Service Abstraction Layer

Oliver Vogel
SYSTOR AG
e-mail: mail@ovogel.de

- Conference Draft -

Abstract

Building extensible and scalable systems is a challenging task. Nowadays, different clients communicating over different channels, like HTTP or WAP, must be offered with the same set of business services.

The number of channels having to be supported and the number of demanded services increase over time. This has to be considered during the development of such a system as the architecture must support new channels and services seamlessly. Thus, during the development of a multi channel enabled system architects are faced with the following question:

"How do you develop a system which can fulfill requests from different clients communicating over different channels without having to modify your business logic each time a new channel has to be supported or a new service is added?"

SERVICE ABSTRACTION LAYER gives an answer.

Classification

Architectural

Example

Consider a research platform for analysts of an investment bank. In order to conduct their research, analysts need information about financial instruments, exchange rates, company balances and other investment related issues. Based upon this information analysts run calculations and estimates. The research platform needs to support the analysts in that activity by offering business services which provide business codes, like calculations of earnings per share, market capitalization, EBIT and which can calculate the performance of a given instrument or the aggregate of certain business codes.

Analysts are used to work with their favorite spreadsheet application, but as the research platform shall be build according to a 3-tier architecture the business logic must not be included into the spreadsheet. Instead it has to reside on the business tier and as a consequence the spreadsheet application may contain presentation and simple validation logic only. To increase portability the investment bank has decided to implement its research services in the Java programming language. However, the corporate spreadsheet application is a COTS solution which offers a Visual Basic API only. To build a bridge between these different worlds, a COM-Java bridge [COMBridge2000] can be used allowing the communication between objects living in these different worlds. Therefore, the business tier needs to be able to handle service requests coming from a client using the previously mentioned COM-Java bridge.

As the business services do not only represent added value for analysts but also for customers of the investment bank, the bank has decided to offer these services on their web site. This introduces another channel as the communication protocol for web applications is HTTP. In order to provide its services on the web, the business tier has to accept HTTP requests. Besides the COM and HTTP channels it is also planned to support WAP devices in the near future. Figure 1 illustrates the architecture of the research pla tform.

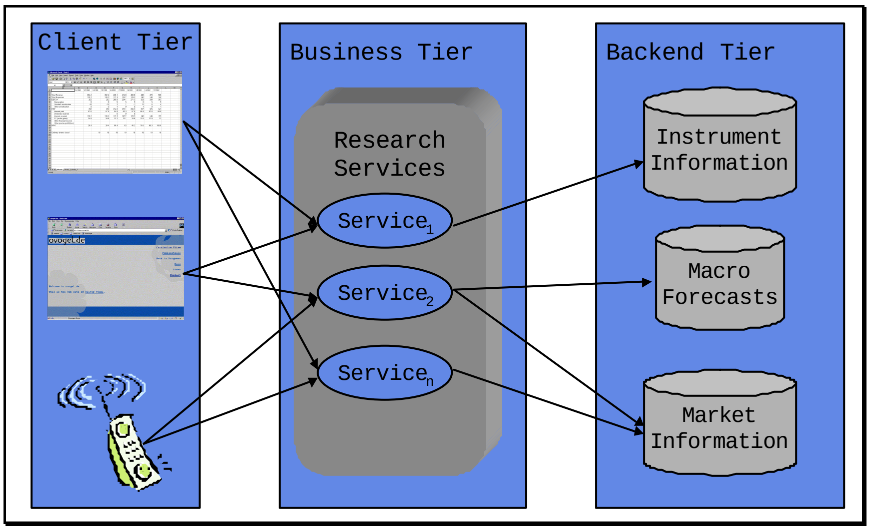

Figure 1: Architectural view on research platform

Each channel has its own communication protocol, like HTTP or WAP. As the research services have to be available for any client communicating over any channel, the corresponding business logic needs to handle these different protocols. How can this requirement be achieved? An ad hoc approach may result in a design where the logic needed to communicate with different types of clients is directly combined with the logic necessary to offer the required research service. This solution has several drawbacks. First of all every service has to implement the communication logic. Therefore, each time a new research service is added, the communication logic must be integrated into the new service. Moreover, the research services must be modified each time a new channel has to be supported. For example, it could be possible, that WSDL [WSDL2001] or SOAP [SOAP2001] will have to be supported, if the investment bank intends to enter the B2B market. The previously mentioned approach also has drawbacks on the client side as clients use the concrete interfaces of the research services. These interfaces might change and new services might be added over time. Thus, any modification on the service side will directly influence the client side.

The architectural concept illustrated in this example lacks in two major points. First of all it combines communication logic with business logic. Further, it allows clients to use concrete interfaces to access services. These architectural decisions lead to an architecture which is not stable and is prone to changes as on the one hand fundamentally different concerns are not clearly separated and on the other hand clients are faced with fragile interfaces.

SERVICE ABSTRACTION LAYER illustrates how to avoid these negative decisions and how to create an flexible and extensible architecture.

Context

Nowadays, many software systems use 3-tier architectures [Kassem+2000, Ayers+1999]. In such architectures a client, a business and a backend tier can be identified.

As the business tier has to satisfy requests from clients, like web browsers, Java Swing or Visual Basic applications, it must deal with service requests coming from different channels. Each channel is characterized by its own communication protocol. Therefore, a server, residing on the business tier, must support these communication protocols. As the services respectively the business logic representing the services are channel neutral, the channel specific knowledge should not be incorporated into the business logic. Moreover, clients must have the ability to call any kind of service located on the business tier. Therefore, they must be provided with a generic mechanism to request services.

