Position Paper for EuroPLoP 2001 Workshop

3-Tier Pattern Language

(c) 2000 Markus Völter, MATHEMA AG, Germany

markus.voelter@mathema.de

Introduction

This position paper contains two parts. In the first part, I want to point out that three tiers are a more or less arbitrary (or at least, coarse-grained) number of tiers. In reality, there are usually more than three. The second part presents a couple of design pattern thumbnails for

Part 1: Three (or more?) Tiers

First, a distinction has to be made between logical tiers and phyiscal tiers:

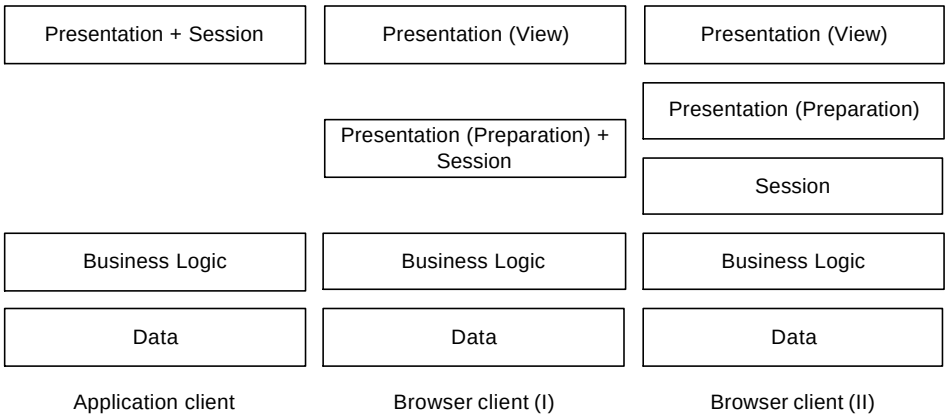
- **Logical tiers** describes the separation of the application into *potentially distributable* parts.
- **Physical tiers** describe the actual deployment of these logical tiers onto different machines.

Logical Tiers

Logical three tier architectures are quite common in today IT landscape. The tiers are usually Presentation – Business – Data. However, as the following illustration shows, usually there are more than three tiers, depending on the scenario.

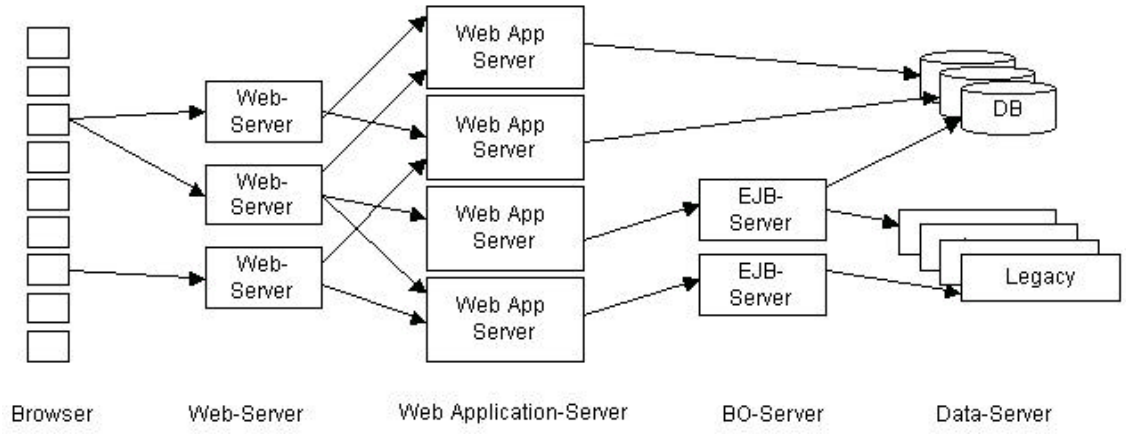
The "Application client" structure is a typical application that can keep track of the state of the program. Only if real business-logic has to be used, the business logic tier is accessed. The business logic tier itself uses the data tier for storing its persistent state.

The "Browser client" scenarios are different. A browser can only show content and collect user input (much like a 3270 terminal in the old days). The content that should be displayed has to be prepared in order for the browser to display it. This is the responsibility of the webserver. In addition, the browser itself cannot keep track of the application state. Therefore, this state has to be managed somewhere else. Usually, as in "Browser client (I)" this is done by the web server's session management. If this is not possible (e.g. because a system features several web servers, and therefore the session state has to be kept in a central place), the session management can also be achieved by a separate layer (as in "Browser client (II)"), usually this is technically a part of the business logic.



