1 Layered Architecture

1.1 Introduction

Most enterprise applications are composed of a large number of components throughout the environment. In addition to that, these components often have different levels of abstraction. In order to adhere to the ever changing business requirements, how can we come up with an application architecture that supports:

- Maintainability
- Reusability
- Scalability
- Robustness
- Security

1.2 Best practices

This application architecture should:

- Promote separation of concerns among the components in the architecture (Separating the user interface from the business logic , and the business logic from the data store)
- Allow for changes to one part of the architecture to have a minimal impact on others parts. For example, changing a web interface should only have an impact on the user interface part of the application.
- Promote reuse; components should follow the Low Coupling / High Cohesion pattern.
    - Individual components should be cohesive.
    - Unrelated components should be loosely coupled.
- Allow for the different components of the architecture to be deployable independently
- Allow for the different components of the architecture to be testable independently
- Support the physical distribution across different tiers

1.2.1 Choose a layered architecture

We need to separate the various components within the architecture into logical and technical layers.

Components in a particular layer:

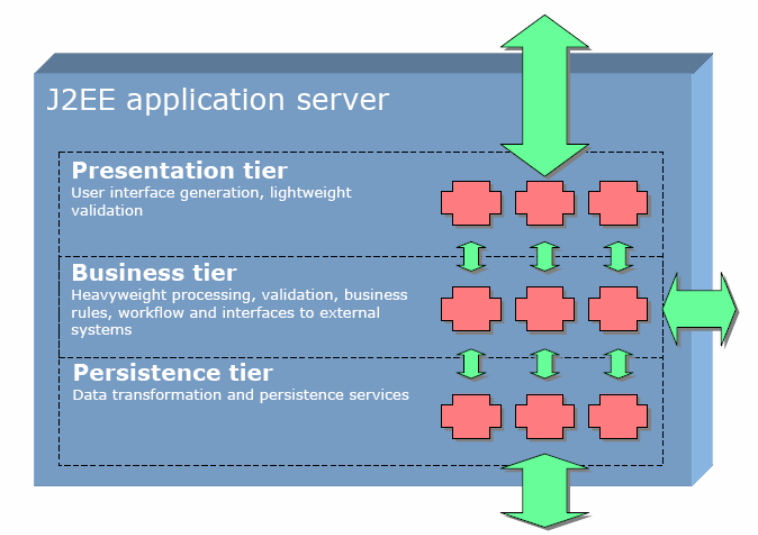
- Should have the same level of abstraction.
- Should only have a dependency to
    - other components in the same layer
    - components in the layer directly underneath.

Layers help us to control the dependencies and clearly define the boundaries within the architecture.

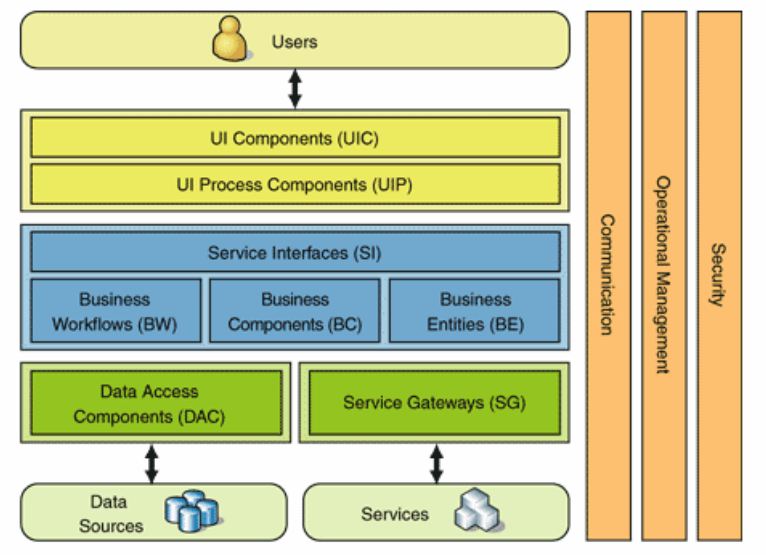
On a high level, most enterprise application architectures will be decomposed in 3 layers.

- Presentation Layer
- Business Layer
- Data Layer

A typical J2EE 3 tier architecture will be depicted like this:

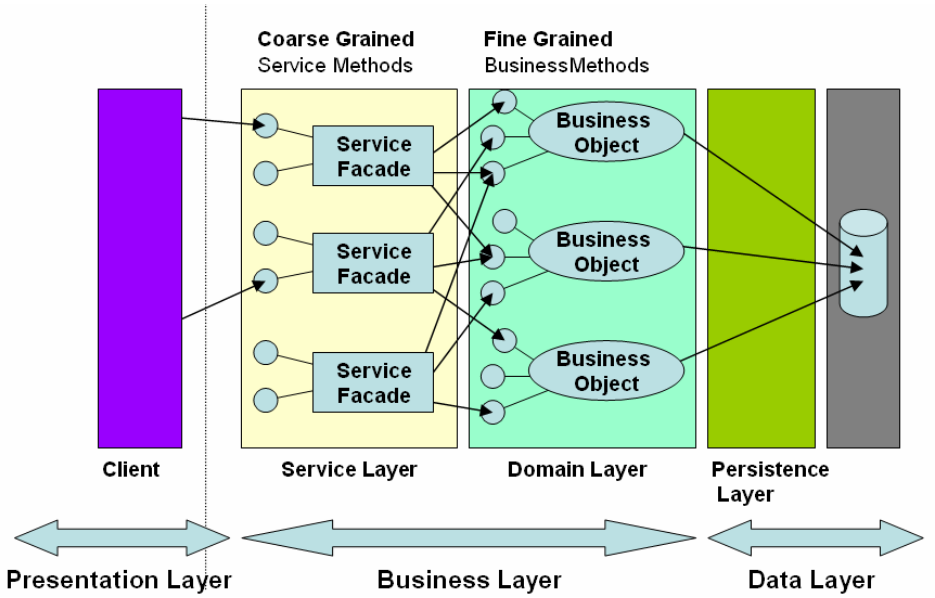


A typical Microsoft.NET 3 tier architecture will look like this.



As you can see, often different terminology is used to express the same thing. (data tier/persistence tier, business tier/service tier,….)
In this section, we'll discuss the following layers that should be present in any enterprise architecture

- Presentation Layer
- Service Layer
- Business Layer
- Persistence Layer
- Domain Layer



1.2.2 Apply design patterns to layer the architeture

A layered architecture is typically achieved by applying the following high level patterns

1.2.2.1 Layer Supertype [Fowler03]

If the components in the layer share a set of common behaviors, you extract those behaviors into a common class or component from which all the components in the layer inherit.

- eases maintenance
- promotes reuse
- reduces dependencies between layers

1.2.2.2 Abstract interface

When lower level components are called by higher level components, we can define an abstract interface for those lower level components. This allows those lower level objects to evolve without having an impact on the higher level components.

1.2.2.3 Layer Facade

When multiple objects in a layer need to be exposed in a unified way, it's often advisable to expose the entire layer via a layer façade. This allows us to hide the internal complexities of each individual object within the layer. (Rather then applying the abstract interface pattern on each individual object). [Gamma95]. Since other

layers will access the façade directly, it reduces the coupling between individual layers.
Serious thought needs to be put into the design of this high level interface, as it will be the entry point for a lot of other components and should be relatively stable.

2 Domain Layer

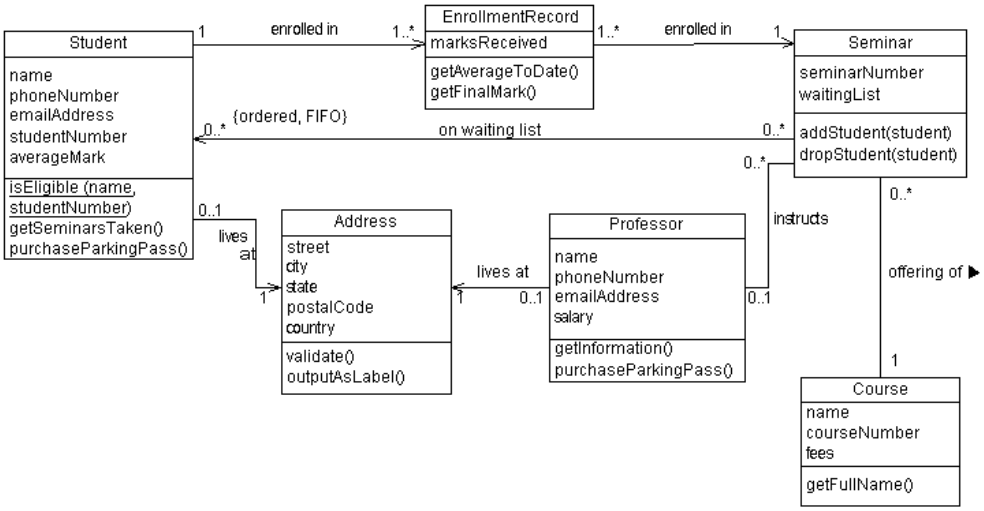
2.1 Introduction

Implementing the business logic of an application is one of the most important aspects during the development cycle. We should strive to keep all business logic centralize in one part of the application.

2.2 Best practices

2.2.1 Introduce a domain layer in the architecture.

Domain objects represent entities in the application domain, encapsulating both the data and behavior (Fowler). According to OO principles, this is the preferred place to store business logic.
A domain layer defines a set of (possibly interconnected) domain objects.



The domain layer is the most important layer of the application, as it encapsulates all business rules and data associated with the application.
This is very important, as often we come across domain layers whose domain object represent mere data, and no (or very little) behavior. This is referred to as the anemic domain model (Fowler). An anemic domain model is reliant on other layers to implement the behavior or business logic of the application.
In order to take full advantages of object oriented programming, the domain object should also implement the behavior instead of just the data. This is the place where the real power of OO comes to play.

The domain layer contains

- Business objects representing entities of the application domain
- Relationships between business objects
- Business objects containing both data & behavior
- No dependencies with upper layers (services, presentation)

2.2.2 Apply design patterns on the domain layer

2.2.2.1 GRASP

GRASP is a collection of fundamental principles in object design and responsibility assignment, expressed using patterns. GRASP stands for 'General Responsibility Assignment Software Patterns'.

GRASP defines the following patterns:

Information Expert

A general principal of object design and responsibility assignment?

Creator

Who creates?

Controller

What first object beyond the UI layer receives and coordinates a system operation?

Low Coupling

How to reduce the impact of change?

High Cohesion

How to keep objects focused, understandable, and manageable?

Polymorphism

Who is responsible when behavior varies by type?

Pure Fabrication

Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling?

Indirection

How to assign responsibilities to avoid direct coupling?

Protected Variations

How to assign responsibilities so that the variations or instability in the elements do not have an undesirable impact on other elements?

2.2.3 Utilize the full power of OO

The Domain layer is the place to fully utilize the power of OO. Make use of inheritance , polymorphism to the fullest.

2.2.4 Avoid the anemic domain model

The Domain layer should implement the behavior of the objects. Domain objects should not be merely data.

2.2.5 Minimize the dependencies with other layers

The Domain layer should have minimal dependencies with other layers or libraries.