Forces

This pattern resolves the following forces:

- **Separation of Concerns**: A clear separation of concerns should be achieved by isolating business logic from communication logic.
- **Support of different channels**: It should be possible that service requests can be issued from different channels. Thus, different communication protocols shall be supported.
- **Controlled evolution**: Changes to the system in order to support new channels should not require changes throughout the system. Thus, business services should not be influenced by new channels.
- **Generic request handling**: Adding new business services should not require to change the request handling logic.
- **Interface fragility**: Interfaces to business services should be stable and should not have to be modified each time a new service is added.
- **Communication transparency**: The communication between clients and business service providers shall be transparent to avoid that clients are strongly coupled to concrete service providers.

Problem

How do you develop a system, which can fulfill requests coming from different clients communicating over different channels without having to modify your business logic each time a new channel has to be supported or a new service is added?

Solution

SERVICE ABSTRACTION LAYER introduces an extra layer to the business tier containing all the necessary logic to receive and delegate requests. Incoming requests are forwarded to *ServiceProviders* which are able to satisfy requests. A *ServiceProvider* executes requested *Services* and returns *ServiceResults* to clients. By abstracting concrete subsystems to *ServiceProviders*, method calls to *ServiceRequests* and method results to *ServiceResults* new subsystems can be integrated seamlessly. To allow clients to access *Services* over different channels special *ChannelAdapters* are used. These adapters convert channel specific *ServiceRequests* into channel neutral *ServiceRequests* and channel neutral *ServiceResults* into channel specific *ServiceResults* respectively.

Core Solution Building Blocks

The core building blocks are introduced in a stepwise manner starting at the highest level of abstraction.

SERVICEABSTRACTIONLAYER

In order to follow *SEPARATE CONCERNS* [Voelter2001, p. 7 ff.] the first step in creating a multi channel architecture is to introduce a *ServiceAbstractionLayer*. By introducing a *ServiceAbstractionLayer* the logic needed to receive and to delegate service requests to appropriate *ServiceProviders* can be localized in a clearly defined place.

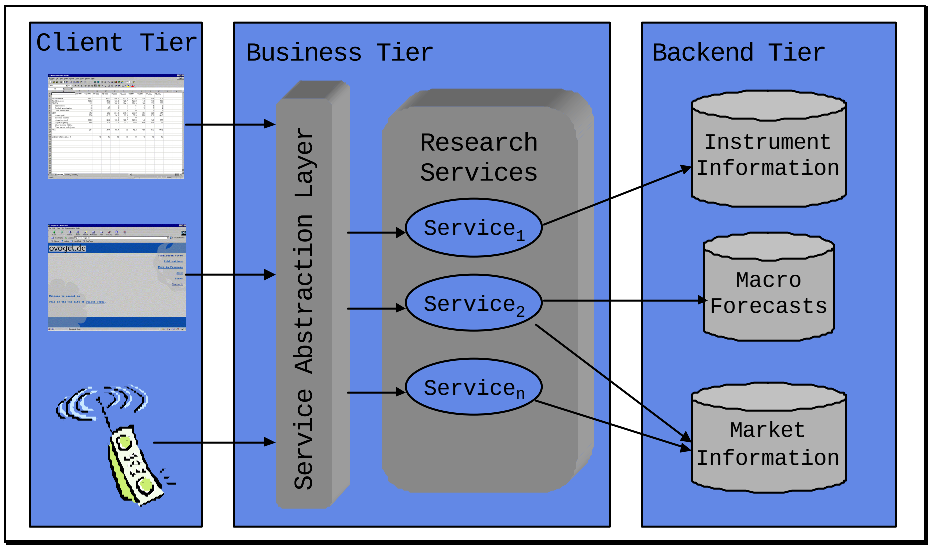

Figure 2: Introduction of a Service Abstraction Layer

As shown in Figure 2 clients communicate with the *ServiceAbstractionLayer* only. Thereby, clients are decoupled from concrete business services. This reduces the number of dependencies and thus eases the evolution of the overall system. In consequence of a *ServiceRequest*, the *ServiceAbstractionLayer* delegates the request to the *Service*, which can fulfill the request. Introducing a *ServiceAbstractionLayer* is an application of the *LAYERS* pattern [POSA1996, p. 31ff.]. The *ServiceAbstractionLayer* offers a generalized protocol for the communication between clients and services. It can be considered as a framework, offering well defined hot spots for the integration of different clients and diverse *Services*. The primary abstractions of the framework will be introduced in the following paragraphs.

SALFAÇADE

The *ServiceAbstractionLayer* should hide its internal representation from its clients to ease the exchange and modification of its internal structure and logic. Therefore, a *FAÇADE* [Gamma+1995, p. 185ff.] should be used which acts as the main interface of the *ServiceAbstractionLayer*. This interface should offer operations to create and send *ServiceRequests*.

SERVICEPROVIDERS

Services within a software system are related to each other. Normally, services belonging semantically together are arranged in subsystems. Once again, this is a simple application of *SEPARATE CONCERNS*. The *SERVICE ABSTRACTION LAYER* pattern uses the concept of *ServiceProviders* to aggregate *Services*. *Services* dealing with financial instruments could be associated with a *FinancialInstrumentServiceProvider*. The same concept applies for macro forecasts and market information (s. Figure 3). Thus, subsystems appear to the *ServiceAbstractionLayer* as *ServiceProviders*. In other words a *ServiceProvider* is an interface for the *ServiceAbstractionLayer* to get access to business s*ervices*. A *ServiceProvider* acts as a *FAÇADE* [Gamma+1995, p. 185ff] to a concrete subsystem. Moreover, it is a kind of *ADAPTER* as it connects a subsystem to the *ServiceAbstractionLayer*. Furthermore, a *ServiceProvider* administers *Services*. Therefore, it controls the lifecycle of its *Services* and offers mechanisms to register and deregister *Services*.

