

Service Engineering Design Patterns

Nikola Milanovic

Hasso-Plattner-Institute, Potsdam University

nikola.milanovic@hpi.uni-potsdam.de

Abstract

Design of service-oriented applications differs from design of the component-based applications in granularity, level of abstraction and dynamic environment in which binding between clients and servers takes place. Service engineering design patterns are introduced as "best practice solutions" to common recurring problems that designers face when developing service-based applications. Design aspects that are covered are service access, communication, security, dynamic configuration and interaction (composition). The goal is to introduce a disciplined method for development of service-oriented applications.

Keywords: Web services, design patterns, composition, availability

1 Introduction

Design pattern is a description of the core of an engineering problem that occurs repeatedly in practice, and description of the solution to that problem, such that it is reusable in different contexts. Design patterns were first described in [1] and applied to civil engineering. They were introduced to object-oriented software engineering in [4]. The benefits of design patterns are that they provide high-level language for describing design issues and that combinations of design patterns lead to development of reusable architectures.

Many design patterns have been identified for object-oriented software systems, and some of them have become standard design elements (e.g., Observer, Façade, Command) or have been incorporated in programming languages (e.g., Factory Method). With the introduction of service-oriented computing, it has been noted [11] that usability of "new" patterns is approaching zero. We are thinking at the new level of granularity when developing service-oriented applications: instead of the object/class, we work at the subsystem/application level. Surprisingly little research is being done in the area of developing methodologies of "good" service-oriented engineering, with the notable exception of [12] and [2, 3], where several practi-

cal Web service design patterns are explained. The design patterns proposed in this paper should be considered complementary to them, as "best-effort" solution for enforcing quality design [13]. A comparison with the SOA blueprints will be given at the end.

There are two main differences between object-oriented and service-oriented design patterns: 1) different levels of design granularity and abstractions; 2) service-oriented design is not inherently client-server, but dynamic: clients can choose among many servers (services). The goal of the paper is to identify design patterns that are specific to service-oriented design, and to express them using composition operators, thus creating reusable and verifiable building elements for service-oriented applications.

2 Composable Service Architecture

Our previous work, which is used to formalize design patterns, resulted in development of the composable service architecture [6, 8, 9]. In general, composable service architecture solves the problem of service composition, alleviating problems identified in [7], namely lack of support for description of non-functional and semantic properties, difficult verification and automatic service composition. The main properties of the architecture are:

- extended non-functional (QoS) service description using contract-based mechanism (pre-conditions, post-conditions and invariants)
- enhanced search capabilities compared to UDDI
- formal composition operators and verification of composition correctness
- automatic service composition

Each service has dual description: XML contract (transport over a network) and abstract machine (formal notation used for reasoning about composition correctness). Service composition is performed by merging abstract machines using five basic composition operators:

- *sequence* (\triangleright) executes two or more services in a sequential order
- *selection* (\odot_C) chooses one of the operand services based on the predicate C
- *choice* (\square) executes two or more services in parallel and non-deterministically chooses output of one and only one of them
- *parallel with communication* (\parallel_P) executes two or more services concurrently and then chooses one of the outputs based on the predicate P
- *parallel without communication* (\parallel) executes two or more services concurrently without synchronization
- *loop* (\odot_P) executes a service iteratively until the exit condition P is met

New services are constructed by applying composition operators to the existing services. Each composition operator is associated with a set of rules that describe how new abstract machine is constructed after its application. After a composed service has been constructed, its correctness is checked:

- *type checking* ensures that all types are correctly defined and that there is no infinite set inclusion in the resulting abstract machine
- *invariant preservation* ensures that composed invariant is preserved by all operations
- *correct termination* ensures that all composed operations will terminate correctly and feasibly

In the following sections, several Web service design patterns will be identified. Instead of describing them using UML diagrams (similar to OO patterns), we will use composition operators to express them. The key difference will thus be the ability to formally introduce and verify application of design patterns in service-oriented software systems.