2.2.6 Unit test the domain layer

The Domain layer is ideally suited for unit testing. Testing is an often underestimated part of the development process.

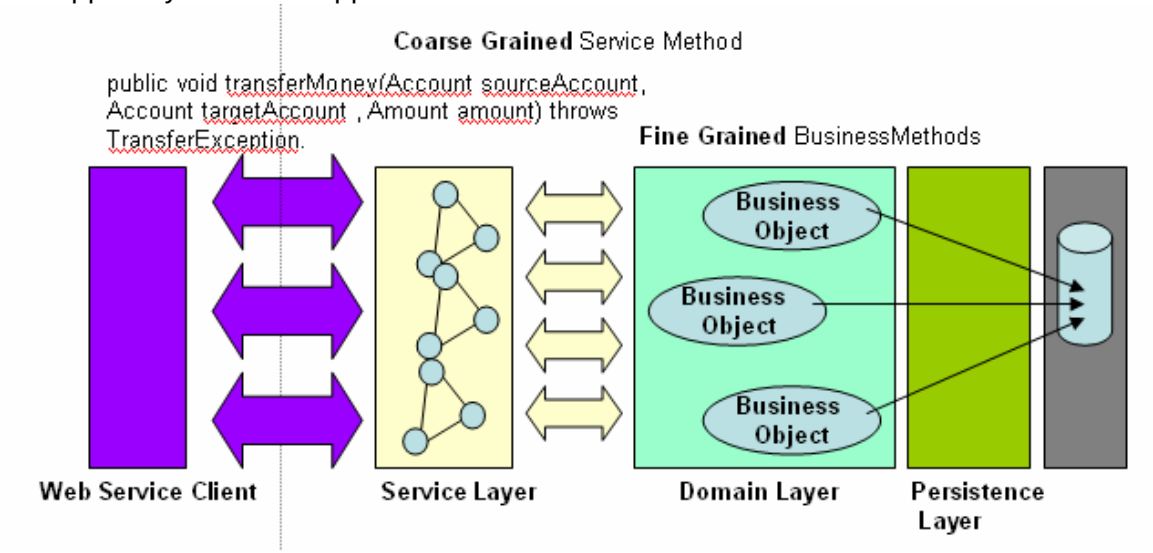
3 Service Layer

3.1 Introduction

The methods exposed by the domain objects in the domain layer are typically very fine-grained operations. In order to implement a particular use-case, one requires several domain objects, and one also has to know how to execute the operations on those domain objects in order to implement the use-case. Since domain logic isn't usually use-case specific, we need a place to store this use-case logic or service logic (loosely coupled business logic).
We need a way for our application to expose a more course-grained business interface to our application so that it can be used by the presentation layer, or possibly even other applications.

3.2 Best practices

Introducing a service layer into the application architecture acting as the primary interface to expose loosely coupled and relatively course grained business logic to the upper layers of the application.



This service layer is primarily responsible for

- Exposing a coarse grained business interface
- Separating use-case logic (or service logic) and domain logic
- The implementation of use-cases.
- Manipulating the domain objects found in the domain layer
- Transaction management
- Security Management
- Data storage / retrieval (through delegation via the persistence layer).

Typically, a service layer contains several service interfaces that group together similar use-cases. For example, we could have an AccountManager service interface that exposes the following methods:

```
public void transferMoney(Account sourceAccount, Account targetAccount , Amount
amount) throws TransferException.

public void withdrawlMoney(Account sourceAccount,Amount amount) throws
InsufficientBalanceException
```

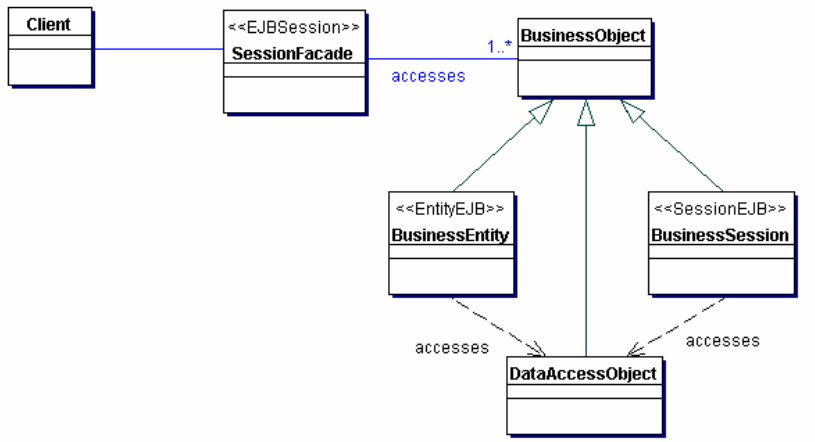
A typical implementation of the `transferMoney` method could be

- Open database connection
- Retrieve the source account
- Check if the user can transfer money from this account
- Check if the user has enough credit available on the account
- Check if the targetAccount is applicable for a transfer
- Transfer the amount from the sourceAccount to the targetAccount in a single transaction
- Clean up resources.

The service interface hides this complexity by exposing a coarse grained business interface to the caller (in this case, a simple transferMoney method that is relatively easy to comprehend). In that sense, the service interface acts as a façade, providing course grained operations. This service method has a clear signature in terms of input/output parameters and exceptions.

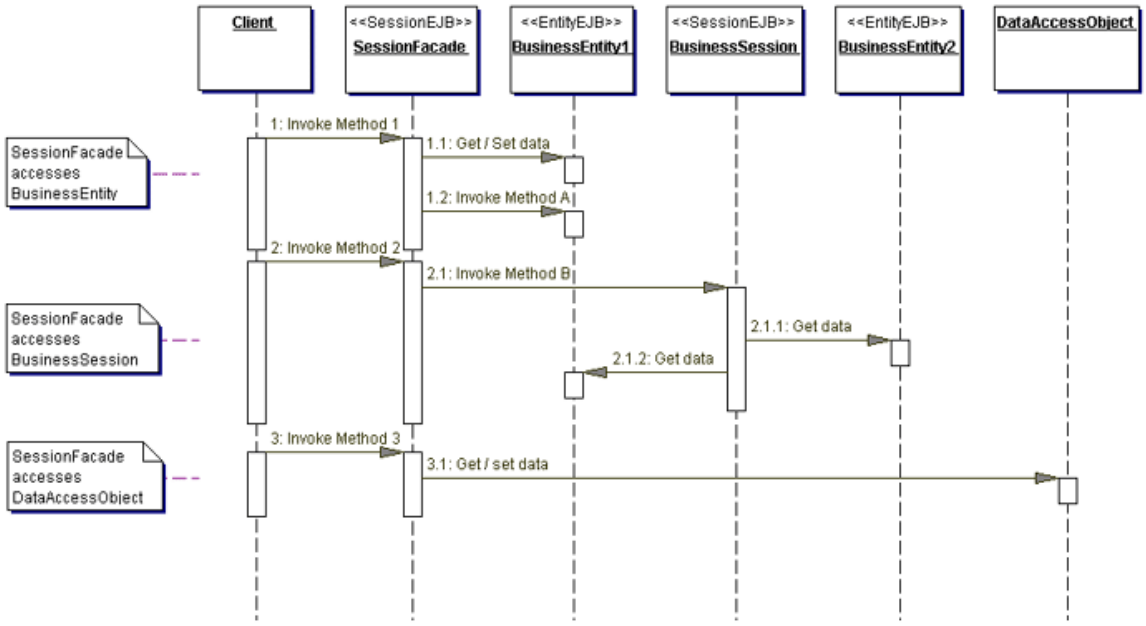
3.2.1 Introduce a session façade in the architecture

The service layer is typically implemented using the session façade pattern.
The idea is to provide the client with a uniform way to access use-case logic. The service layer can contain multiple faces, where each façade combines a set of related use-cases.



UML class diagram depicting a stateless session bean acting as a service facade

The session façade acts as the entry point for the client, and encapsulates the orchestration that occurs when calling an operation on the service layer.



UML sequence diagram depicting a client accessing a service facade

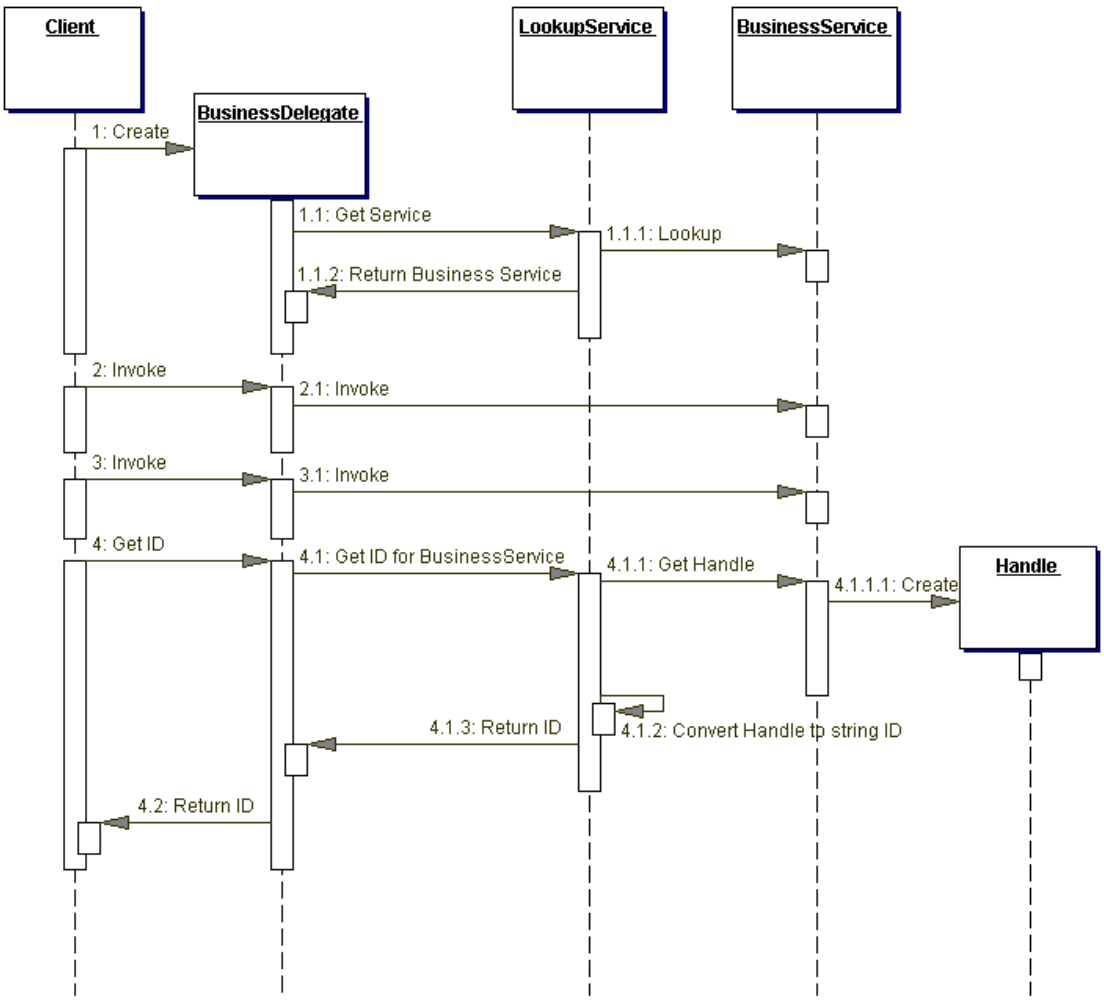
As the session façade exposes a course grained interface, network roundtrips can be minimized, as most of the orchestration, and access to the lower level domain methods will occur behind the scenes.