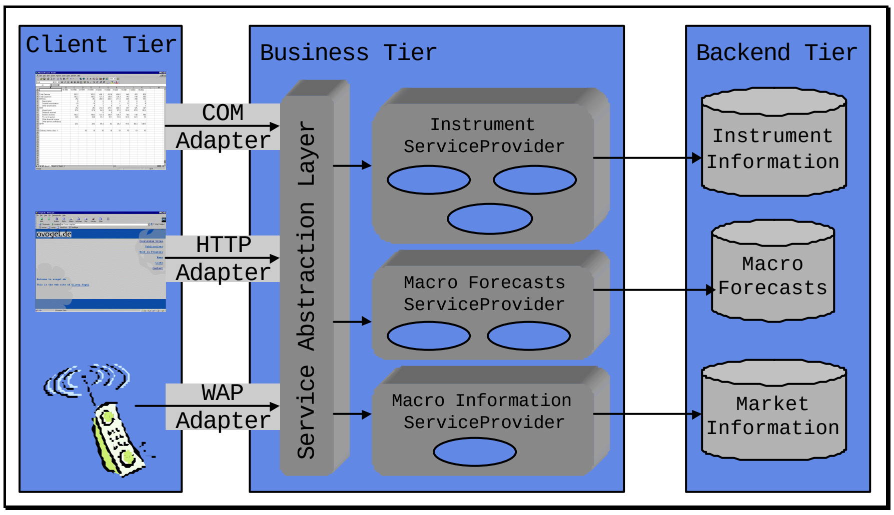

Figure 3: Architecture supplemented with ServiceProviders and ChannelAdapters

SERVICE

A *Service* encapsulates a distinct portion of functionality, which has to be offered by the software system. Thus, for example the calculation of the return on investment can be embodied in a *Service* object. This object would be part of the *FinancialInstrumentServiceProvider* mentioned earlier. Every Service must adhere to the same interface. This prerequisite requires flexible means to model *ServiceRequests* and *ServiceResults*.

SERVICEREQUESTS

Due to the fact that clients no longer know the concrete business services, they can only formulate their requests in a symbolic way. Thus, instead of programming against a concrete interface, clients specify *Services* using symbolic names. In order to enable *Services* to fulfill requests, clients must supply additional information in form of parameters. All this information can be encapsulated in *ServiceRequest* objects. A *ServiceRequest* represents an instance of the *COMMAND* pattern [Gamma+1995], p. 223ff.]. ServiceRequests can be flexibly modeled by using the *PARAMETERLIST* pattern [Sommerlad+1998]. This pattern describes how a list of name/value pairs can be used to hold the parameters of a method. Thus, instead of supplying all parameters explicitly a client simply supplies a *ParameterList* object containing the parameters.

SERVICERESULTS

ServiceRequests need to be designed in a generic way to allow that any kind of request can be expressed. The same requirement applies to *ServiceResults* as *Services* shall be able to return any kinds of results. Therefore, *ServiceResults* should be also designed according to the *PARAMETERLIST* pattern.

Optional Solution Building Blocks

The optional solution building blocks complement the *SERVICEABSTRACTIONLAYER* pattern by resolving further design issues.

CHANNELADAPTERS

Clients communicate with the *ServiceAbstractionLayer* over different channels. These channels require different communication protocols. The *ServiceAbstractionLayer* has to be able to retrieve *ServiceRequests* coming from these channels and to delegate them to appropriate *ServiceProviders*. As a service request is encoded according to a specific communication protocol it must be transformed into a *ServiceRequest* object before it can be sent to the *SALFacade*. For example, an HTTP request is made up of an URI followed by a list of name/value pairs. Such a request cannot be sent to the *SALFacade* directly as the façade expects a *Ser-*

viceRequest object. Therefore, an HTTP request has to be adapted to a ServiceRequest. This can be achieved by using a special HTTP adapter performing the transformation between an HTTP and a *ServiceRequest*. Generally speaking, clients can be connected to the *ServiceAbstractionLayer* with *ChannelAdapters*. A *ChannelAdapter* is an application of the *ADAPTER* pattern [Gamma+1995, p. 139ff.]. Figure 3 illustrates how clients can gain access to the *ServiceAbstractionLayer* respectively to business services over different *ChannelAdapters*. The *ChannelAdapters* are also responsible for translating *ServiceResults* into channel specific result structures.

SERVICEREQUESTHANDLERS

The *SALFacade* receives *ServiceRequests* from clients. To fulfill the request it has to find the appropriate *ServiceProvider*. The algorithm needed for that purpose should not be included into the façade directly as it might vary from time to time. For example, during testing you might want to log each incoming request and perform tracing on each method call. During production this could be troublesome, because of decreasing efficiency. To exchange the service request algorithms *ServiceRequestHandlers* should be introduced. These handlers contain the different request handling logic and offer the same interface. Thus, the fact that different algorithms are available, is hidden from the façade. The *STRATEGY* pattern [Gamma+1995], p. 315ff.] is a suitable candidate for that purpose.

Structure

There are different views on the *ServiceAbstractionLayer*. First of all, there is the client view. Clients should have to follow a simple programming model. This means that the exposed interfaces of the *ServiceAbstractionLayer* are easy to use and well defined. Furthermore, the number of interfaces clients depend on should be reduced to a minimum. The *SERVICEABSTRACTIONLAYER* pattern provides clients with the interfaces illustrated in Figure 4. As the interfaces are modeled in a generic way they will remain stable and so free clients from having to adapt to interface changes.