3 Proxy Pattern

The easiest way to access a Web Service is directly, using specific API on the client side that connects to the WSDL interface of a target service. However, this method creates strong coupling between client and called service, and makes reuse difficult, since the same calling code has to be repeated. A solution is to use a service proxy pattern that decouples target service from the client by using surrogate (proxy) service instead of a target service. A proxy pattern should not be confused with the proxy class created by compiling WSDL, a feature offered by most Web Service

development frameworks. For a proxy pattern, it is irrelevant how actual service invocation is performed (using a proxy class or generating SOAP messages directly).

3.1 Single Proxy

Single proxy pattern is used to access single Web Service indirectly. It inserts additional proxy Web Service between client and server. The task of a proxy service is to read input parameters, invoke target Web Service and receive results. Therefore, communication with the target service is implemented only once, inside proxy service. This facilitates reuse and it is also possible to change the interface of the back-end service without notifying the clients, as it is enough to update only proxy accordingly. Proxy pattern is described (Figure 1a):

$$client \triangleright proxy \triangleright target$$

The main benefit of using a proxy pattern is that it enforces loose coupling between client and called service. Also, this extra layer of indirection can be used for logging or load balancing, as will be shown later.

3.2 Multiple Proxy (Transformer)

A service can have more than one proxy. In case of multiple proxies, they are used to convert (transform) interface of the target service according to the expectations of different clients. The alternative name for this pattern is transformer. Transformer is used to enforce understanding on semantic meaning of parameters, or to help connecting services developed with different back-end technologies.

Adopting XML and SOAP does not guarantee interoperability. There are many styles and implementations of SOAP and practice has shown that they are not compatible with each other. The issue of data types is extremely tedious to manage if partner services are exchanging other types than primitives like integers and strings. The problem is exacerbated by the fact that different service providers will provide different sophistication when publishing their services' descriptions. The issues of naming data, assigning business meaning to XML data and serialization/deserialization are some of the real-world problems. Since it is not realistic to expect that single XML schema can be enforced to all business partners, the idea is to provide a separate proxy for every partner, resulting in transformer pattern. The composition describing this pattern is (Figure 1b):

$$(client_1 || \dots || client_m) \triangleright$$

$$(proxy_1 \odot_{C_1} proxy_2 \odot_{C_2} \dots \odot_{C_n} proxy_n) \triangleright target$$

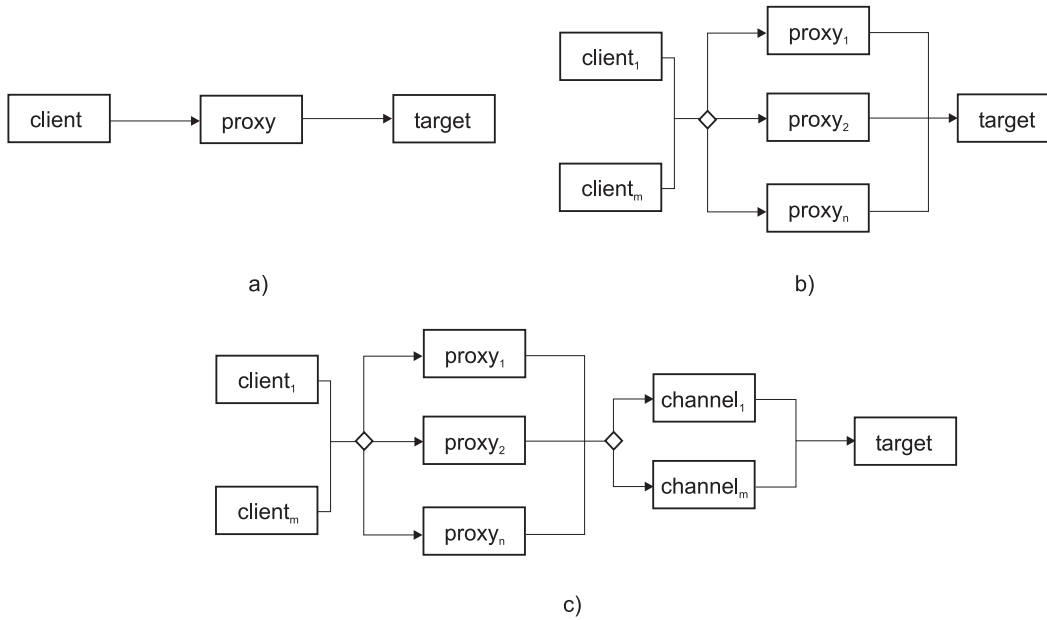


Figure 1. Proxy Pattern

This pattern should not be used to connect to multiple services using multiple proxies, as this is the task of the façade pattern. Instead, multiple proxy always connects to a single target service.