3.2.2 Introduce Business Delegates in the architecture

Presentation tier clients need a clean & simple way to access business services.
In order to avoid exposing the actual implementation of the business service, and to reduce the coupling between the consumer & producer of business services, we introduce a Business Delegate.

The business delegate makes sure that

- Changes in service implementations don't have an effect on client code.
- The coupling between the consumer & producers of business services is minimized.
- Technical details such as lookups and access control can be encapsulated in the business delegate

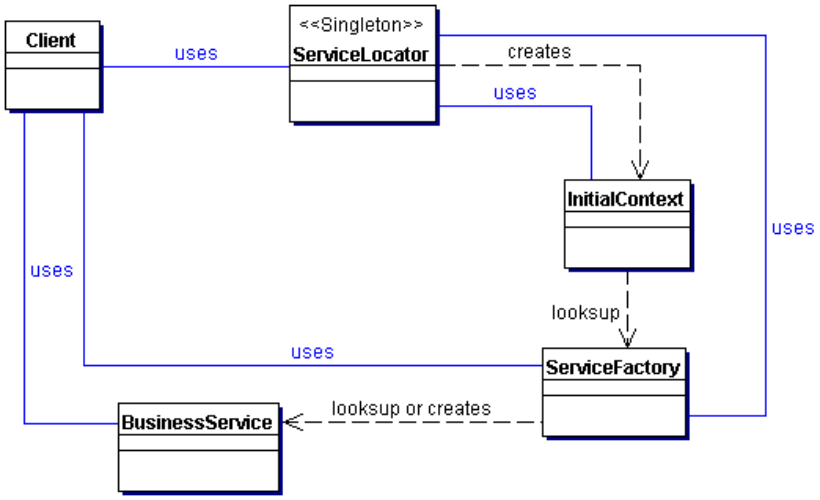


UML sequence diagram depicting a client accessing a business delegate

As you can see, the client never accesses the business service directly. Instead, it creates a business delegate that performs the actual lookup of the service, and delegates the request coming from the client to the actual business service.
The business delegate will use the service façade when a client issues a request. In that sense, the business delegate acts as a proxy and delegate client requests to the service façade, further decoupling the client from the service logic.

3.2.3 Introduce a Service Locator in the architecture

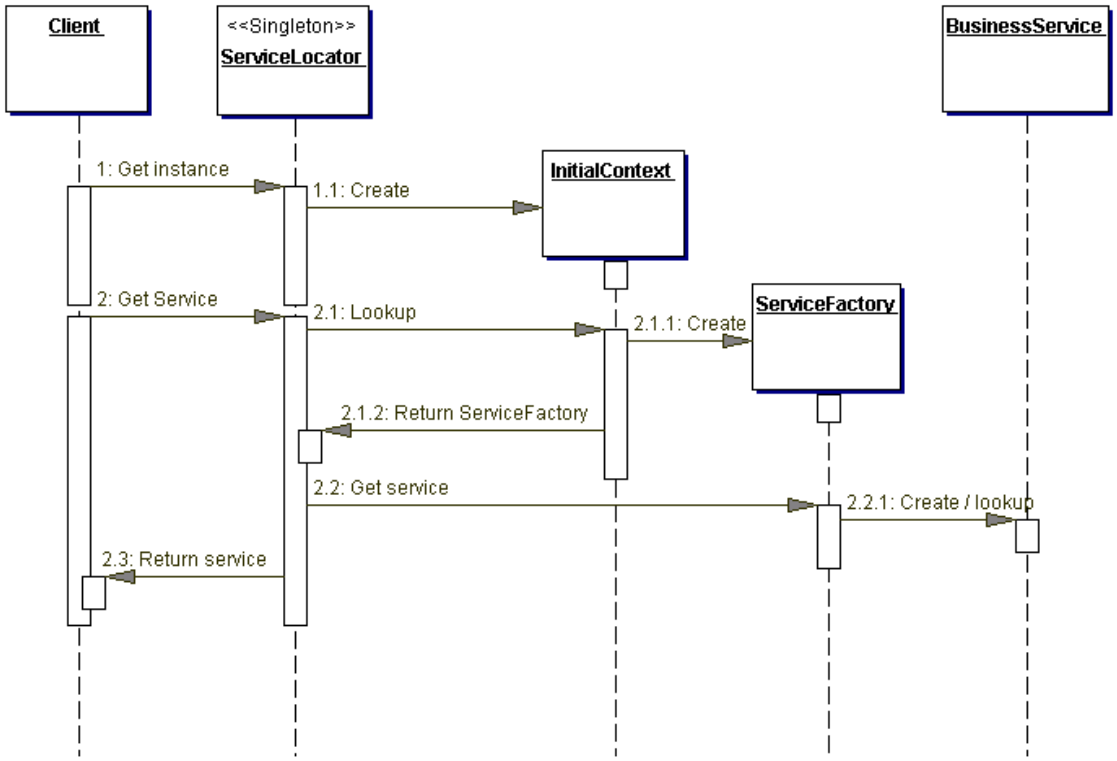
Often times in our applications, we need to access resources such as JDBC connections, enterprise java beans, JMS resources…
In order to provide our application with a uniform way to access these resources, we introduce the Service Locator. The Service Locator shields way all technical issues such as resource lookups (via JNDI, registry …), object recreation, caching ….
It can be used in different layers throughout the architecture. For example, the service layer can use the Service Locator to access a PersistenceManager via JNDI, capable of providing transparent persistency.
It can retrieve a reference to a JMS queue to provide asynchronous messaging.
The Business Delegate can use a Service Locator to retrieve a particular service façade upon a client request.



UML class diagram depicting the actors of the service locator pattern

As you can see in the sequence diagram below , the primary focus of the ServiceLocator is to return a particular resource to the client , while hiding the technical complexities of retrieving the resource.

The client retrieves an instance to the ServiceLocator (typically implemented as a singleton) and requests a particular resource of service. The Service Locator will then perform the low level plumbing to retrieve the resource, and return it to the client.



UML sequence diagram depicting a client accessing the service locator

3.2.4 Apply Inversion of Control (IoC)

An alternative to using the Service Locator Pattern is to implement an Inversion of Control mechanism.
Inversion of Control is all about dependencies. Objects usually collaborate with other objects and thus form all kind of dependencies among them. Identifying those dependencies is a key element in constructing viable application architectures.
An object that is dependant on another object usually has some kind of reference to it. This creates a dependency between the 2 objects.
Inversion of Control is typically referred to as dependency injection. Dependency injection is all about preparing an object so that it can be properly instantiated and used within the application. (creating & wiring objects).

This means if object A needs a reference to object B in order for it to do something useful, object B should be injected into object A before object A is put into use.
The injection itself is typically done by an IoC container (Pico Container , HiveMind , Sprint IoC, ….) The IoC container resolves the various dependencies amongst different objects. IoC containers allow developers to wire objects via a configuration file, or using code.

There are several ways to inject a dependency into an object

- Constructor injection
- Setter injection
- Interface injection

3.2.4.1 Constructor injection

Constructor injection is all about resolving the dependencies at instantiation time.

```
public class ObjectA {
   private ObjectB objectB;

   public ObjectA(ObjectB objectB) {
      this.objectB = objectB
   }
}
```

Here, we can see that ObjectA cannot be created without resolving its dependency with objectB. An instance of ObjectB needs to be passed along to the constructor of ObjectA.

The actual implementation that will get injected is configured at the IoC level. Keep in mind that objects using IoC don't have a technical dependency with any kind of IoC framework or container. They are plain objects that don't need to implement a particular interface , or extend some kind of framework class.

3.2.4.2 Setter injection

As opposed to constructor injection, where the constructor is used to inject the dependencies , setter injection uses a setter exposed on the object that has a dependency to another object.

In this case, objectB needs to be injected into objectA by using the setObjectB() method.

```
public class ObjectA {
   private ObjectB objectB;
```

```
    public ObjectA() {
    }

    public void setObjectB(ObjectB objectB) {
       this.objectB = objectB
    }
}
```

Again , this is configured at the IoC container level. ObjectA will be instantiated by the framework using the default constructor. Immediately after that , the setter will be called to resolve the dependency with ObjectB.

3.2.4.3 Interface injection

...

3.2.4.4 Service Locator vs. Dependency Injection

Both patterns force a decoupling between the consumer of a particular service, and the underlying implementation of that service.
When using a service locator, the clients asks a reference to a business service via the Service Locator. The Service locator is responsible for constructing the service, and returning an implementation to the consumer.

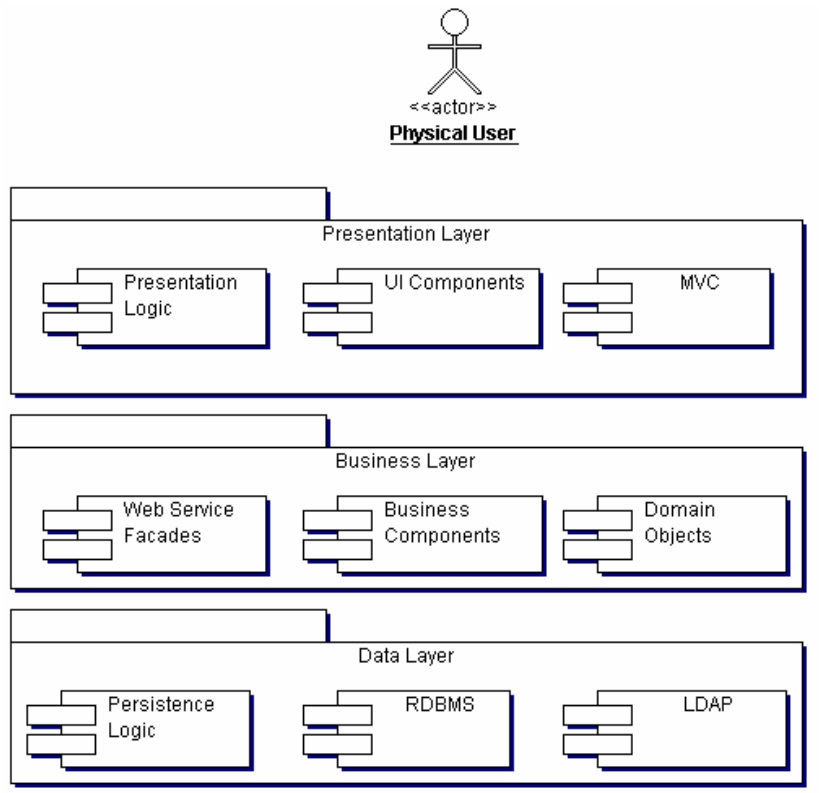
By relying on dependency injection, our service becomes immediately available to the consumer, without relying on a call (and therefore dependency) to the service locator.