Phyiscal Tiers

To simplify some technical aspects such as load-balancing or failover, deploying typical three (or more?) tier applications is usually done using the following phyisical tiers.



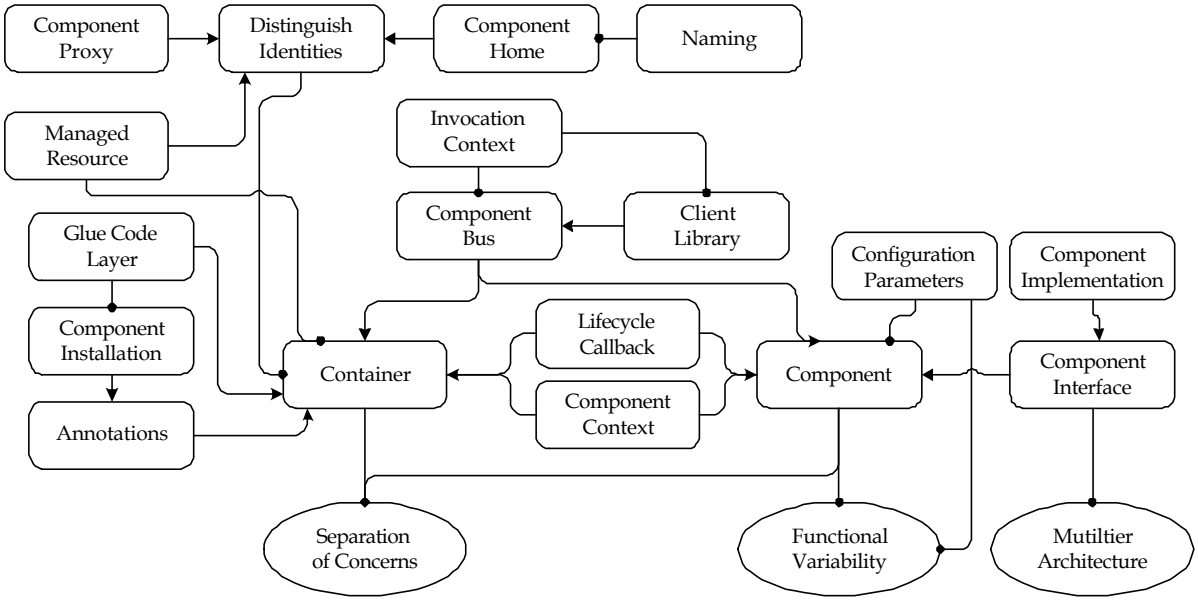
In the illustration, the system is partitioned in several ways:

- The static HTML content is provided by "normal" web servers.
- Dynamic content is forwarded to web application servers which directly access a database for catalog data, user preferences, etc, usually for read-only access. They also manage sessions.
- Whenever interactive business logic is needed, the web application servers use business object servers in the next layer, which in turn access the databases or legacy systems.

This configuration ensures that the load is handled as early in the system as possible. The BO (Business Object-) servers are only accessed when they are really needed, this is also true for the web application servers. As a consequence, each of the layers can be scaled individually according to the usage profile encountered in real life.

Part 2: Business Layer patterns

The business layer of a 3 Tier System is usually implemented using a *Component Architecture*. While, for reasons of brevity, we cannot describes the characteristice in detail (these details are in [1] and [3]) a rough overview chart should at least give some hints for people who have been working with these kinds of systems:



Based on such an architecture, applications can be built. Over time, several patterns have emerged for building these kinds of applicatios. They are described in [2] and [3], and we will present a collection of *thumbnails* to give you an idea. Some of these patterns are specific to the EJB component model, but they can be applied to other architectures, too.

Session Bean Façade

Usually the access to Business Entities require that certain methods must be called in some defined order or within the same transaction. Also methods of some Entities Beans might only make sense in conjunction with other Entity Beans. However, you cannot define that a transaction should span multiple methods of an Entity Bean or different Entity Beans.