3.3 Proxy with Channel

Further expansion of the proxy pattern is decoupling of communication protocol and business interface. A channel service is introduced that deals with communication protocol issues while proxy service deals with parameters and business logic. That way communication protocol can be changed without changing either client or service proxy. Multiple proxies can share a single channel, or choose among several available ones. The composition expressing this pattern is (Figure 1c):

$$(client_1 || client_m) \triangleright ((proxy_1 \odot_{C_1} \dots \odot_{C_n} proxy_n) \triangleright (channel_1 \odot_{C_1} \dots \odot_{C_p} channel_p)) \triangleright target$$

Client sends a request to one of the proxies. Proxy performs data transformation and prepares the request for the target service. Then it selects appropriate channel and sends raw data. The channel packs received data into appropriate message and invokes the target service. The channel also receives results, and returns them to proxy for relevant formatting. This means that proxy and target service are strongly coupled, but channel can be changed without notifying the client.

4 Façade Pattern

While proxy pattern facilitates access to a single Web Service, façade pattern performs the same for compositions of services. The problem that façade pattern solves is how to access a composition of Web Services.

Contrary to the OO-Façade, multiple network calls are not a problem when invoking Web Service composition, since composite request is sent to the server (e.g., BPEL server) which manages network calls. However, the problem is the coupling between the client and called services. Although mediated by the composition server, the client has to have intimate knowledge of all services involved. This makes replacement of services in composition difficult. Reusability is not that problematic, as composition can be stored in a directory and then re-invoked when necessary. Finally, in this case it is up to the client and/or composition server to specify non-functional behavior of the constituent services (e.g., transactions) which is a weak design point that can introduce potential inconsistencies.

The idea is to represent fine grained operations offered by partner Web Services in a composition through a single Façade service, which offers a coarse grained composite operation to the client. The pattern is expressed (Figure 2a):

$$(client_1 || \dots || client_n) \triangleright facade \triangleright (target_1 \circ target_2 \dots \circ target_m)$$

where $\circ \in \{\triangleright, \odot, \square, ||, \cup\}$.