3.2.5 Avoid storing all business logic in the service layer.

One should avoid putting to much business logic in the service layer. Good OO design dictates that all behavior should be implemented in the domain layer (see section above).
The service layer should merely orchestrate the interactions between domain objects needed to execute a use case. It should make use of the encapsulated business logic that the domain objects expose, rather then provide its proper implementation for those business rules
Too much business logic in the service layer most likely means that the domain objects primarily contain data while having little or no behavior (anemic data model)

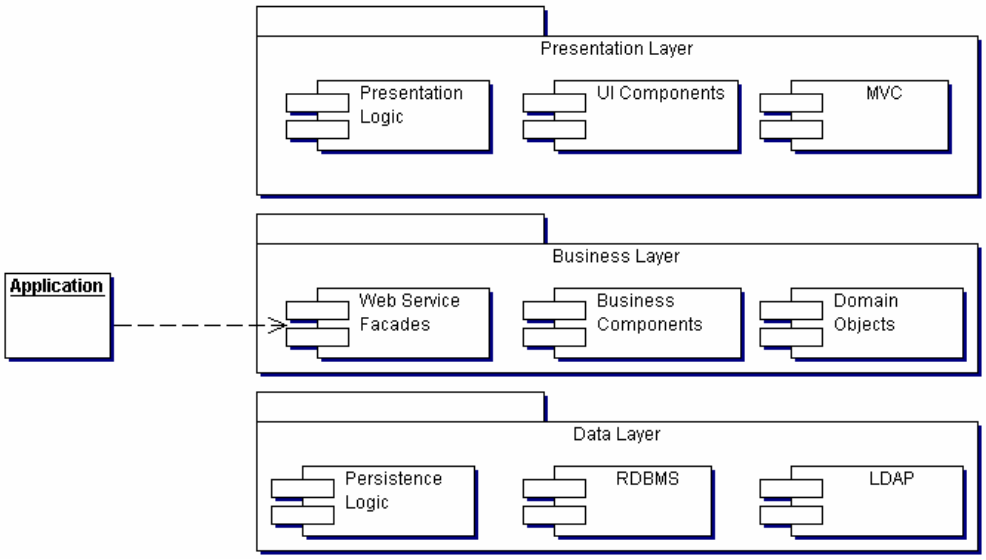
3.2.6 Use the service layer as an alternative entry point

Another advantage of the service layer is the fact that it provides an alternative entry point into the application Before, applications were considered very user centric (use cases being executed by end user in front of a screen), whereas nowadays, not only users, but also other applications want to access the business logic that was implemented.



user centric view of an application

Without a clear service layer, it's very difficult to expose the business logic of an application to external applications. In a service oriented architecture, where everything evolves around services, it's vital that applications also have a service oriented view of themselves. This means that the user interface in the presentation layer won't necessarily be the only thing interested in our business logic. Other applications will also want to make use of the services that our application exposes.



Service oriented view of the application

3.2.7 Integration with the PersistenceLayer

The service layer is the ideal location for interaction with the PersistenceLayer (see later on). As this later is basically responsible for orchestrating so called service logic, it can hook the domain objects into the persistence layer in order to provide support for transparent persistency).

3.2.8 Handle transactions in the Service Layer

The service layer is the ideal location for handling transactions. J2EE containers provide declarative transaction support, and a stateless session bean running in the container (acting as a service façade) can have clear transactional boundaries.

3.2.9 Handling security in the Service Layer

The service layer is also well suited to handle security issues
!!

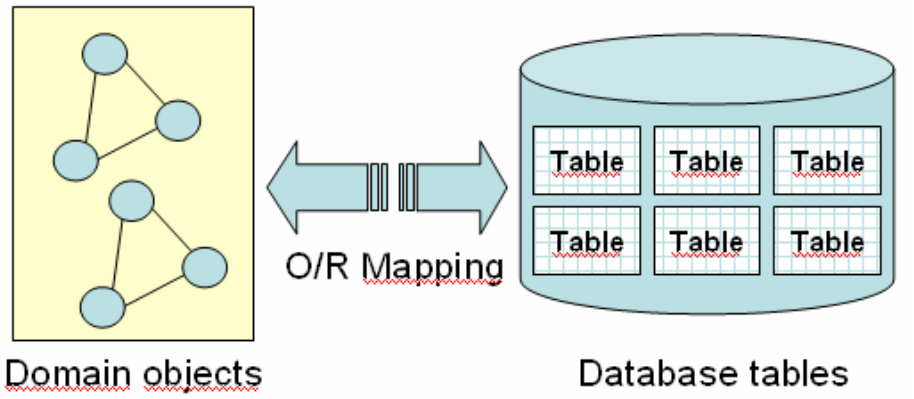
4 Persistence Layer

4.1 Problem

Most applications require some kind of access to a data store (typically, a relational database). This adds an additional level of complexity to the application, as it requires the developer to interact with the data store each time objects need to be stored or retrieved from the data store. These operations are typically performed on the domain objects, so you'll often see SQL statements embedded in the application code. This introduces a coupling between the SQL statements in the application code, and the underlying schema of the database.
Performance, and data integrity through transactions (ACID) is vital when it comes to persisting application data to the data store. In order to alleviate the developer from interacting with the data store directly, we need to have a transparent way to persist domain objects into a data store, allowing the developer to focus on implementing the actual business logic of the application.

4.2 Solution

Introducing a layer solely responsible for moving data between the data store and the application objects alleviates the developer from having intimate knowledge of the underlying data store. The domain layer shouldn't be aware of the fact how domain objects get persisted.
The persistence layer is typically called from the service layer in order to store domain objects from the persistent storage and also to retrieve them and uses some form of O/R mapping in order to map domain objects to database tables.



Mapping between domain objects and database tables

A robust persistence layer should have the following key elements:

- Support different types of underlying data stores.
  The persistence layer shouldn't be limited to one specific relational databases but should support several vendors.
  It should also support other data stores such as flat files, object databases,…..
- Offer transparent persistency
  Domain objects shouldn't contain any persistence code, and shouldn't even be aware of the fact that they can be persisted.
  The persistence layer is responsible for storing/fetching domain objects to/from the data store.
- Object/Relational mapping
  The framework should include some kind of O/R mapping tool. Sometimes, complex mappings need to be applied in order to store a domain object (graph) into the persistent storage. OO concepts such as inheritance don't map all that well into database structures. Therefore, a good O/R mapping tool is mandatory to provide this translation.
- Support Transactions
  The persistence layer needs to support transactions.
- Support Object Identifiers
  The persistence layer should support the concept of an object identifier (the OO equivalent of a primary key in a database) to uniquely identify an object.

4.3 Best practices

4.3.1 Understand the difference between the object and the RDBMS Model

- Understanding the difference between an Object Oriented Model, and a Data Model is vital when implementing a persistence strategy for your application.
- A key difference is the way these 2 models are created. The techniques and tools used are quite different. The fact that Data Models are usually the domain of DBA's , whereas Object Models are usually done by application developers/architects doesn't help to bridge this gap.
- The key thing to remember here is that we shouldn't confuse OO modeling with Data Modeling. There is a misconception that reuse can be achieved by mimicking the data model in the object model.

4.3.2 OO Model != existing RDBMS model

- It's generally a bad idea to base your OO model on an existing data model.
- Chances are that the existing data model is far from perfect. Although it can't hurt to have a look at the underlying data model, in order to get a feel of the technical issues you might encounter later on, it shouldn't impose any restrictions on the design of the object model.

- Even if you have full control over the data model , there are still fundamental differences in the way objects should be modeled , and how they should be stored in the database.

4.3.3 Different object models can map to the same data model

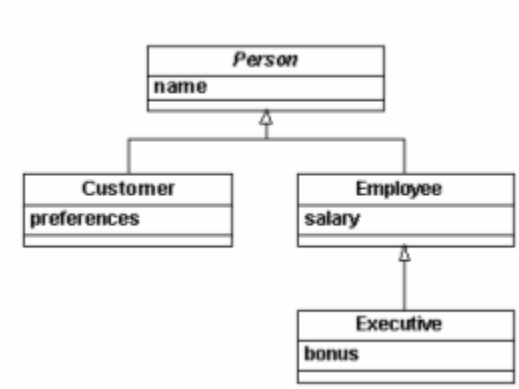
- The mere fact that different object models can map to the same data model should give you the flexibility to implement the object model to the best of your abilities, regardless of the underlying data model.
- Likewise, the persistence layer/framework that has been chosen shouldn't put any constraints on how the domain layer is implemented. Domain objects should be modeled with the full power of OO, regardless of how these domain objects will eventually get persisted.

4.3.4 Leverage O/R Mapping tools

4.3.5 Use Object Inheritance Mapping where possible

One of the key features of OO is inheritance. However, in the relational database world, there isn't a notion of inheritance since database tables are typically flat structures.

As an example, we'll use the following inheritance hierarchy



Simple inheritance hierarchy

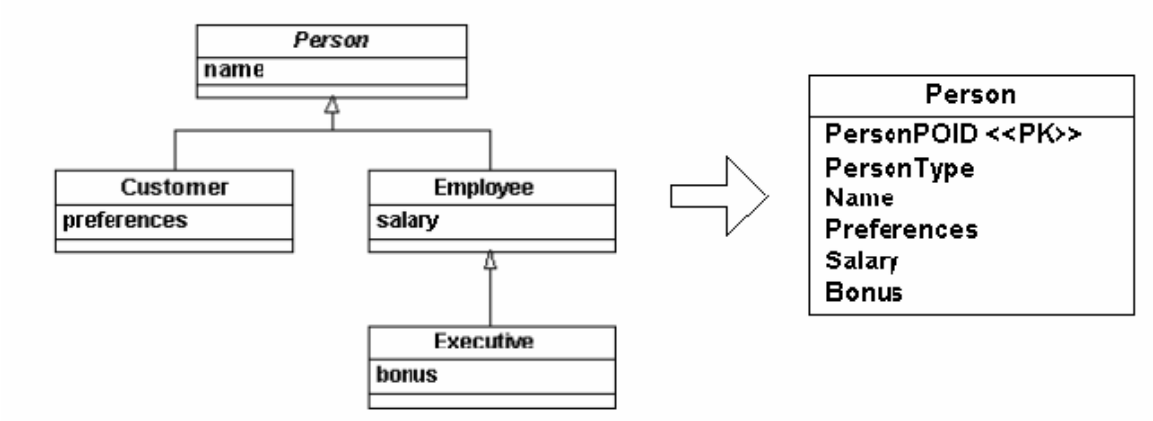
There are 3 major design patterns available to help us map an object inheritance model, into an underlying data model.

- The entire class hierarchy is mapped to a single table
- Each concrete class is mapped to its own table
- Each class is mapped to its own table

Most persistence frameworks should support these 3 patterns.

4.3.5.1 The entire class hierarchy is mapped to a single table

Using this pattern, the combined attributes of all the classes are stored into a single table.

