[Ber96] Philip A Bernstein. Middleware – a Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, February 1996. doi:10.1145/230798.230809.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, 2000.

[Coa11] Smart Manufacturing Leadership Coalition. *Implementing 21st Century Smart Manufacturing – Workshop Summary Report*. SMLC, June 2011.

[DLW05] D. Draheim, C. Lutteroth, and G. Weber. Robust Content Creation with Form-Oriented User Interfaces. In *CHINZ 2005 – 6^{th} International Conference of the ACM's Special Interest Group on Computer-Human Interaction*, ACM International Conference Proceeding Series, vol. 94. ACM Press, 2005. doi:10.1145/1073943.1073953.

[Dra10] Dirk Draheim. The Service-Oriented Metaphor Deciphered. *Journal of Computing Science and Engineering*, 4(4), December 2010.

[DW04] D. Draheim and G. Weber. *Form-Oriented Analysis – A New Methodology to Model Form-Based Applications*. Springer, October 2004.

[Edw14] M. Edwards. *Service Component Architecture*. OASIS, April 2014.

[Emm93] M.A. Emmelhainz. *EDI – A Total Management Guide*. Van Nostrand Reinhold, 1993.

[FRSF10] P. Fraternali, G. Rossi, and F. Sanchez-Figueroa. Rich Internet Applications. *IEEE Internet Computing*, 14(3):9–12, 2010. doi:10.1109/MIC.2010.76.

[FT02] R.T. Fielding and R.N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, 2002. doi:10.1145/514183.514185.

[Gea95] Erich Gamma and et al. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing – Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1st edition, 1992.

[Gro06] Object Management Group. *CORBA Component Model Specification, version 4.0, formal/06-04-01 dtc/06-02-01*. Object Management Group, April 2006.

[Gro09] Object Management Group. *Service Oriented Architecture Modeling Language – SoaML OMG Specification*. Object Management Group, 2009.

[Gui05] G. Guizzardi. *Ontological Foundations for Structural Conceptual Models, PhD Thesis, CTIT Series, No. 05-74*. CTIT, 2005.

[Haa05] Laura Haas. Building an Information Infrastructure for Enterprise Applications. In *1^{st} VLDB Workshop on Trends in Enterprise Application Architecture*, LNCS 3888. Springer, 2005. doi:10.1007/11681885_1.

[KtET00] N. Kassem and the Enterprise Team. *Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition*. Sun Microsystems, 2000.

[Lar04] Craig Larman. *Applying UML and Patterns, 3rd edition*. Prentice Hall, 2004.

[LMTV11] P. F. Linington, Z. Milosevic, A. Tanaka, and A. Vallecillo. *Building Enterprise Systems with ODP – An Introduction to Open Distributed Processing, 1st edition*. Chapman & Hall, 2011.

[LSACM08] M. Lopez-Sanz, C. J. Acura, C. E. Cuesta, and E Marcos. Defining Service-Oriented Software Architecture Models for a MDA-based Development Process. In *7th Working IEEE/IFIP Conference on Software Architecture*, 2008. doi:10.1109/WICSA.2008.15.

[MSTW12] H. Ma, K.-D. Schewe, B. Thalheim, and Q. Wang. A Formal Model for the Interoperability of Service Clouds. *Service Oriented Computing and Applications*, 6(3):189–205, 2012. doi:10.1007/s11761-012-0101-7.

[Ous98] J.K. Ousterhout. Scripting – Higher-Level Programming for the 21st Century. *Computer*, 31(3):23–30, 1998. doi:10.1109/2.660187.

[Par85] David L. Parnas. Software Aspects of Strategic Defense Systems. *Communications of the ACM*, 28(12):1326–1335, December 1985. doi:10.1145/214956.214961.

[PS05] G. Picinelli and J. Skene. Service-oriented Computing and Model Driven Architecture. In *Service-Oriented Software Systems Engineering – Challenges and Practices*. Idea Group, 2005.

[Qea07] D. Quartel and et al. COSMO: A Conceptual Framework for Service Modelling and Refinement. *Information Systems Frontiers*, 9(2–3):225–244, July 2007. doi:10.1007/s10796-007-9034-7.

[Sol03] R. Soley. *Model Driven Architecture, white paper formal/02-04-03, draft, 3.2*. Object Management Group, November 2003.

[Wei91] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265, September 1991. doi:10.1145/329124.329126.

[ZLC+12] Q. Zhao, X. Liu, X. Chen, J. Huang, G. Huang, and H. Mei. A Data Access Framework for Service-Oriented Rich Clients. *Service Oriented Computing and Applications*, 6(2):99–116, 2012. doi:10.1109/SOCA.2010.5707150.

About the authors

Colin Atkinson has been the leader of the Software Engineering Group at the University of Mannheim since April 2003. His research interests are focused on the use of model-driven and component based approaches in the development of dependable computing systems. Contact him at atkinson@informatik.uni-mannheim.de, or visit http://swt.informatik.uni-mannheim.de/.

Philipp Bostan is research associate at the Software Engineering Group at the University of Mannheim. His research interests are focused context-sensitive on services and service discovery, as well as service-oriented software development. Contact him at bostan@informatik.uni-mannheim.de.

Dirk Draheim is head of the data center of the University of Innsbruck. He is also adjunct reader at Software Engineering Group at the University of Mannheim. His research interests are focused on software science and the specification of software-intensive systems. Contact him at draheim@acm.org, or visit http://draheim.formcharts.org.