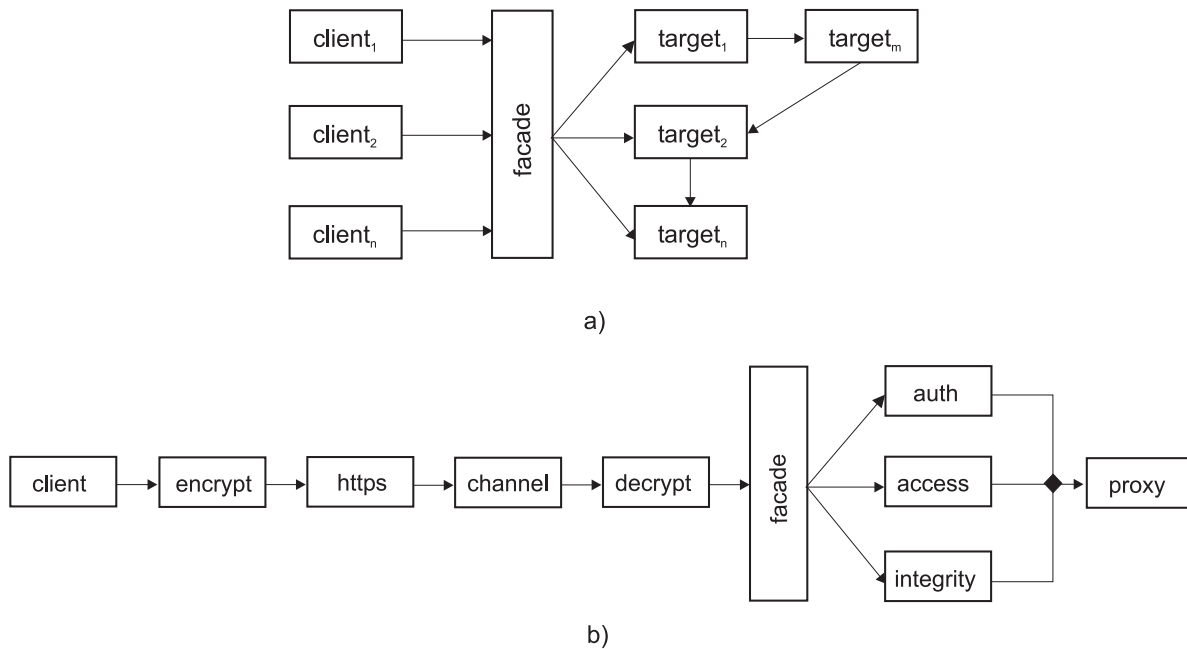


Figure 2. Façade and Security Pattern

Façade and target services are strongly coupled, but that can be eliminated by applying proxy pattern for each target service. This is the question of the composition complexity, as combining proxy and façade patterns can prove to be an overkill for the application, depending on its size and requirements.

The benefit of applying façade pattern is in decoupling client from the partner services comprising the composition. Also it is more natural that the provider of one or more partner services specifies non-functional behavior (e.g., transactions, security, timeliness) instead of the client or the composition server.

4.1 Synchronous Façade

If a synchronous façade pattern is used, client blocks and waits until entire composition is executed. This type of invocation is used when transactional attitude is strict and client must know the result of entire composition execution before proceeding further.

A good example where synchronous façade is used is a bank transfer scenario. A composition is created that accepts two bank account numbers, security credential and an amount to be transferred from one account to the other. This is solved by composition of three services: authorization, withdrawal and deposit.

Client cannot continue its execution before all three partner services complete. If any of the partner services abort, entire operation has to be aborted and compensated, which

is the task of the façade service. Transactional behavior can be solved at the level of composition server, but ensuring consistency by the service provider itself (assuming that all three partner services are published by the same provider) at the façade level is more natural and recommended. Finally, if the façade is running inside the same application server as partner services, network overhead will be eliminated.

4.2 Asynchronous Façade

Asynchronous façade is used when the nature of a composition is such that a client does not require immediate response, furthermore, where such behavior would be harmful to the overall business logic and performance. This case is dictated by scalability and timeliness requirements. Such compositions allow clients to continue their own processing while their requests are queued. Asynchronous façade queues client requests and responds after all composite services have finished (using callback or polling by the client).

The example of a composition where asynchronous façade is a good solution is airline reservation system. A client invokes a composition of services that make hotel and flight reservation and a payment service. If a synchronous façade pattern is used to encapsulate this composition, two problems are encountered: long delay and reliability. Checking flight and hotel availability can take a long time, and even involve human processing. Therefore, for business cases where transactions can take long time to execute, it is unacceptable for a client to block and wait. Using asynchro-

nous façade, a client is free to continue its own processing the moment it submits the request to the façade service and it will be notified once the composition terminates, successfully or unsuccessfully. Fault-tolerance is also a problem with long running business transactions. If a synchronous façade has been used and one partner service aborts, entire composition will abort. However, when asynchronous façade is used, the request is kept in a façade queue and the execution is retried, thus making long running transactions resilient and reliable.

5 Security Patterns

There are two ways security requirements can be addressed. The first is to exploit native security capabilities of partner services that build the application. Although easier, it can result in incorrect composition in case when partner services do not support relevant security properties themselves. The second option is to 'reinforce' a composition with dedicated services, creating a security pattern or wrapper for the entire composition.

