

EuroPLoP 2001 Design Fest

Designing a Three-Tier Architecture Pattern Language

Submission by Oliver Vogel, Systor AG, Switzerland
e-mail: mail@ovogel.de

Motivation

The design fest “Designing a Three-Tier Architecture Pattern Language” at EuroPLoP 2001 seeks to discover, document and consolidate often used design solutions in the development of three tier architectures.

Within this position paper I will introduce the *SERVICE ABSTRACTION LAYER* [Vogel2001] pattern as I think that it eases the development of three tier architectures by allowing to add and to offer new services in a flexible manner. I will prove this statement by discussing two projects I have been working on and by referencing other software architectures, which have applied this pattern. *SERVICE ABSTRACTION LAYER* fits into the category of *Adaptive Business Logic* outlined in the design fest description.

After talking about real life examples the essential aspects of the *SERVICE ABSTRACTION LAYER* will be presented.

Examples

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 increases over time. To resolve this force many software systems are build according to a three tier architecture. Such software systems consist of distinct tiers namely a client, a business and a backend tier [Kassem+2000, Ayers+1999].

In the following chapters software systems will be introduced, which have all been built according to the 3-tier architecture paradigm. Thereby the requirements, which have driven the architectural design will be presented as well as the resulting solution architecture.

Example 1: Building a research information system

Consider a research platform for analysts of an investment bank. Analysts are used to work with their favorite spreadsheet application. Therefore they want to access business services, which provide business codes, like calculations of earnings per share, market capitalization and EBIT from within their spreadsheet application. Furthermore these business services also represent added value for customers of the investment bank. Thus, 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. By having different clients communicating over different channels the business services must be accessible from these channels regardless of which protocol is used. This requirement implies that the business servers are separated from the different client applications and placed on a business tier. From their business services can access backend systems, like macro and financial information stored in databases. This results in a classical three tier architecture.

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 services. 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. 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: It combines communication logic with business logic and allows clients to use concrete interfaces to access services. These architectural decisions would 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. In order to avoid these bad consequences the *SERVICE ABSTRACTION LAYER* has been used as it allows to create an flexible and extensible architecture able to support different channels and additional services more easily.

Example 2: Building a business component oriented retail platform

A reliable retail platform is mission critical for the success of a retail bank. In this example the mentioned retail bank plans to migrate its existing C++ fat client retail application to a J2EE conform retail platform.

For this reason the logic of the C++ application has to be distributed onto the different tiers identified in the J2EE blueprint. On the client tier different types of clients, like web browsers and ultra light weight clients (ULC) should be supported. Clients accessing the business logic contained on the business tier shall be able to do so in a unified way. This implies a stable and generic programming model and a high level of abstraction. Therefore, the services of the retail platform, i.e. the opening of an account, have been encapsulated in business process components, which exhibit the same interface. Furthermore the business process components must adhere to the same communication protocol. This has been achieved by using process request and process result objects, which abstract the input and output parameters of method calls. Thus, instead of calling concrete methods directly, like `createAccount()` on a fictive *AccountManager* object, clients use a descriptive programming approach. In other words clients create a *ProcessRequest* object, set the name of the process they like to access, in this case `CreateAccount`, supply the necessary parameters in the body of the request and send it to the business tier. The business tier contains *ProcessProviders* which offer *Processes*. A central instance receives all requests and delegate them to appropriate *ProcessProviders*. As the programming model remains the same new *ProcessProviders* can be added easily and business logic can be adapted without affecting the clients of the business tier. The described solution is an instance of the *SERVICE ABSTRACTION LAYER* pattern. Using this pattern a *Business Process Layer* has been implemented, which allows to add and remove *ProcessProviders* without having to stop the server the layer is running on.

Further Examples

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 different 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 between 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 available 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.

Service Abstraction Layer

The following paragraphs are an extract of my submission to EuroPLoP 2001, which deals with the *SERVICE ABSTRACTION LAYER* pattern.

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 Overview

SERVICE ABSTRACTION LAYER shows how to develop a flexible and extensible architecture which is capable of providing any kinds of services and which can support different clients without the necessity of changing business services.

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.

Structure

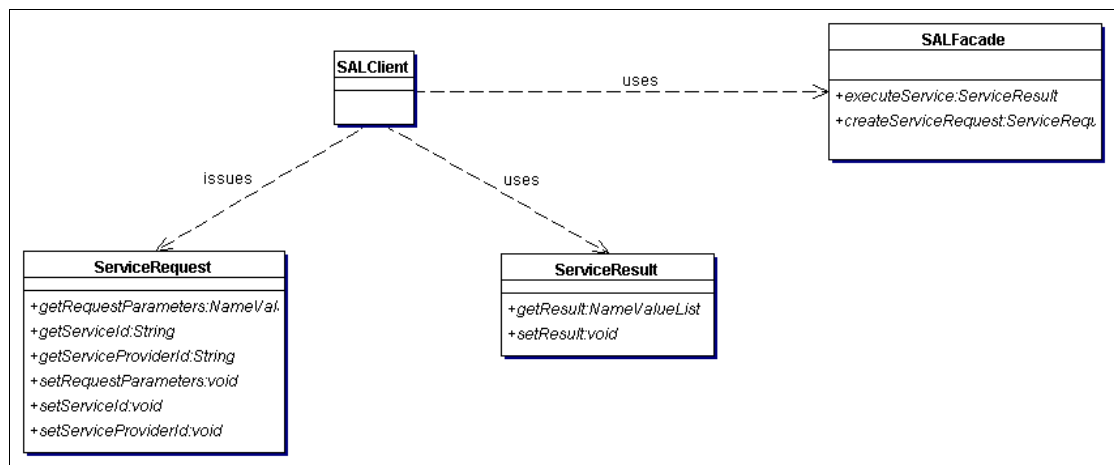


Figure 1: Client View on Service Abstraction Layer

The internal structure of the *ServiceAbstractionLayer* is depicted in Figure 2.