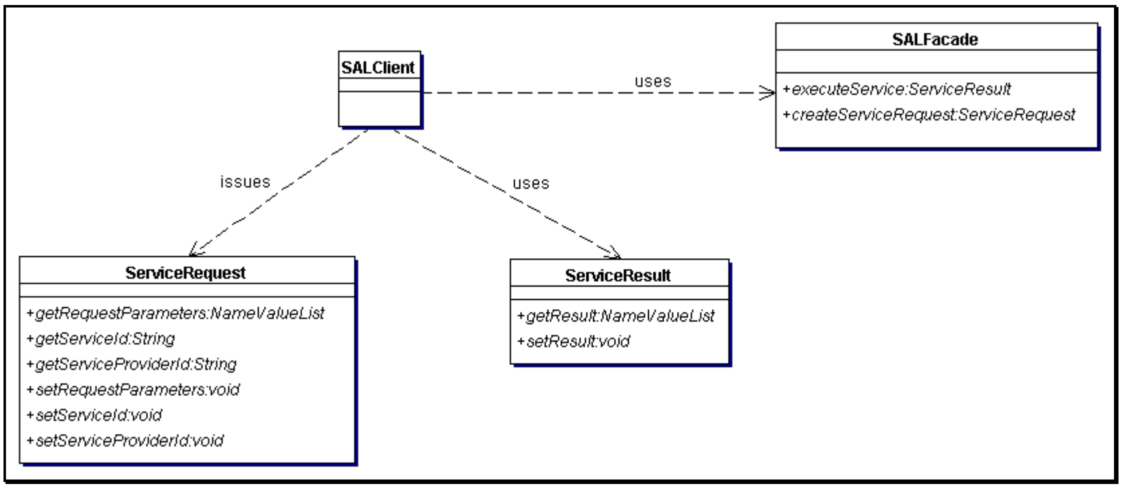
SALClient
uses
SALFacade
+executeService:ServiceResult
+createServiceRequest:ServiceRequest
issues
uses
ServiceRequest
+getRequestParameters:NameValueList
+getServiceId:String
+getServiceProviderId:String
+setRequestParameters:void
+setServiceId:void
+setServiceProviderId:void
ServiceResult
+getResult:NameValueList
+setResult:void

Figure 4: Client View on Service Abstraction Layer

A special type of *SALClient* is a *SALChannelAdapter*, which is responsible for the adaptation of a specific channel. There can be adapters for different channels, like HTTP or COM as shown in Figure 5.

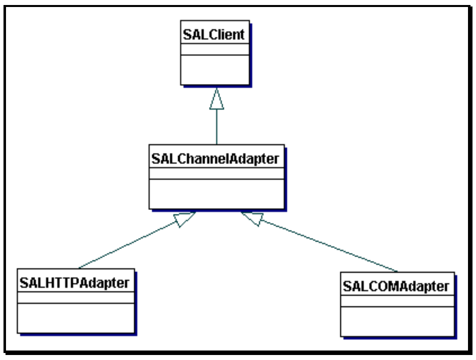

Figure 5: Channel Adapter Hierarchy

The internal structure of the *ServiceAbstractionLayer* is depicted in Figure 6.

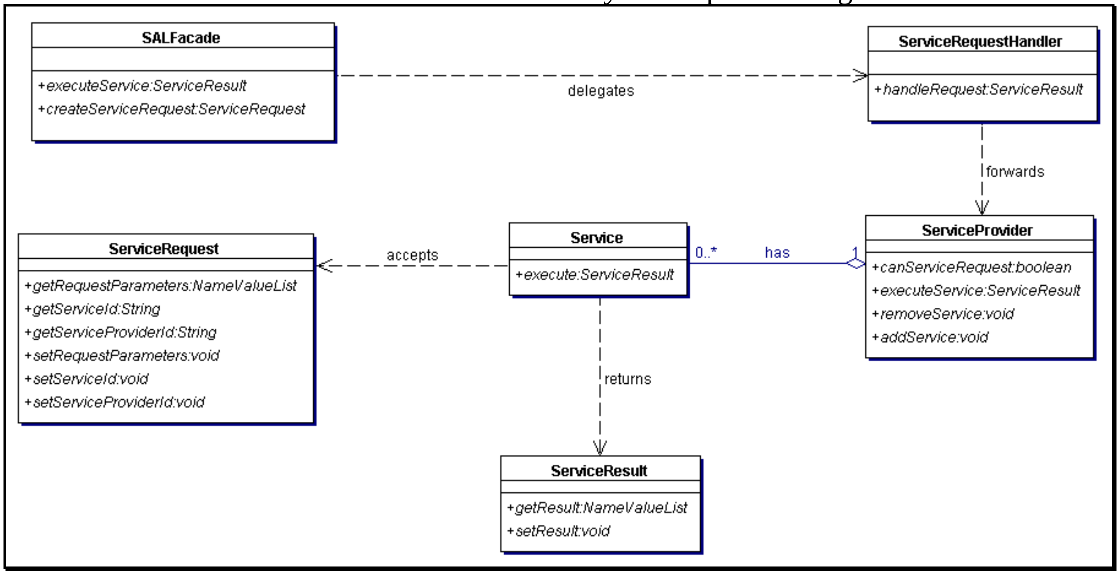
SALFacade
+executeService:ServiceResult
+createServiceRequest:ServiceRequest
delegates
ServiceRequestHandler
+handleRequest:ServiceResult
forwards
ServiceRequest
+getRequestParameters:NameValueList
+getServiceId:String
+getServiceProviderId:String
+setRequestParameters:void
+setServiceId:void
+setServiceProviderId:void
accepts
Service
+execute:ServiceResult
0..*
has
1
ServiceProvider
+canServiceRequest:boolean
+executeService:ServiceResult
+removeService:void
+addService:void
returns
ServiceResult
+getResult:NameValueList
+setResult:void

Figure 6: Internal Structure of Service Abstraction Layer

Dynamics

Figure 7 shows how a *ServiceRequest* is issued by a client and handled by the *ServiceAbstractionLayer*. The *ServiceProviderPool* acts as a registry of *ServiceProviders*. It knows which *ServiceProvider* can handle which *Service* and directly maps the input parameters to the corresponding *ServiceProvider*.