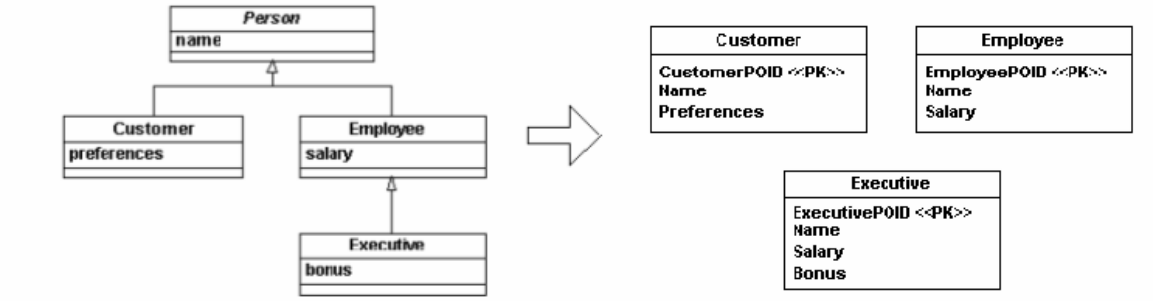


Mapping to a single table

A type column is added to the table in order to differentiate between the different concrete entities (Customer, Employee and Executive). That way , the persistence layer can reproduce the correct object for any given row.

4.3.5.2 Each concrete class is mapped to its own table

Here, each concrete object is represented by its own table. Each table has the corresponding objects attributes (as well as those attributes defined through inheritance). Every table has it's own primary key.

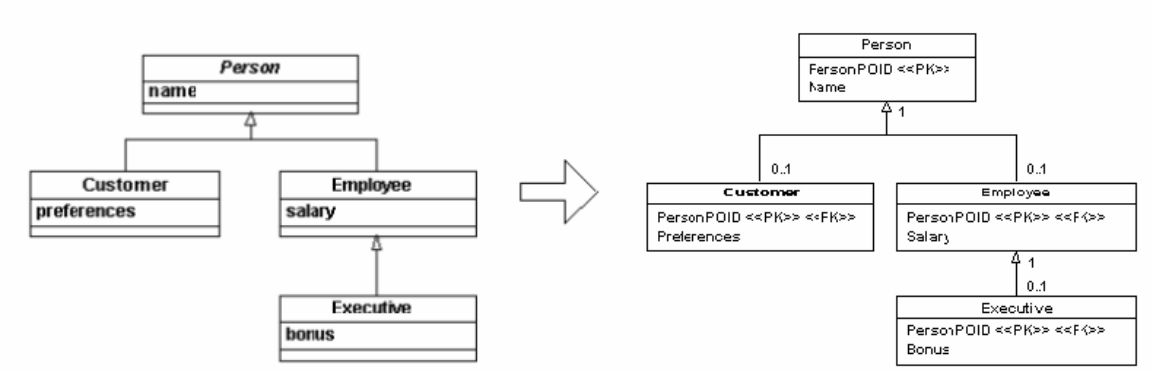


Mapping to separate tables

4.3.5.3 Each class is mapped to its own table

In this case, all classes (including the abstract classes) are stored in their own tables. Every table has its own primary key. The tables that represent concrete

classes (Customer, Employee and Executive) have a primary key that also acts as a foreign key to the Person table.



Mapping to separate tables

4.3.6 Object relationship Mapping

Discuss 1-1 , 1-n , m-n strategies
….

4.3.7 Buy vs build

Building a persistence framework is a difficult and complex task. It's not advisable to roll out your own persistence framework, as several open-source and commercial persistence frameworks are available. Developing a persistence framework in-house would soon become very complex and potentially unmanageable in the future.

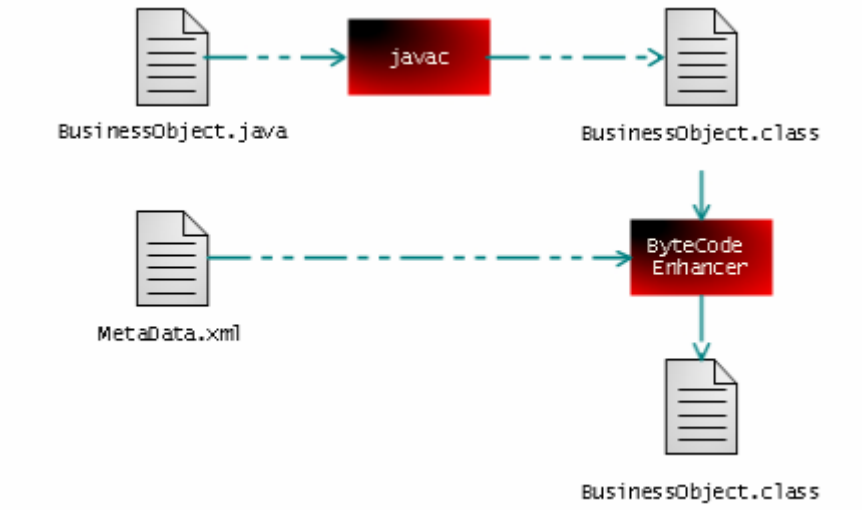
4.4 Persistence Frameworks

Several persistence frameworks are available, both in the .NET, and in the Java/J2EE realm. The primary persistence frameworks in use at the time of writing are

- Java/J2EE
  - JDO
  - Hibernate
  - Enterprise Java Beans 2.x (Entity Beans)
  - Enterprise Java Beans 3
- Microsoft .NET
  - ADO.NET
  - Object Spaces

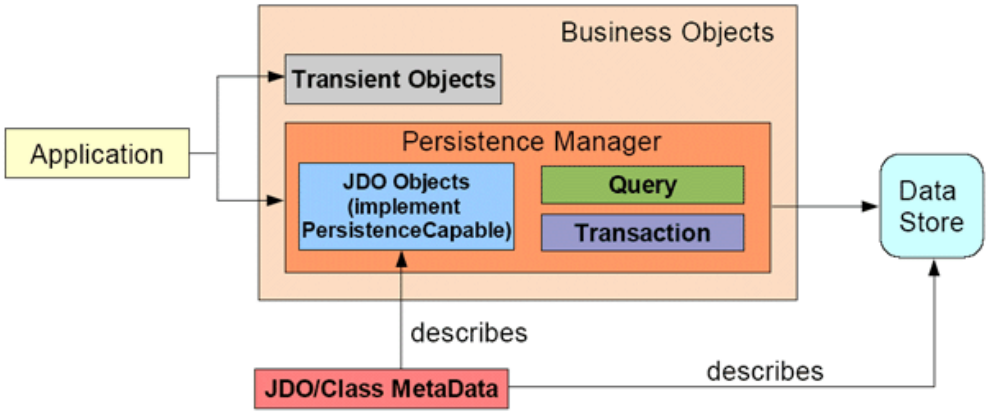
4.4.1 Java Data Objects (JDO)

JDO is an implementation of the Domain Store pattern and is defined by the JCP as JSR-12. JDO uses a byte-code enhancer to make a domain object PersistenceCapable. Objects that are PersistenceCapable can take advantage of the JDO PersistenceManager to achieve transparent persistency.



JDO Class Enhancement Process

The JDO PersistenceManager is responsible for storing domain objects in the data store, and retrieving them. It uses a JDO Meta Data file that describes how JDO objects (objects implementing the PersistenceCapable interface) should be persisted in the data store.



JDO Overal Architecture

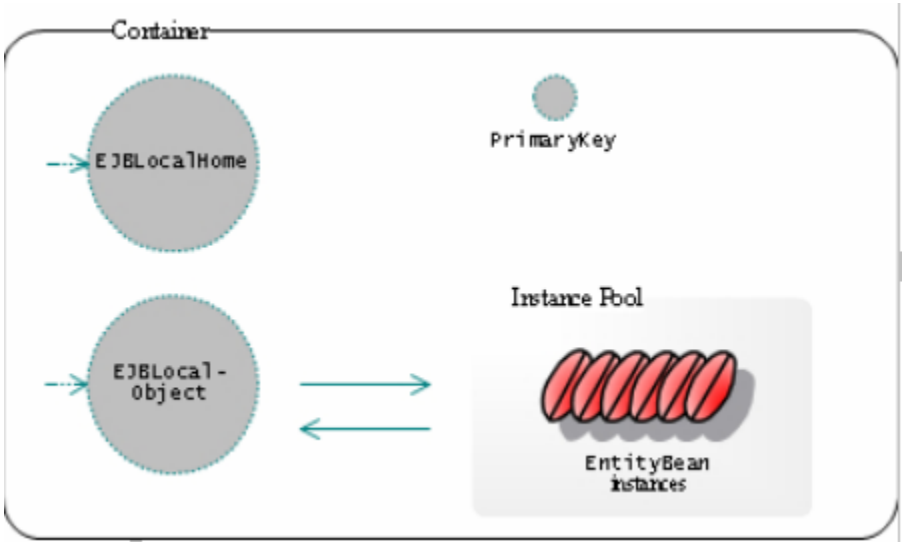
4.4.2 Enterprise Java Beans 2.x

The EJB specification defines entity beans as the standard way to persist business object. Entity bean must adhere to a strict contract.

Some entity bean characteristics :

- An EJBLocalHome interface providing methods to create the actual entity bean.
- An EJBLocalObject will be returned that contains the actual business methods of the entity bean.
- The PrimaryKey class associated with the entity bean is the object representation of the primary key of the entity on the data store level.

Entity Beans run inside an EJB container that handles lifecycle operations, transaction / security management…



The downside of handling persistency through the EJB2 specification is that business objects need to follow a strict interface. Business objects implemented as Entity Beans aren't plain old java objects, and can only run inside the EJB container. The EJB 2 specification doesn't allow us to use the full power of OO when it comes to modeling our business objects (no support for inheritance).
We would advise against the use of entity beans as business objects, and would opt for a more transparent persistence mechanism (see JDO, Hibernate & EJB3 specification).

4.4.3 Enterprise Java Beans 3

The new EJB3 spec aims to simplify the entity bean programming model in order to provide transparent persistency. It does this by supporting a lightweight domain model (plain old java objects), including inheritance and polymorphism. It doesn't force the developer to extend EJB specific classes/interfaces, and gives the developer more freedom in terms of modeling the business objects.

The Java Persistence API (used by the EJB3 specification) manages the persistence, and object/relational mapping with J2EE and J2SE.

The EJB3 specification is still in early draft, and therefore not suitable for production use. Currently, very little vendors provide an EJB3 container, or support for the various EJB3 APIs. However, in the future the EJB3 specification (and also the Java Persistence API) will become the standard way of handling persistency in a J2EE or J2SE environment.

4.4.4 Hibernate

Hibernate is a popular open source persistence framework for the java community. It supports transparent persistency of business objects (implemented as plain old java objects). It has many similarities with JDO but instead of using byte-code Enhancement uses runtime reflection to determine the persistent fields of an object, and its relationship with other objects. The persisted objects are described in a mapping file (similar to the JDO mapping file).

Hibernate has a large user base and is probably one of the most popular persistence frameworks out there. Currently, Hibernate isn't based on a particular standard, but it is planning to incorporate the EJB3 APIs with regards to persistence.

4.4.5 Other J2EE/J2SE Persistence frameworks

….

4.4.6 ADO.NET

ADO.NET isn't really a persistence framework, but rather a relatively low level API (comparable to JDBC in the J2EE world) for handling access to various data-stores. It doesn't really offer transparent persistence like the previous persistence frameworks but it is the standard way in .NET to communicate with data stores.