Basic security attributes that can be reinforced in a composition are: transport, encryption, access control and message integrity. Security pattern for simple proxy invocation is (Figure 2b):

$$client \triangleright encrypt \triangleright https \triangleright channel \triangleright decrypt \triangleright facade \triangleright (authorize \square access \square integrity) \triangleright proxy$$

Transport protection secures the channel transmitting data from client to facade/ proxy. It encrypts the connection between two services, thus protecting the channel. Secure Sockets Layer (SSL) and HyperText Transfer Protocol Secure (HTTPS) can be used for this purpose. While in transit, all data is secure. However, data itself is not encrypted, meaning that at the endpoint information is easily read. Therefore, all data is also encrypted before being sent to the channel and decrypted at the receiving end. Finally, after being received by the facade and before being forwarded to proxy, authorization, access control and message integrity verification is performed. It is up to the architect to decide which of these security services should be used in a particular use case, since they introduce significant overhead.

6 Dynamic Input Pattern

Web Service messages can be very complex, comprising many input/output parameters. It is in accordance with basic postulates of service-oriented computing, which recommend using small number of operations with relatively large and complex messages. Those messages are often not

known until runtime, that is, they are constructed dynamically. Data necessary for message construction can come from an XML file, a relational database or from a human user input.

Many Web Services also operate with persistent resources which have to be identified (e.g., relational database, table name and primary key). This data is also not known during design time. In such cases it is convenient to remove logic for dynamic message construction outside of the client, into a combination of service proxy and configuration manager. This pattern is expressed by (Figure 3a):

$$client \triangleright (dynamicProxy || configurator) \triangleright integrator \triangleright target$$

Dynamic proxy decouples client from server and configurator consults persistent resource to retrieve necessary data. Both services forward their output to integrator which generates the complete request by filling missing information that it receives from the configurator. In order to remove latency required for consulting persistent resource, configurator caches configuration data in memory, accelerating dynamic message construction and lowering overall response time.

7 Log Pattern

Reputation system is the necessary part of a service-oriented architecture. In order for reputation system to be fair and usable, it is convenient to introduce standard design pattern that requires services to log their input and output messages. That way logging logic is removed from the partner services. Apart from being useful for building standardized reputation systems, logging pattern can be used for debugging and testing, which is often neglected. Developers creating service-oriented applications are often in darkness when trying to debug and diagnose application errors. Even access to error logs is a problem since target services execute in different application servers and access to their logs is not always trivial and/or possible. Introducing standard logging pattern helps in the validation and diagnosis of service-oriented applications.

Practice shows that it is recommendable to perform logging in a database, since long XML messages result in unusable log files. It is desirable to have related messages grouped together (concurrency issues). If all related messages are stored in a database, a simple cross join retrieves all relevant data. The pattern is expressed (Figure 3b):

$$client \triangleright (proxy || requestLog) \triangleright (target || responseLog)$$

Alternatively, proxy can also log response, eliminating the need for separate response logger:

$$client \triangleright (proxy || requestLog || responseLog) \triangleright target$$

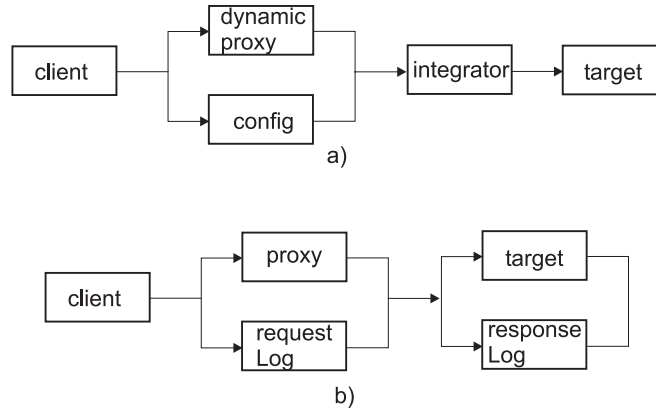


Figure 3. Dynamic Input (a) and Logger (b) Patterns

8 Load Balancer Pattern

Frequently a pool of Web Services that perform the same function is available. Instead of invoking them on a random basis, they can be load balanced using proxy pattern. Inserting a proxy between clients and a pool of target services, and equipping that proxy with a load balancing algorithm optimizes the performance of the whole system. The role of a load balancer is to distribute client requests among available service instances. The pattern is expressed (Figure 4a):

$$\begin{aligned}
 & TI_1, TI_2, \dots, TI_m \text{ EXTENDS } T \\
 & (client_1 || \dots || client_n) \triangleright proxy \triangleright loadBalancer \\
 & \triangleright (TI_1 \odot_{C_1} \dots \odot_{C_m} TI_m)
 \end{aligned}$$