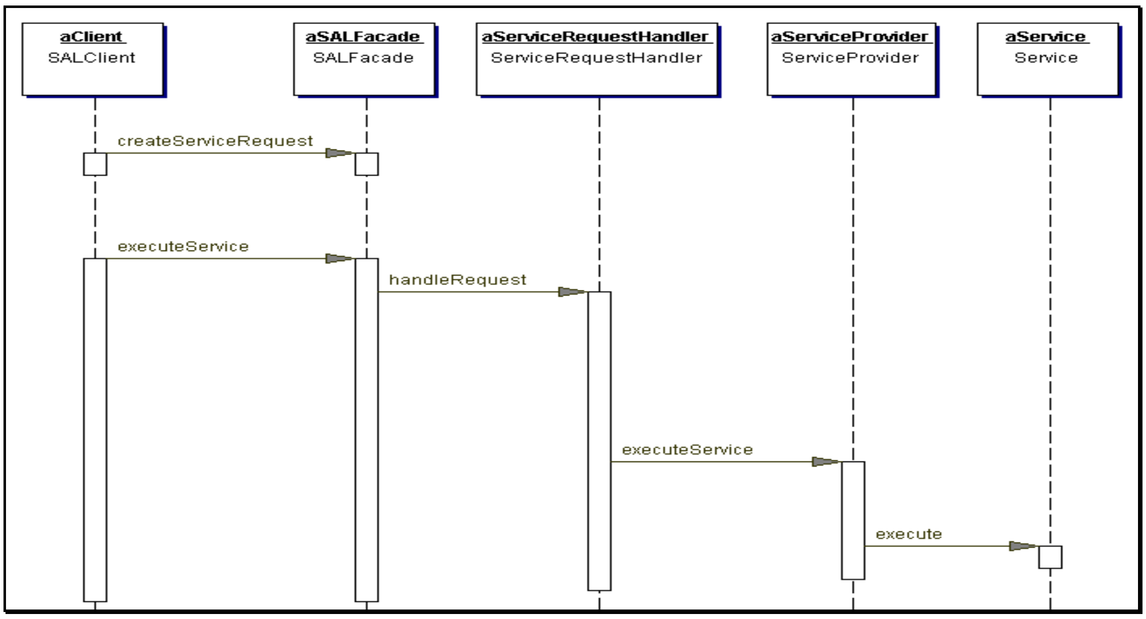

Figure 7: Issuing and handling of a ServiceRequest

Consequences

The *SERVICE ABSTRACTION LAYER* pattern has the following benefits and liabilities:

Benefits

1. ***SERVICE ABSTRACTION LAYER* separates business logic from communication logic** and allows the evolution of these different concerns independently.
2. **Business logic does not depend on specific channels.** Supporting a new channel does not require changes to the business logic and vice versa. Therefore, business services can be offered over any channel immediately as they are independent of concrete communica-tion protocols and models.
3. ***SERVICE ABSTRACTION LAYER* uses a high level of abstraction**. By abstracting concrete subsystems and method calls to *ServiceProviders* and *Services* new business logic can be integrated as the client's programming model is not affected. Thus, clients are isolated from implementation details.
4. **Clients can use a stable generic request interface,** which isolates them from the con-crete structure of the system. Therefore changes to the business tier do not influence cli-ents using the system.
5. **New business logic can be integrated without breaking the architecture.** If a new sub-system has to be supported, a new *ServiceProvider* can be implemented. The *Service-Provider* acts as a façade to the subsystem. Its services can be either represented as *Ser-vices* or the *ServiceProvider* can delegate the requests to the subsystem's components di-rectly.
6. **Service handling algorithms can be exchanged.** Besides the delegation of *ServiceRe-quests* incoming requests might also have to be logged for traceability. Furthermore, load balancing might have to be applied. To have the freedom to apply different strategies at run time different *ServiceRequestHandlers* can be implemented and plugged in, if neces-sary.
7. ***SERVICES* and *CHANNELADAPTERS* can be implemented indepdendently.** Developers with business expertise can develop business services and developers with communica-tion protocol know how can develop adapters.

Liabilities

1. **Another level of indirection may increase complexity**. SERVICE ABSTRACTION LAYER introduces a separate layer for the communication logic. This increases the number of layers in the architecture. Complexity may be enlarged, if the service access points of the different layers are not well documented. Therefore the effort to maintain and to understand the overall architecture might increase. However, not using a separate layer surely leads to a higher complexity as no clear separation of concerns is achieved.
2. **Weak typing problem**. SERVICE ABSTRACTION LAYER uses *ServiceRequests* to formulate requests. This descriptive approach leads to weak typing. Therfore, the correctness of a request cannot be determined at compile time. To eliminate the problem of weak typing a kind of meta repository could be used that specifies legal syntax and semantics for *ServiceRequests*. Document Type Definitions and XML Schemas are ideal candidates for that purpose. The same applies for *ServiceResults*.
3. ***SERVICE ABSTRACTION LAYER* may reduce efficiency.** As Service Abstraction Layer uses a generic mechanism to handle requests, requests have to be interpreted at run time. This interpretation requires extra effort and may reduce the performance of the overall system, if the responsible algorithms are not wisely implemented. For example, iterating over all *ServiceProviders* each time a request occurs is not very efficient. Therefore, efficient lookup strategies as mentioned in the Dynamics section have to be applied.

Implementation

This section will cover more technical design issues concerning the implementation of the SERVICE ABSTRACTION LAYER pattern.