.NET ObjectSpaces

ObjectSpaces is an object/relational framework for the .NET environment that provides an abstraction layer between business objects, and their underlying storage in a data store. It runs on top of ADO.NET (the standard API for database related access) and attempts to introduce the concept of transparent persistency into the .NET environment

ObjectSpaces is currently still in BETA, and therefore not really suitable for production code, leaving the choice of a transparent persistence framework in .NET somewhat limited. ObjectSpaces is going to fill a huge gap in the .NET environment

4.4.7 Other .NET Persistence frameworks

EntityBroker
http://www.thona-consulting.com/content/products/entitybroker.aspx

LLBLGen Pro

http://www.llblgen.com/

Norpheme (open-source)
http://www.norpheme.com/

Sisyphus (open-source)
http://sisyphuspf.sourceforge.net/home.htm

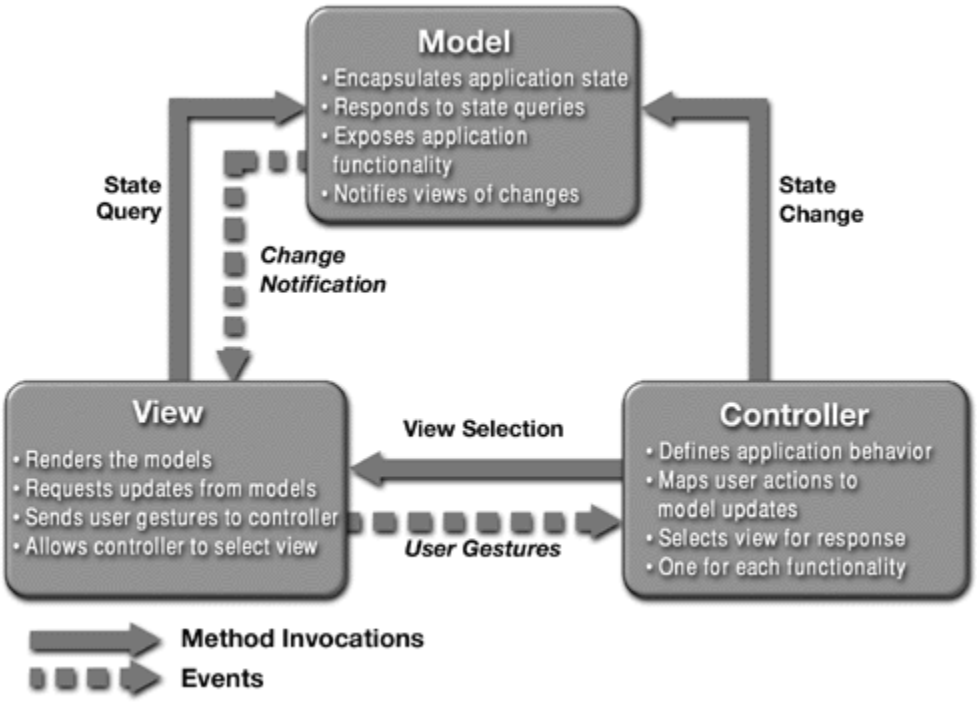
5 Presentation Layer

5.1 Introduction

Most applications expose some kind of user interface to the end user. This user interface can be a web site (thin client, running remotely on a web server), or an application client (fat client, running locally).
The user interface is responsible for capturing user actions, and giving feedback to the user. We want to clearly separate the user interface code of the application from the rest of the application, so that changing the front-end doesn't require us to rewrite the rest of the application.

5.2 Best practices

Introduce a presentation tier in your application that is solely responsible for the user interface aspects.
Most applications today use some kind of Model View Controller (MVC) implementation on the presentation layer.



Notice how the model 'exposes application functionality'. This doesn't mean that the model should contain actual core business logic, but should merely expose this

functionality. The Model is still part of the presentation layer, and should call the appropriate business delegates to access the business tier of the application.

5.3 Apply presentation tier design patterns

Some common design patterns implemented on the presentation layer include:

5.3.1.1 Intercepting filter

This pattern applies to request pre- and post-processing. It applies additional services needed to process a request. Authorization logic could be centralized in such a filter. This is typically implemented as a servlet filter.

5.3.1.2 View helper

A view helper encapsulates the presentation and data access logic portions of a view, thus refining the view and keeping it simpler. Often, these are implemented as JSP tags and JavaBeans.

5.3.1.3 Composite view

This pattern makes view presentation more manageable by creating a template to handle common page elements for a view. Often, Web pages contain a combination of dynamic content and static elements, such as a header, footer, logo, background, and so forth. The dynamic portion is particular to a page, but the static elements are the same on every page. The composite view template captures the common features.

5.3.1.4 Front controller

This pattern provides a centralized controller for managing requests. A front controller receives all incoming client requests, forwards each request to an appropriate request handler, and presents an appropriate response to the client. Most presentation tier frameworks already implements the notion of a front controller, and allows you to extend this part of the framework.

5.3.1.5 Value object

This pattern (sometimes referred to as a Transfer Object) allows us to move data between tiers (usually the Presentation and Business tiers)
In one call, a single value object (or Transfer Object) can be used to retrieve a set of related data, which is then returned to the client.
Most applications will have many value objects representing business entities (or a composite structure, comprising of several business entities).

5.3.1.6 Business delegate

This pattern decouples the intervenes between a remote business object and its client, adapting the business object's interface to a friendlier interface for the client.

It decouples the Web tier presentation logic from the EJB tier by providing a facade or proxy to the EJB tier services. The delegate takes care of lower-level details, such as looking up remote objects and handling remote exceptions, and may perform performance optimizations, such as caching data retrieved from remote objects to reduce the number of remote calls.

5.4 Best Practices

5.4.1 Build upon an existing presentation framework

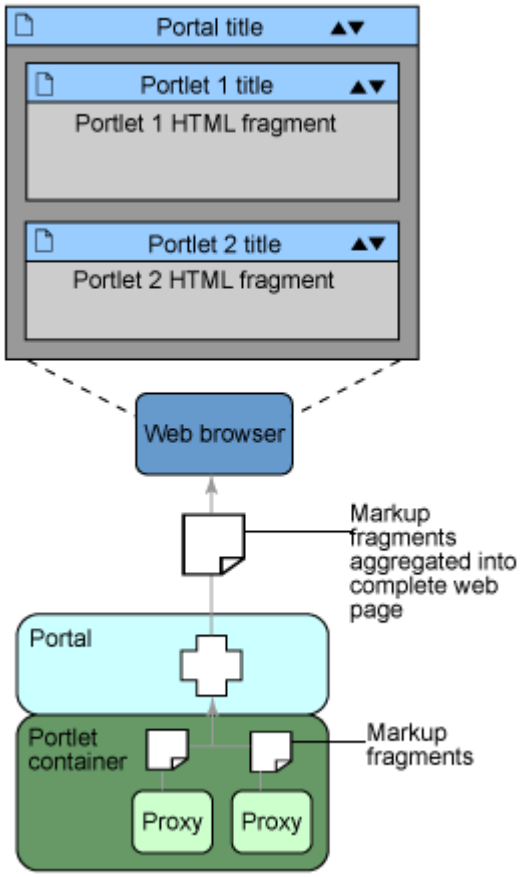
Building upon an existing framework will make sure that these best practices will be used, and will improve the applications time to market.

5.4.2 Choose a corporate standard

- Although every application has its own characteristics, most presentation frameworks are generic enough to fulfill most of the requirements.
- Therefore, it's important to have some kind of corporate standard when it comes to the presentation framework. (Unless the application has very specific needs, an alternative framework might be chosen to implement the solution; however, this should be rather exceptional)
- Due to the large number of presentation frameworks out there, a clear vision is needed when choosing to build upon such a framework.
- See the section below that lists the most popular presentation frameworks in the java realm, and the guidelines that can help assist you in choosing the right presentation framework for your specific needs.

5.4.3 Develop with portal integration in mind

- A lot of companies have jumped on the Portal bandwagon in order to provide a uniform view of their underlying applications.
- Portals can provide a single point of entry for employees, partners & customers within an organization.
- A portal allows for different enterprise applications to be plugged in, using a variety of integration patterns (Portlet JSR-168 specification, URL rewriting , ….)
- A portal basically aggregates different markup fragments (created by the portlets running in the portlet container) into a single webpage that is presented to the user. A single portal page can contain many portlets.

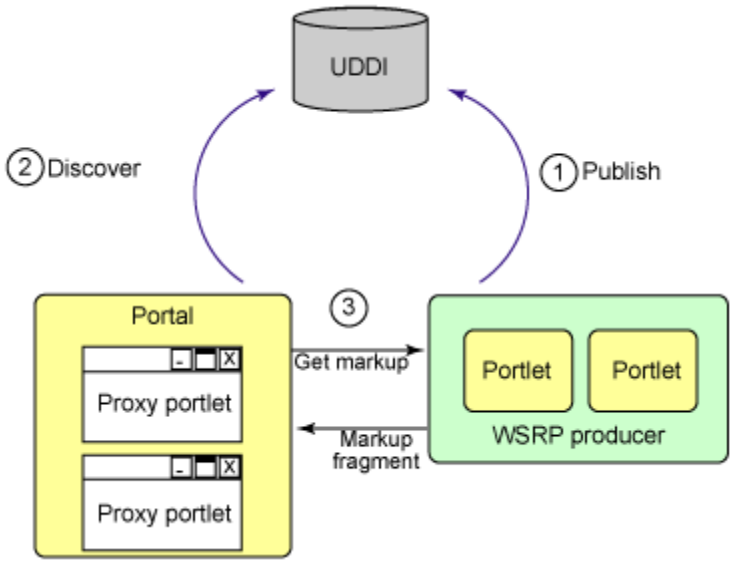


Constructing a Portal page

Portlet development has been standardized using the JSR-168 specification by Sun. JSR-168 provides a programming model to develop & deploy portlets in a portlet container. The portlet container is responsible for hosting the portlets.

5.4.4 Have a strategic vision to integrate external applications into the corporate portal.

The **OASIS WSRP** specification simplifies the integration of external applications or services into portals by using web services.
Without programming effort, a portal administrator can select various 'services' (remote portlets), and plug them into the portal without any programming effort.



WSRP

In the image above, we can see that

1. A WSRP producer has published a couple of services (portlets) into a UDDI registry.
2. A WSRP consumer has the ability to discover those services through the UDDI registry
3. The administrator for the WSRP consumer (typically also a portal server) can request the markup that is generated by the remote portlet, and aggregate that markup on the local portal.

Portal technology is gaining a lot of momentum over the past couple of years, and practically all major vendors offer some kind of portal product.

5.4.5 Add existing applications to the corporate Portal

At some point in time, the need will arise to integrate existing applications into the company portal. As stated earlier, there are several integration patterns available when it comes to adding applications to a portal.

5.4.6 ASP.NET

ASP.NET is the presentation tier programming model for the .NET platform. It offers similar features as most java based presentation frameworks.
Much like JSF, ASP.NET also exposes a rich component model that the developer can use.

5.5 Presentation Frameworks

The usage of frameworks on the presentation layer provides developers with a quick & easy way to construct user interfaces for the java platform. Many of the best practices & presentation tier design patterns are encapsulated in these so called presentation frameworks. These frameworks (primarily open-source) allow a development team to concentrate on the business logic, instead of worrying about the lower level plumbing code when it comes to the presentation tier.