If a synchronous load balancer is implemented, only a guessing algorithm can be used, since there is no way that load balancer can know for sure which services are available and which are not. This decision has to be taken based upon imperfect historical data. This is a push model, where requests are pushed to the target service instances without their cooperation. With asynchronous load balancer, a pull model can be used. Load balancer can store requests in a queue and target instances can retrieve and process (pull) them once they are free.

9 Publish-Subscribe Pattern

The main actors in this pattern are publishers, subscribers and communication middleware services (message bus). Publishers produce values (computations, measurements) and put them on the message bus. Subscribers opt to receive a selection of available data from the bus. The bus comprises aggregator, transformer, queue and directory. Aggregators perform data merging, transformers perform

transformation on raw data (e.g., a FFT), queue stores raw, aggregated or transformed data, while directory collects descriptions of available data. Subscribers consult directory when choosing data to subscribe to. The pattern is described (Figure 4b):

$$\begin{aligned}
 & \{[(target_1 \triangleright publish_1) || \dots || (target_n \triangleright publish_n)] \triangleright \\
 & (aggregate_1 \square \dots \square aggregate_p \square transform_1 \square \dots \square \\
 & transform_q) \triangleright (queue || directory)\} || \\
 & \{(client_1 \triangleright subscribe_1) || \dots || (client_m \triangleright subscribe_m)\} \triangleright \\
 & (queue || directory)
 \end{aligned}$$

Taking the case of critical infrastructure protection as an example (see case for SOA in power grid in [5]), publishers are sensors that give various measurements along the grid elements (power plants, turbines, generators, transformers and distribution lines), subscribers are fault predictors that subscribe to information they require to perform prediction, while message bus is the heterogeneous communication middleware that must fulfill non-functional requirements of the subscribers. Each subscriber can specify a set of constraints that must hold (e.g., a fault predictor may require timely and secure delivery of measured data).

The proposed pattern offers additional availability benefit: composability over multiple domains. Subscribers may require data that originate at different providers, separated by legal, safety and technological barriers. Flexibility of unified interfaces, correctness guarantees and composability enables aggregation of data that would otherwise be inaccessible.

10 Producer-Consumer Pattern

This pattern comprises producer, consumer and storage (queue) services. Producer receives input data, processes it