IMPLEMENTING THE SERVICEABSTRACTIONLAYER

The *ServiceAbstractionLayer* contains the communication logic needed to handle *ServiceRequests* issued by clients over different channels. The layer itself can be organized on its own tier. Thus, there would be a communication respectively middle tier. This tier could be independent of specific applications and be used as a general middleware for any kind of application.

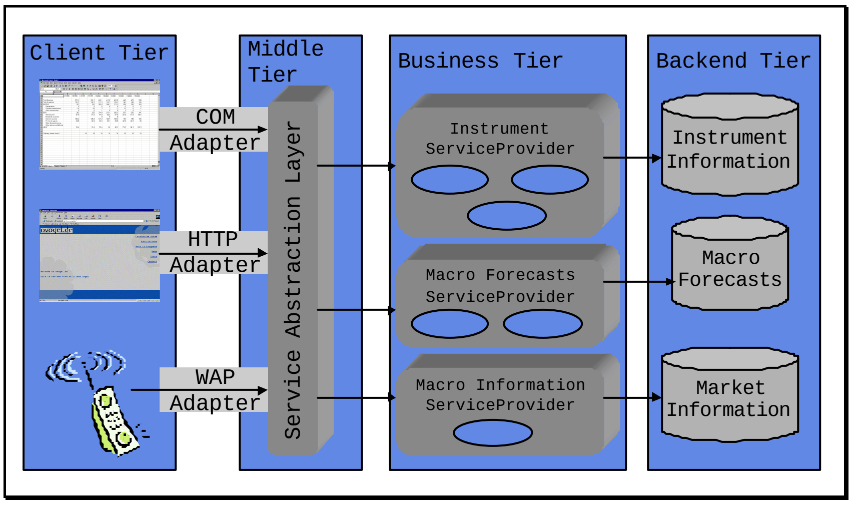

Figure 8: A separate communication tier

IMPLEMENTING THE SALFACADE

The *SALFacade* is implemented according to the FAÇADE design pattern. Client needs access to the *SALFacade* as it is the primary access point of the *ServiceAbstractionLayer*. The clients may only program against interfaces. Therefore, they should not instantiate objects directly. This can be achieved by applying a creational pattern like FACTORY METHOD. In our implementation of the SERVICE ABSTRACTION LAYER pattern the concept of home objects is used to get access to primary public abstractions. With public we mean objects which are used by clients of the *ServiceAbstractionLayer*. A home object is responsible for the lifecycle of its related objects. In the case of the *SALFacade* a *SALFacadeHome* offers static methods for the creation of *SALFacade* objects.

IMPLEMENTING A SERVICE

Services might be stateful. Therefore, clients mustn't call the same *Service* object concurrently. Instead, each client has to use its dedicated *Service* object. This prerequisite can be implemented by either creating a new *Service* object each time a *ServiceRequest* comes in or by caching *Service* objects in a pool and delegating the *ServiceRequest* to a not used *Service* object. To create clones of a *Service* object the *PROTOTYPE* [Gamma+1995, p. 117ff.] pattern can be applied. *Services* can be pooled by using the *INSTANCE POOLING* pattern [Voelter2001, p. 27ff.].

Services might need results of its former executions. Therefore, the *ServiceProvider*, as the administrator of *Services*, may need to get and set the state of a *Service*. This can be achieved by using the *MEMENTO* pattern [Gamma+1995, p. 283ff.]. *MEMENTO* allows to obtain the state of an object without breaking its encapsulation. The state can be copied to a *ServiceState* object, which can be stored for later use.

CONFIGURING THE SERVICE ABSTRACTION LAYER

The *ServiceAbstractionLayer* should be configured dynamically. Therefore, a *SALConfigurator* could read a configuration file containing the names of all available *ServiceProviders* and its *Services*, create the objects according to their class descriptions and attach them to a *ServiceProviderPool* respectively to the corresponding *ServiceProvider*. Figure 9 shows the configuration of the *ServiceAbstractionLayer*.

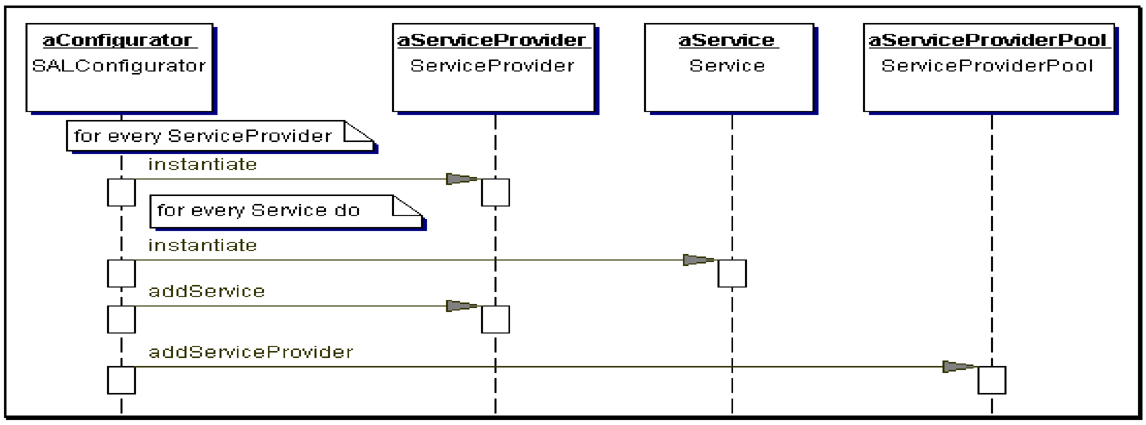

Figure 9: Configuration of the Service Abstraction Layer

Example Resolved

Figure 10 illustrates a simple, concrete *ServiceProvider* for financial instruments. It offers *Services* to aggregate and to retrieve specific instrument values.