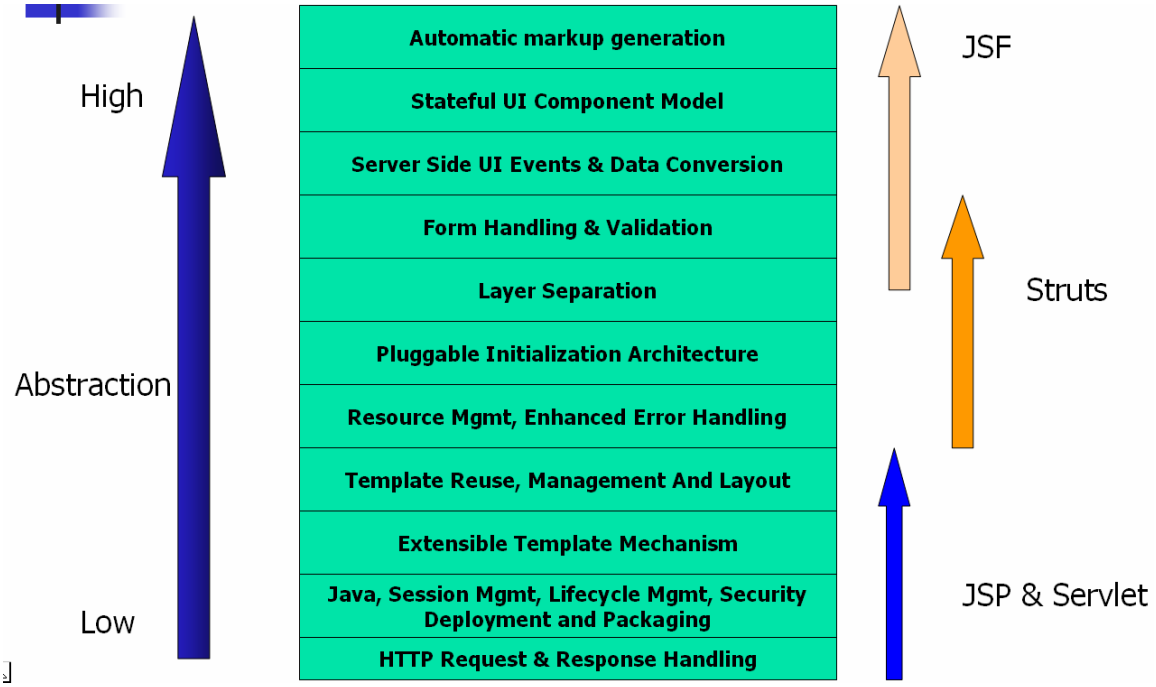
Java developers have a large choice when it comes to presentation frameworks. Some popular frameworks include

- Java Server Faces
- Struts
- Spring Webflow
- WebWork
- Tapestry

The .NET framework uses ASP.NET as its presentation framework. No third party frameworks are available for .NET.

5.5.1 Evolution of presentation frameworks in the J2EE space.

Most presentation frameworks available today (either commercially or via open source) are based on the traditional Servlet/JSP spec.
During recent years, developers have tried to shield us away from the more lower level plumbing code associated with the Servlet/JSP specification.
These days, nobody would build a large scale web application without using some kind of presentation framework that sits on top of the Servlet/JSP spec.



Evolution of presentation frameworks in j2EE

5.5.2 Java Server Faces (JSF)

A presentation framework that's currently receiving a lot of momentum is Java Server Faces (JSF). JSF is gaining a lot of momentum due to the fact that its standards based. Java Server Faces is defined by JSR-127 where it is described as a UI Framework for Java web applications. JSF shares many concepts with other web frameworks such as Struts or Tapestry, but it adds a powerful and stateful UI component model.

JSF promotes the notion of reusable web components and defines a request processing lifecycle where every aspect of the

Some of the characteristics of JSF:

- Protocol and markup independent
- Managing UI component state across requests
- Support form processing
- Provide data validation
- Provide a strong event model
- Provide type conversion
- Provide error and exception handling
- Handle page-to-page navigation

JSF has a steeper learning curve then Struts, and it's easier to find developers who have substantial experience with Struts development, however, we need to take into account that JSF is a Sun specification, and we've already seen that vendor support for JSF is substantial. Large industry players including Oracle, IBM and BEA have already embraced this technology, and are offering third party tools that will ease the development of JSF based web applications.

We do believe that JSF is a mature technology, and that it can offer many advantages over other presentation frameworks. The fact that its born out of a Sun specification, and that vendors have already embraced the technology will undoubtedly make JSF a success.

We recommend looking into this technology, especially for new projects. There's no need to rewrite all your existing Struts applications using JSF as Struts will remain among us for quite some time.

Pros

- J2EE Standard
- Vendor support
- Rich UI component model

5.5.2.1 Cons

- Relatively new technology
- Steeper learning curve
- Not many experienced JSF developers available.

5.5.3 Apache Struts

Another popular web framework in the j2EE world is Struts. Struts is a presentation framework that is based on the MVC Pattern. Its primary functions are:

- Handle request/reply
- Provide a controller
- Exception handling
- Modeling value objects for the presentation layer
- Form validation

Some bad practices when using Struts:

- Implementing application business logic into Struts actions
- Communicating with the data store from Struts Actions
- Handling Transactions in a Struts Action
- Handling application security in a Struts Action / JSP

Pros

- Still the defacto standard for developing web applications
- Lots of developers with Struts experience
- Big community

5.5.3.1 Cons

- Not based on a standard.
- Future of struts is still unclear (as opposed to JSF)
- Difficult to unit test

5.5.4 Spring Web MVC

Spring's web MVC framework is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as support for upload files. The default handler is a very simple Controller interface, just offering a ModelAndView handleRequest(request,response) method

Pros

- Still the defacto standard for developing web applications
- Lots of developers with Struts experience
- Big community

5.5.4.1 Cons

- Not based on a standard.
- Future of struts is still unclear (as opposed to JSF)
- Difficult to unit test

5.5.5 WebWork

WebWork is a Java web-application development framework. It is built specifically with developer productivity and code simplicity in mind, providing robust support for building reusable UI templates, such as form controls, UI themes, internationalization, dynamic form parameter mapping to JavaBeans, robust client and server side validation, and much more.

5.5.5.1 Pros

- Simple architecture
- easy to extend
- Tag Library is easy to customize - backed by Velocity
- Interceptors are pretty slick

5.5.5.2 Cons

- Small Community

- Poor Documentation
- Client-side validation immature

6 Overall design considerations

6.1 Introduction

- Minimize network roundtrips
- Centralize business logic
- Moving data between tiers
- Transaction handling
- Exception handling
- Managing State
- Managing the code base

6.2 Best practices

6.2.1 Minimize Network roundtrips

- In any multi layered architecture where objects can be distributed across the network, we need to be aware of the fact that network calls are expensive.
- Too many fine grained method calls will have a performance penalty.
- An important design goal should be to minimize network roundtrips.
- This is achieved by exposing a coarse grained business interface to the clients, so that they are able to fetch their data with the least possible method calls.

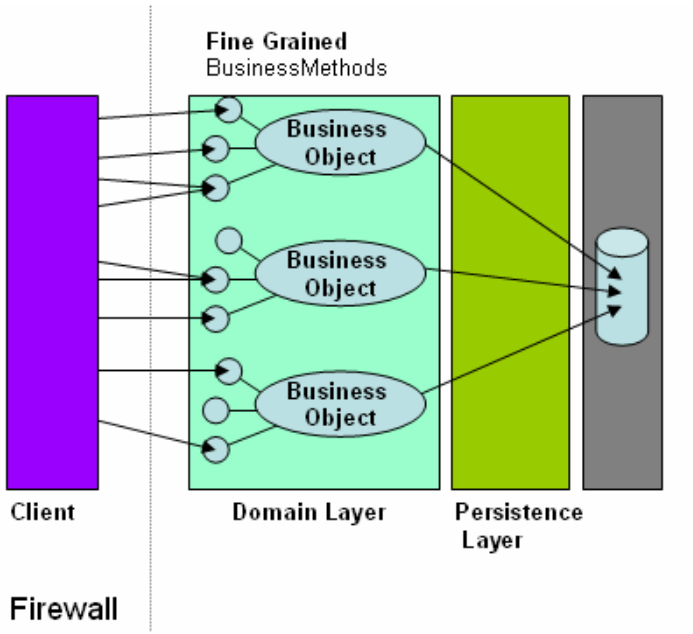
6.2.2 Design patterns

6.2.2.1 Session façade

The service layer that we described in the previous section lends itself perfectly for offering these course grained business interface. The Service façade pattern is based on the Façade (GoF) pattern

In the picture below, we see an architecture that doesn't have a clear service layer. This means that

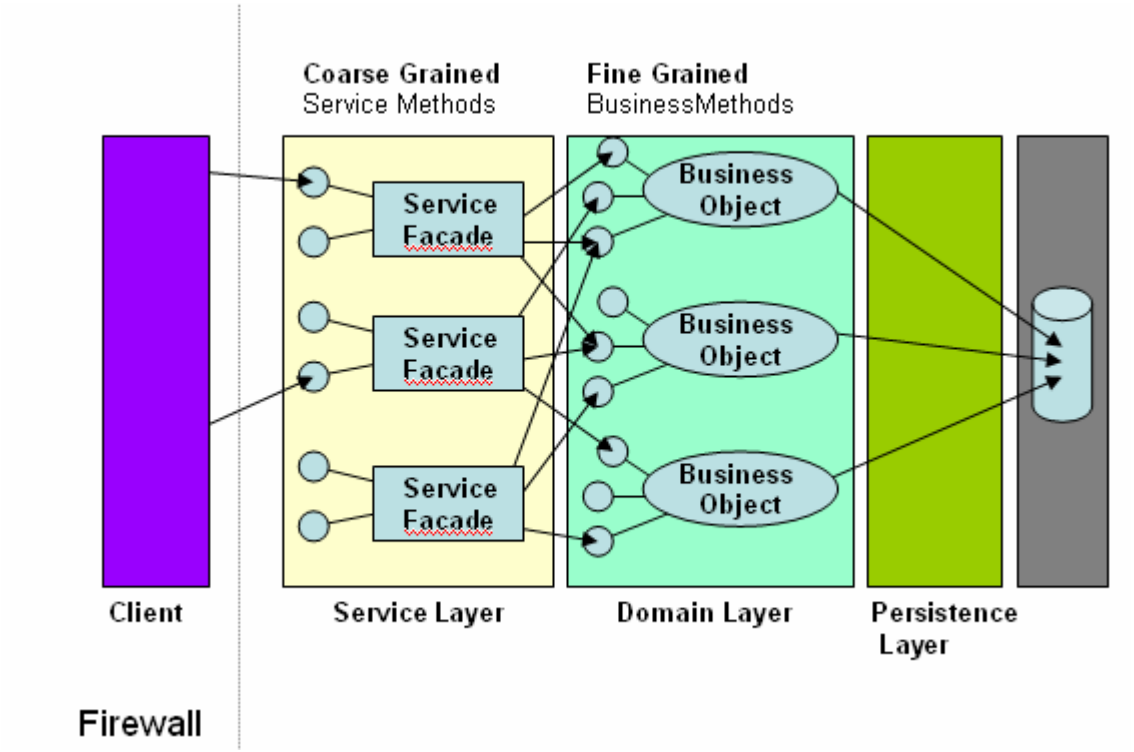
- A lot of fine grained methods need to be called in order to execute a use-case, resulting in a negative performance impact.
- The orchestration of the business processes will be scattered throughout the architecture (the client will need to perform most of the orchestration).
- Interactions between business object can be very complex, the client shouldn't be aware of those interactions, as its primary interest is service logic, as opposed to pure domain logic.