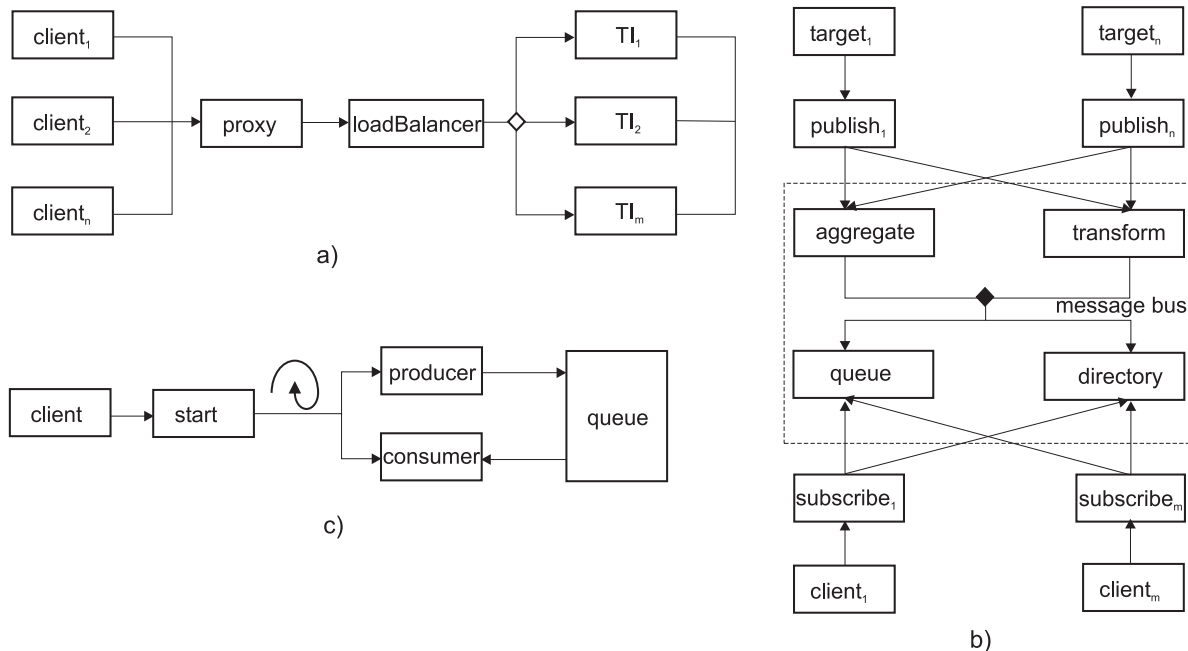


Figure 4. Load balancer, Publish-subscribe and Producer-consumer patterns

and puts it into the queue. Consumer takes values from the queue, performs its own processing and sends data to the output. They are both executing asynchronously. Producer and consumer model two elements of a business logic. For example, producer can receive requests for bank transfer, pre-process them (authorization, feasibility) and then put a request in a persistent storage (queue) for producer to take and perform the actual transfer and generate the report. This part of the workflow can involve human interaction. The entire process is executing inside a loop controlled by an external service *start*. The pattern is expressed (Figure 4c):

$$start \circ_{(exit > 0)} (producer \triangleright queue) || (queue \triangleright consumer)$$

Since producer and consumer are not synchronized, it is important that producer does not put elements into the full queue, or that consumer does not try to take an element from the empty queue. Apart from typing and feasibility, enforcing this consistency is one of the main tasks of this pattern which is accomplished by invariant preservation.

11 Design Patterns and SOA Blueprints

Service design patterns are related to SOA Blueprints initiative, an effort to develop, circulate, maintain, and update a set of example business profiles which illustrate practical issues in deploying SOA-based solutions [10, 14]. SOA Blueprints comprise requirements specification and concepts. Requirements specification enumerates coarse

grained services typically found or recommended for an enterprise environment (e.g., manufacturing, procurement, payroll, accounting). Concepts express infrastructural and logical properties of these coarse grained services. In this context, service design patterns are similar to blueprint concepts. We will now discuss all blueprint concepts and make comparison with equivalent design patterns:

- *Synchronicity*. Not addressed separately in design patterns, being part of the composable service architecture. However, where pattern expression depends on the synchronicity, it has been noted (e.g., synchronous and asynchronous façade).
- *Component services, composite services, serial and parallel orchestration, conversational services*. In composable service architecture, all services are treated as conversational, but can be reduced to composite/atomic. Parallel and sequential execution are part of the richer composition operators set (including also selection, choice and loop).
- *Data service*. No distinction about services used to fetch data is made either in composable service architecture or in service design patterns.
- *Publish-subscribe service*. Directly comparable to publish/subscribe design pattern.
- *Service broker*. Can be seen as combination of proxy and façade patterns.

- *Exception handling and compensation.* The issues of forward and backward error control are addressed at the architecture level (services that do not declare these properties in their contracts may not be composable, depending on the application requirements).
- *Interception and extensibility.* A mechanism for inserting additional functionality to the system without modifying and affecting existing components is the fundamental idea of the composable service architecture.
- *Interoperability.* It is unclear why this should be a blueprint concept or a design pattern. In our understanding this is the basic property of any SOA system.
- *Security.* Supported directly with security patterns.