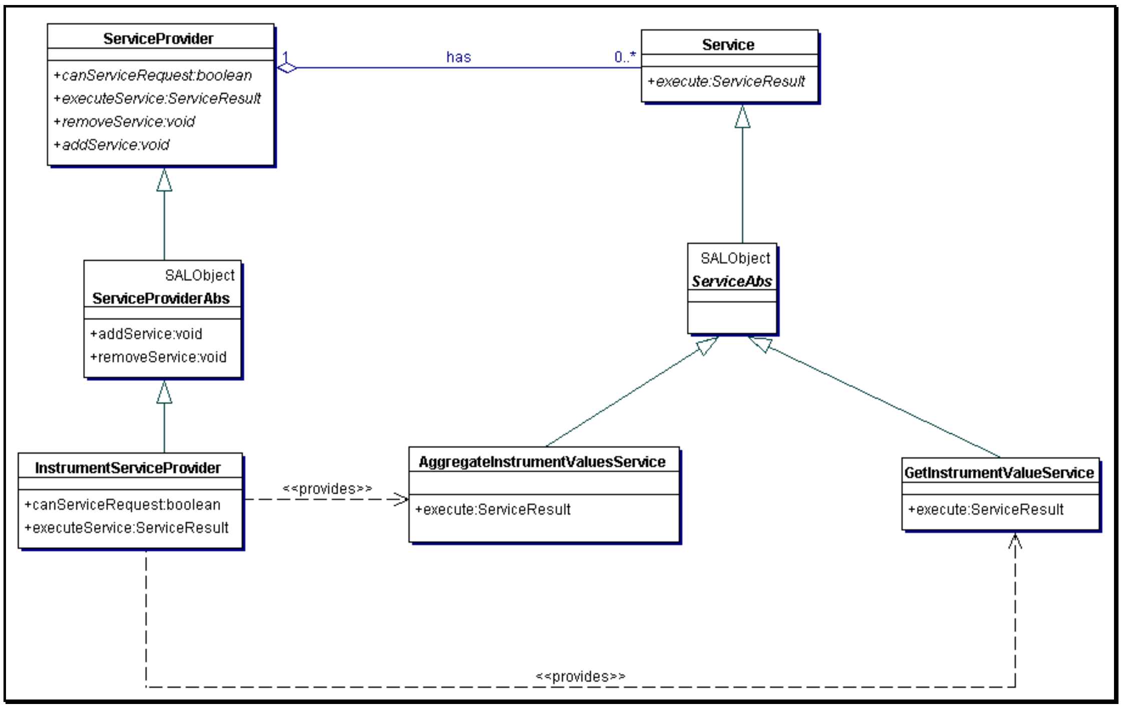

Figure 10: View on concrete Service Provider Subsystem

The following code snippets show, how an incoming HTTP request is converted into a *ServiceRequest* and send to the *ServiceAbstractionLayer*. The methods are part of a *SALHTTPAdapter*. The `request()` method requires a session object containing the HTTP parameters and builds a *ServiceRequest* by calling `buildServiceRequest()`. Afterwards, the *ServiceRequest* is issued to the *ServiceAbstractionLayer* in the `executeServiceRequest()` method by using `executeService()` of the *SALFacade*. The result is put into the session for later use.

```
public void request(JWSession inSession) throws JWException
{
 JWEvent lEvent = inSession.getCurrentEvent();

 ServiceRequest lRequest = buildServiceRequest(lEvent);

 ServiceResult lResult = executeServiceRequest(lServiceRequest);

 inSession.putData(lServiceResult,"ServiceResult");
}
```

```
protected ServiceRequest buildServiceRequest(JWEvent inEvent)
{
 SALFacadeIfc lSALFacade = SALFacadeHome.create();

 ServiceRequest lRequest = lSALFacade.createServiceRequest();

 Hashtable lParamTable = inEvent.getParams();
 Enumeration lNamesEnum = lParamTable.keys();

 NameValueList lNameValueList = new DefaultNameValueList();
 while (lNamesEnum.hasMoreElements())
 {
   String lParamName = (String) lNamesEnum.nextElement();
   String lParamValue = (String)
                        inEvent.getParameterValue(lParamName);

   NameValue lNameValue = new DefaultNameValue();
   lNameValue.set(lParamName,lParamValue);
 }

 lServiceRequest.setRequestParameters(lNameValueList);
 return lServiceRequest;
}
```

```
protected ServiceResult executeServiceRequest(ServiceRequest inServiceRe-
quest)
{
 SALFacade lSALFacade = SALFacadeHome.create();
 ServiceResult lServiceResult = null;

 try
 {
    lServiceResult = lSALFacade.executeService(inServiceRequest);
 }
 catch (SALException e)
 {
       // handle exception
 }
 catch (SALFatalException e)
 {
       // handle exception
 }
 return lServiceResult;
}
```

Known Uses

The design principles of the *SERVICE ABSTRACTION LAYER* pattern are used in the following architectures:

RIS

RIS is a research information system developed for a leading investment bank. It allows analysts to conduct their research and enables the publication of that research on different channels. RIS uses a *Service Abstraction Layer* to connect research services to the plat-form and to publish these services on different channels.

myBank

myBank is a portal offering personalized access to financial services for private and retail clients of a leading Swiss bank. *SERVICE ABSTRACTION LAYER* is applied to connect the dif-ferent financial service providers to the portal.

JWelder

JWelder [Systor2001a] is a framework for the development of business applications based on business components. It uses a *SERVICE ABSTRACTION LAYER* for the communication be-tween business components and their clients.