Therefore: Provide a SESSION BEAN FAÇADE i.e. a Stateless Session Bean that accesses Entity Beans. As both Components are located on the server only server internal communication takes place and no network overhead will occur. In addition, one operation in the Stateless Session Bean might call several methods on different Entity Beans. A transaction can then span all methods called on the Entity Beans. Avoid direct access of Entity Beans whenever possible.

Type Manager

Entity Beans provide concurrency synchronization, pooling, and an extensive lifecycle management. These features impose some performance penalty. For applications, where

many clients access a set of entities concurrently and these additional features are used, the performance penalty is acceptable.
However, in systems where these features are not required, Entity Beans don't deliver optimum performance. This is particularly true for highly concurrent read access to your data.

Therefore: Do not use Entity Beans to represent the records in the database. Instead use a SESSION BEAN to work directly on the entities in the database. Use VALUE OBJECTS to represent the data in each call, together with the PRIMARY KEY to identify the entity to which the operation should apply.

Event Listener

You don't want to have is mutual, or circular, dependencies because this has very bad consequences regarding maintenance and deployment, and it reduces the reusability of the COMPONENTS. Dependencies should always be unidirectional only – resulting in a "layered system" [POSA]. In layered systems dependencies exist only in one direction, namely from higher to lower layer. Lower layers are never allowed to directly access higher layer. But then, how do you communicate information from lower layers to higher layers?

Therefore: Ensure that direct dependencies exist only in one direction. In this direction, use direct method invocations. For the way back, use event-like communication based on a generic event-receiver interface. Receivers of events register with the producer. Receivers are notified if an event is published.

Business Component

It is often hard (and sometimes conceptually impossible) to "press" the complete functionality for a requirement into one EJB . Thus, you end up with a set of EJBs which are always used together as a group. However, there is no "formal" grouping for these COMPONENTS, and clients have a hard time because they have to operate on many instead of one COMPONENT.

Therefore: Use an abstraction called a BUSINESS COMPONENT which consists of several EJBs internally. Distribute and release the COMPONENTS always as one "subsystem". Provide a *Facade* COMPONENT [GoF] which servers as the single access point to the whole business component, simplifying client access. This facade might use WEAKLY TYPED INTERFACES to simplify reuse and integration.

Value Object

Each call to an Entity Bean takes a relatively long time. This is because a remote invocation is necessary and expensive. Also for every call of a beans operation the CONTAINER has to check security and transactional settings. So many invocations of getter operations are therefore inefficient.

Therefore: Add an additional class which contains all the attributes you want to get at the same time. Also add a FACTORY method to your bean that returns an instance of the new class that holds all the data. This is a so called VALUE OBJECT. Whenever you need to retrieve all the

information use the FACTORY method of your Entity Bean to get all the data with only one call. Of course VALUE OBJECTS have to be transmitted by value.

Weakly typed interface

Adding new functionality to a COMPONENT requires changes in the COMPONENT INTERFACE. This usually also requires recompilation of the clients in order use the new features. Redeployment is also necessary, because the contents of the CLIENT LIBRARY will change (new interface classes, etc.). This is unacceptable in many applications, especially high-availability systems.

Therefore: Create a generic interface which has an operation that allows to generically specify commands, including parameters. The component implementation can then interpret this command and return the results accordingly. To make sure the client can really work with new operations, reflective features must be added to allow the client to query the component for available commands.

Attribute List

Your components feature a set of attributes which are accessible through hard-coded accessor operations, such as setter/getter pairs or VALUE OBJECTS. However, depending on the use of the COMPONENT, additional data has to be stored with the COMPONENT, i.e. additional attributes are necessary. You do not want to change the COMPONENTS implementation every time a new attribute is required, perhaps only for a specific use case.

Therefore: Provide a COMPONENT with an attributes list, a set of name-value pairs, which can be accessed by *setAttribute(name, value)* and *getAttribute(name)* operations (or their bulk-accessor optimizations).

Literature and Online Resources

[1] *Markus Völter*, **Server-Side Components – A Pattern Language**, submission to EuroPLoP 2001, see www.voelter.de/cpl

[2] *Markus Völter, Alexander Schmid, Eberhard Wolff*, **Building EJB applications – a Collection of Patterns**, submission to PLoP 2001, see www.voelter.de/book

[3] *Markus Völter, Alexander Schmid, Eberhard Wolff*, **Server Component Patterns – Architecture and Applications with EJB**, to be published by Wiley in early 2002