Some blueprint concepts are the integral part of the composable service architecture (synchronicity, parallel or sequential execution). We see this as a possible benefit, since addressing issues such as error control at the architecture (mandatory) level yields more benefits. Some design patterns are not present in the blueprint concepts (e.g., load balancer). Finally, blueprint concepts do not support correctness verification. One of the advantages of expressing design patterns using composable service architecture is the ability to formally verify application of a given pattern with the chosen set of services. Blueprints requirements specification, on the other hand, is not supported in the design patterns. Therefore, a potential future work is the marriage of the blueprints concepts and design patterns, and subsequent integration into the blueprint requirements specification, and ultimately, in the example blueprints application.

12 Conclusion

Availability of service-oriented applications can be observed at two levels: physical and user-perceived. The usage of design patterns improves service availability at both levels. By careful choice of design patterns, application physical availability is increased. Increasing physical availability means making service resilient and fault tolerant in the presence of physical faults. An example of improving physical resilience that can be achieved is network failure (or congestion): using proxy pattern with alternative channels, network effects can be countered by dynamic channel/protocol switching. Even if no physical faults are present in the system, wrong design decisions can lead to the situations where service cannot deliver end-results to the users due to some specific conditions (e.g., high load). Disciplined design decisions and adherence to the standardized design patterns make service-oriented applications more stable, available and easier to maintain. Furthermore, design decisions can be communicated, shared and discussed in concise, effective and non-ambiguous manner.

The proposed patterns are the result of the author's experience while working on the Web Services Composition Server project [6]. As such, they were developed as answers to the following design problems: service access (proxy, façade), security, configuration (dynamic input), logging, performance (load balancer) and concrete application requirements (publish-subscribe and producer-consumer).

References

- [1] C. Alexander. *A Pattern Language*. Oxford University Press, 1977.
- [2] A. Barros and E. Boeger. A Compositional Framework for Service Interaction Patterns and Interaction Flows. In *Proceedings of the Seventh International Conference on Formal Engineering Methods (ICFEM'2005)*, pages 5–35, Manchester, UK, 2005.
- [3] A. Barros, M. Dumas, and A. ter Hofstede. Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection. In *Technical Report FIT-TR-2005-02*, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, 2005.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Ulisses. *Design Patterns*. Addison-Wesley, 1995.
- [5] C. Hauser, D. Bakken, and A. Bose. A Failure to Communicate. *IEEE Power and Energy*, pages 10–18, March/April 2005.
- [6] N. Milanovic. Contract-based Web Service Composition Framework with Correctness Guarantees. In *Proceedings of the 2nd International Service Availability Forum (ISAS)*, pages 46–59, Berlin, Germany, 2005.
- [7] N. Milanovic and M. Malek. Current Solutions for Web Service Composition. *IEEE Internet Computing*, 8(6):51–59, 2004.
- [8] N. Milanovic and M. Malek. Architectural Support for Automatic Service Composition. In *Proceedings of the IEEE Service Computing Conference (SCC 2005)*, pages 133–140, Orlando, USA, 2005.
- [9] N. Milanovic and M. Malek. Search Strategies for Automatic Web Service Composition. *International Journal of Web Services Research*, 3(2):1–32, 2006.
- [10] OASIS. Service Oriented Architectural Adoption Blueprints TC. In <http://www.oasis-open.org/committees/soa-blueprints/charter.php>, 2006.
- [11] G. Prasad, R. Taneja, and V. Todankar. Web and Enterprise Architecture Design Patterns for J2EE. *O'Reilly OnJava*, <http://www.onjava.com/lpt/a/4161>, 2003.
- [12] J. Snell. Web services programming tips and tricks: Learn simple, practical Web services design patterns. www.ibm.com/developerworks/library/ws-tip-altdesign1/, 2005.
- [13] A. van Moorsel. On Best-Effort and Dependability. In *Proceedings of the 2nd International Service Availability Forum (ISAS)*, pages 99–101, Berlin, Germany, 2005.
- [14] S. Wilkes and J. Harby. *SOA Blueprints Reference Example Requirements Specification*. SOA Center, 2004.