Nicolai Josuttis

Designing a 3-Tier-Architecture

A Position Paper for the Design Fest of the EuroPLOP 2001

Nicolai Josuttis is an independent technical consultant who designs object-oriented software for the telecommunication, traffic, finance, and manufacturing industries. He is well known for his books and articles about C++, graphical user interfaces, object-oriented software development, and systems architecture. His book "The C++ Standard Library" got C++ book of the year 1999 and is published in German, English, and Japanese. He is a partner at System Bauhaus, a German group of recognized object-oriented system development experts. He is an active member of the C++ Standard Committee library working group. He is member of the editorial board of C++ Report.

I appreciate any **feedback**. Please send it to `nico@josuttis.com`.

NICOLAI M. JOSUTTIS
solutions in time

Nicolai Josuttis
Berggarten 9
D - 38108 Braunschweig
Germany

Tel: +49 5309 / 57 47
Web: http://www.josuttis.com
Email: solutions@josuttis.com

Background

At the moment I am working as system architect for a big German banking software application (the costs of the project exceed 100 Million DM). The crucial part of this system is to have an architecture so that the system is large but not complex. You can take this as an 3 tier architecture. However, it depends on how you see it. In practice these three tiers result into something like six tiers.

In this paper I will give an impression of my position why three tiers may be more than three tiers. And, as special topic, I will discuss a bit the way to implement the workflow processing and controlling layer inside this architecture.

Three and More Tiers

First, if you talk about three tiers you might consider this as separation of different layers:

- Three different physical layers, such as client, server and backend or mainframe.
- Three logical layers, such as presentation, core application and data storage.

These different views of layers overlap but don't match. As a result you might have something like six layers:

<table>
<tr><td rowspan="2">Presentation</td><td>User Interface Layer
(Client Side Presentation)</td><td>Client</td></tr>
<tr><td>(Server Side)
Presentation Layer</td><td rowspan="4">Server</td></tr>
<tr><td>Core Application</td><td>Business Layer
(Session Layer)</td></tr>
<tr><td rowspan="2">Data Storage</td><td>Entity Layer</td></tr>
<tr><td>Connectivity Layer</td></tr>
<tr><td>Data Layer</td><td>Backend</td></tr>
</table>

Thus, both the presentation layer and the data storage layer are part of two physical systems: The presentation layer has a part on the client and a part on the server side. The data storage layer has a part on the server and a part on the backend side.

This architecture matches with modern architecture models such as the J2EE platform: The server side of the presentation layer is the web server or web engine that combines the interface to the business logic with the elements of the (graphical) user interface. The server side of the data storage layer is typically a wrapper that hides the technical details of the backend (or enterprise information system). For example, this might consist of entity beans, a CMP engine, a host API or a CICS system.

Workflow Processing and Controlling

A key of business architectures is the question how to integrate the workflow into the system. This is a rather complicated task because to get a flexible maintainable solution you have to server different needs that in a way contradict each other:

- One of the most important design aspects of big systems is the separation of presentation and core application layer. Thus, the implementation of the business logic should be separated from any aspect of the user interface.
- The workflow processing is the key part that combines the presentation with the business logic. Depending on user input the following business logic gets called and processed, which results into different views. Thus, it's hard to separate presentation and business layer.

Of course, you can take the workflow as part of the presentation layer. This results into a solution, in which for each kind of presentation a new implementation of the workflow gets implemented.

We tried to avoid this behavior. Our goal was to implement the workflow only once, so that different presentation layers could use the same implementation. This, for example results into the ability to test the workflow by using an ordinary ASCII user interface.

Of course, still the workflow lies on top of the services and should be separated from the more stable parts of the business logic.

As a result you might end into even more layers: Between the presentation and the data storage you have:

- the workflow processing and controlling layer

- the core business layer
- the entity business layer

OK, here we have a three tier architecture again. But is is part of the core application layer on the server.

And More

Of course, there are other important issues of the design of these architectures. At least some of them I'd like to mention:

- The granularity of distributed objects turned out to be a crucial aspects of successful object-oriented systems. Two many projects started to distribute all objects and failed. However, what are the right numbers? This question highly depends on technical details, which more or less depend on the general communication techniques (such as CORBA or EJB) and the products that are used.
- It turned out that the architecture changed due to some security issues. That is, programmers shouldn't see aspects of ecurity keys, which are passed with calls between tiers. To handle this, the business logic gets part of a container, wrapping it.
- A very important aspect of a system comes from the question of the stable and unstable parts of a system. Unstable parts have consequences. They require more flexibility. Depending on the frequency of changes, the amount of effort comes more or less into play. It might or might not be OK to recompile the whole system. It might or might not be OK to distribute upgrades on clients. As a solution, interfaces may become more abstract or even generic. I've seen at least one system in which the frequency of changes did result into a totally different architecture. For example, the data storage became an OODBMS instead of an RDBMS.

According to the last item, if the frequency of changes and the technical abilities play an important role of system architecture, it might be very dangerous to formulate architectural patterns. At least, careful wording is required so that the match of some pattern contexts doesn't end up in an architecture that is based on these patterns but totally inappropriate.