COMPASS/CDP C3

SYSTOR's *Common Development Platform (CDP)* [Systor2001b] provides a *Common Client Contract (C3)* to connect the business tier to the backend tier. *C3* abstracts avail-able data sources, like RDBMS, XMLDBMS or CMS and offers access to them over a

Data Abstraction Layer. The *Data Abstraction Layer* actually is an instance of the *SERVICE ABSTRACTION LAYER* pattern.

J2EE Connector Architecture

Similar to *C3* is the *J2EE Connector Architecture* [Sharma2000] intended to offer access to a wide range of *Enterprise Information Systems (EIS)*. *J2EE Connector Architecture* abstracts *ServiceProviders* respectively *Enterprise Information Systems* and arranges them on a special layer.

JNDI Service Provider API

In order to access different directory services *JNDI* abstracts concrete directory service APIs and offers a generic API for the transparent access to these *Services*. A concrete directory service is connected by implementing *JNDI Service Providers* [JNDI2000].

XOTcl

XOTcl [XOTcl2001] (XOTcl, pronounced exotickle) is an object-oriented scripting language based on MIT's OTcl. It is intended as a value added replacement for OTcl.

Used Patterns

This section describes the base patterns and how they are used in the *SERVICE ABSTRACTION LAYER* pattern. The patterns are separated according to their usage in either core or optional building blocks.

Core Solution Patterns

COMMAND

COMMAND is used to model *ServiceRequest* objects. Thereby any possible *ServiceRequest* can be expressed.

FAÇADE

FAÇADE isolates clients of the *ServiceAbstractionLayer* from its internal structure and therefore eases its evolution. Furthermore, *ServiceProviders* playing the role of a façade free the *Service Abstraction Layer* from the knowledge of the representation of concrete subsystems.

LAYERS

LAYERS introduces particular levels of abstraction as there is a distinction between the communication services contained in the Service Abstraction Layer and the business services contained in the subsequent business layers. Therefore *LAYER* increases separation of concerns.

PARAMETER LIST

PARAMETER LIST describes how a list of name/value pairs can be used to express parameters of a method [Riehle+2000, p. 2, Sommerlad+1998]. This pattern is used to design *ServiceRequest* and *ServiceResult* objects.

Optional Solution Patterns

ADAPTER

Clients shall be able to request services over different channels. In order to convert between channel specific requests and *ServiceRequests* adapters are used to support different channels. The *ChannelAdapters* are also responsible for converting the *ServiceResults* into channel specific results. *ServiceProviders* also act as adapters as they connect different subsystems to the *Service Abstraction Layer*.

FACTORY METHOD

FACTORYMETHOD encapsulates the creation of an object inside a special factory method. The *SALFacade* can provide a factory method, `createServiceRequest()`, to create *ServiceRequest* objects without having to know its concrete implementation.

INSTANCE POOLING

Instance pooling addresses the caching of objects in order to minimize the overhead related to object creation and destruction.

STRATEGY

The logic responsible for handling *ServiceRequests* should be exchangeable. Therefore, the STRATEGY pattern is used to model *ServiceRequestHandlers*.

PROTOTYPE

PROTOTYPE allows the creation of an object according to a prototypical instance. There is no differentiation between the object and its creator. *Service* objects can be modeled that way.

Acknowledgements

A lot of people have contributed to this architectural pattern with their reviews and comments. I'd like to thank all of them: Arif Chughtai, Dmitri Fertman, Jonathan Jude, Walter Kriha, Ralf Steck and Markus Völter. Special thanks go to Uwe Zdun for being an excellent shepherd. With his effort I was able to improve this paper significantly.

Bibliography

[Ayers+1999]: Ayers, e.a., Professional Java Server Programming, Wrox Press, 1999

[COMBridge2000]: JavaTM 2 Platform, Enterprise Edition Client Access Services (J2EETM CAS) COM Bridge 1.0 Early Access, http://developer.java.sun.com/developer/earlyAccess/j2eecas/download-com-bridge.html

[Gamma+1995]: Gamma, E. e.a., Design Patterns, Elements of Reusable Object-Oriented Software-Design, Addison-Wesley, Bonn 1995

[JNDI2000]: JNDI Service Provider API, http://java.sun.com/products/jndi/serviceprovider

[Kassem+2000]: Kassem N. and the Enterprise Team, Designing Enterprise Applications with the JavaTM 2 Platform, Enterprise Edition, Sun Microsystems, Version 1.0.1, 2000

[POSA1996]: Buschmann, F. e.a., Pattern-Oriented Software Architecture, A System of Patterns, Wiley, 1996

[Riehle1997]: Riehle, D., Composite Design Patterns, OOPSLA 1997, ACM Press, p. 218-228, 1997

[Riehle+2000]: Riehle, Tilman, Johnson, Dynamic Object Model, Submission to PloP 2000

[Sharma2000]: J2EE Connector Architecture Specification, Sun Microsystems Inc., Version 1.0, 2000

[SOAP2000]: Simple Object Access Protocol V1.1, W3C Note, 8th May 2000, http://www.w3.org/TR/SOAP/

[Sommerlad+1998]: Sommerlad, P. & Ruedi, M., Do-it-yourself Reflection, IFA Informatik, Zürich, Submitted to EuroPLoP´98

[Systor2001a]: JWelder, a business component foundation, http://www.systor.com/core/itservices/software/object_technology/jwelder/index.html

[Systor2001b]: An introduction to COMPASS, http://www.systor.com/edu/unizh/docs_ifi_enabling/framework_compass.pdf

[Voelter2001]: Markus Voelter, Server Side Components, A Pattern Language, V1.2, Jan 2, 2001

[W3C2001]: eXtensible Markup Language, W3C Architecture Domain, http://www.w3.org/XML/

[WSDL2001]: Web Services Definition Language, http://xml.coverpages.org/wsdl.html

[XOTcl2001]: XOTcl, http://www.xotcl.org