Architecture omitting the service layer

Below, we see an architecture containing a service layer containing several service facades. The facades are responsible for offering course grained business operations to the client. This will result in

- Less network roundtrips, as the client is now talking directly to the service layer. The client can use the course grained business operations, and doesn't need to worry about how these operations are implemented.
- Separation of concerns. The Service Layer is the place to orchestrate the business flows and should make use of the domain logic implemented in the domain layer.
- The client should only talk to the service facades. (via a business delegate)



Architecture including the service layer

During the design of the session facades, we have several options available to us

- Create a single session façade containing all use cases
- Create a session façade for each use case
- Group together related use-cases into a session façade, and allow for multiple session facades within the service layer

We should strive to group together related use-cases into a single session façade. For example, a banking application might have an AccountManager façade responsible of creating/retrieving accounts. A CreditCheckManager façade could be responsible for performing credit check operations.

Session faces are all about reducing complexity, and not merely shifting it. The session façade should be more then a simple wrapper, but should provide an abstraction useful for the client.

6.2.3 Centralize Business Logic

In terms of implementing business logic, you should always keep the following in mind

- Domain objects contain all pure application business logic. All business logic should be in the domain layer.
- Service layer should be a thin wrapper, orchestrating business flows by manipulating the domain objects, and optionally delegating to a persistence layer.

6.2.4 Moving between tiers

- In any distributed architecture, data will need to be moved back and forth across the various layers (possibly even passing through firewalls).
- For that, we need a lightweight format (XML), and a protocol like HTTP.
- Typically, data will be moved across tiers using transfer objects.

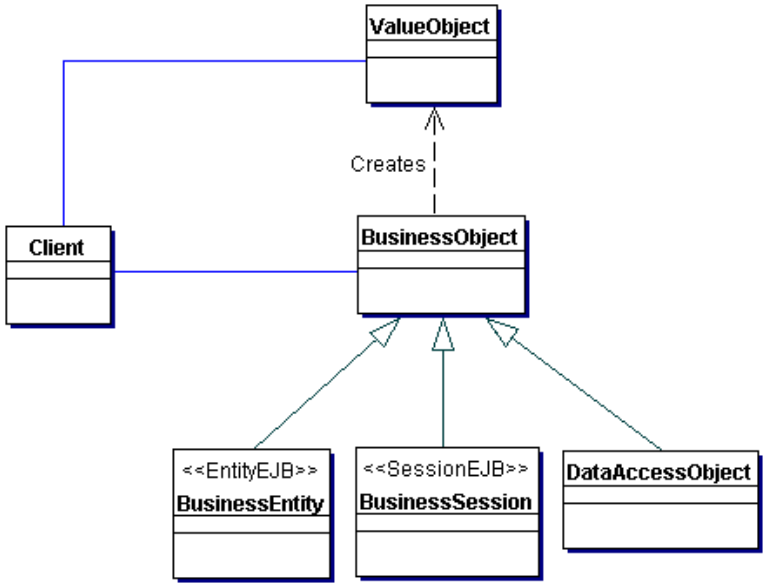
6.2.5 Design Patterns

6.2.5.1 Transfer Object

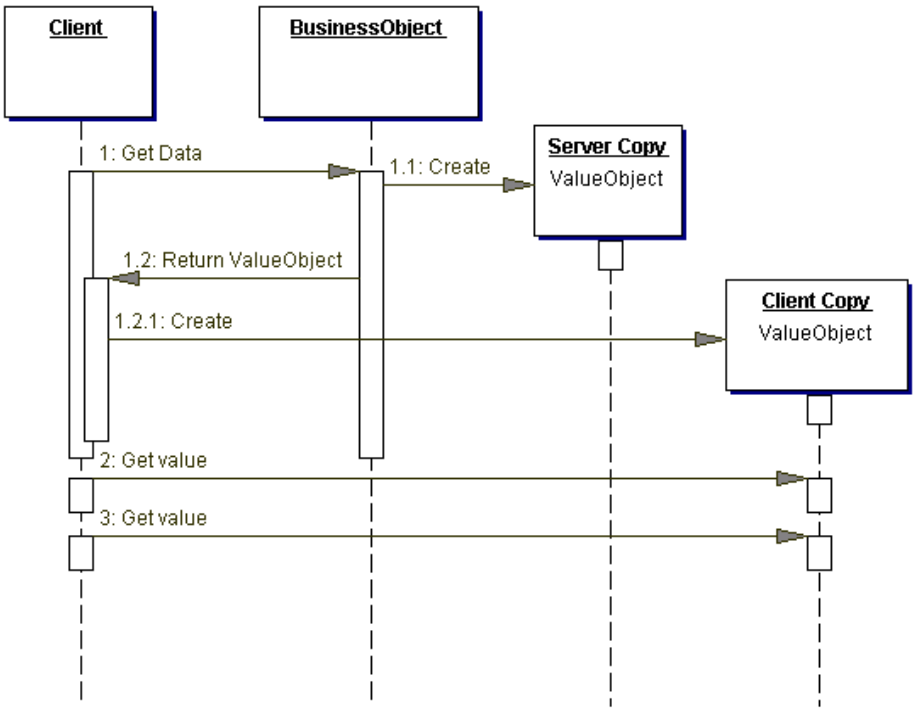
Transfer objects are simple POJOs that are used to carry data across the different tiers. The advantages of transfer objects are twofold:

- Reduces coupling between the presentation layer and the business layer
- Reduces the number of (potential) remote method calls

In order to reduce the coupling between the presentation layer, and the business components residing in the business layer, we want to avoid working with these business objects directly on the presentation layer. Therefore, the concept of a transfer object is introduced to reduce this coupling.
We already saw that the service layer offers course grained business methods to its clients. In order to return data from those business methods to the client, transfer objects are assembled (Transfer Object Assembler pattern) on the service layer, and passed on by value to the client.



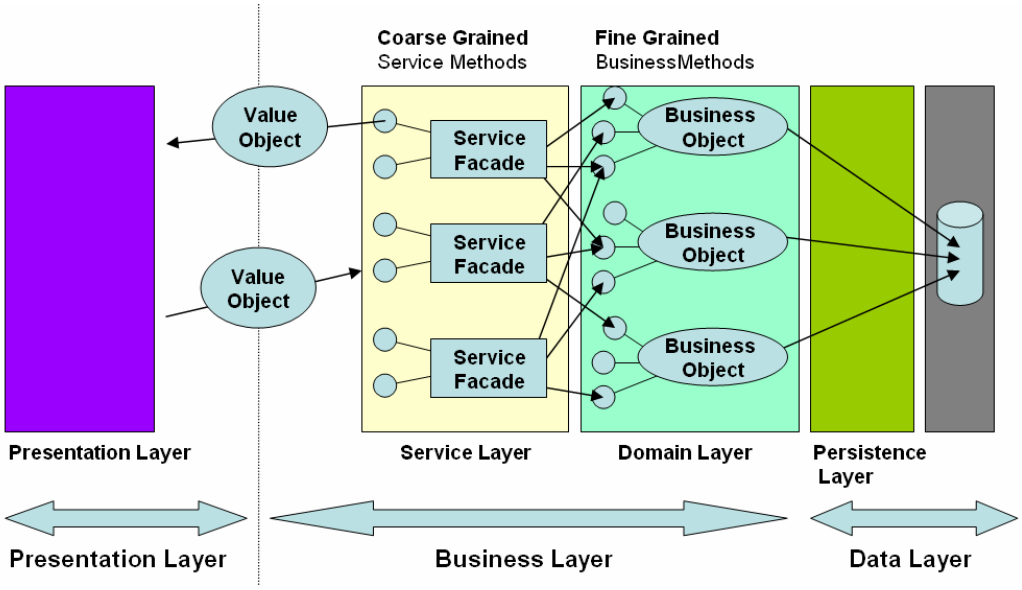
Class diagram of the Transfer Object Pattern



6.2.5.2

Sequence diagram of the transfer object pattern

As you can see, the actual business components never reach the client. The client works with transfer objects that are assembled on the service level.



Moving data from presentation to business tier using value objects

As you can see, both the service layer and the presentation layer share these value objects. This could potentially link the presentation layer to the service layer. For that reason, the service layer is typically split up into 2 parts

- A public API that is made available to the clients
- The actual implementation of the service façade.

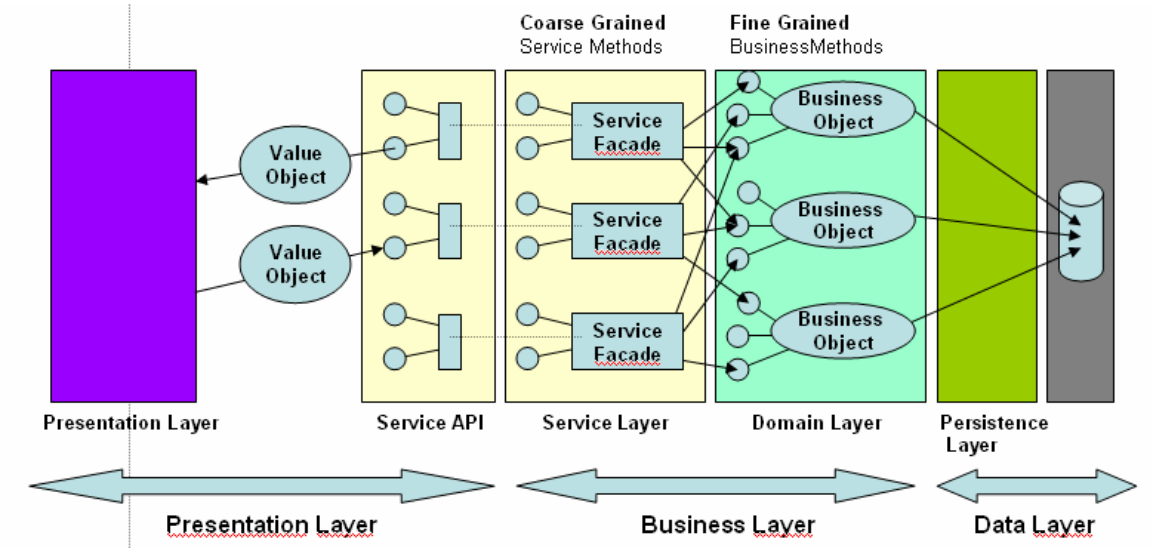
The public API that is made available to the clients contains

- The business interfaces that clients will work with.
- The transfer objects that can be used by clients to communicate with the service layer
- Business delegate factories that can be used by clients to obtain references to services

The actual implementation of the service façade (for example a session beans), and the internal components used by the service layer (domain objects) should never be exposed to the client. This part of the service layer typically includes

- The actual implementation of the session façade.
- The actual implementation of the business service

- The Transfer Object Assemblers necessary to create transfer objects.

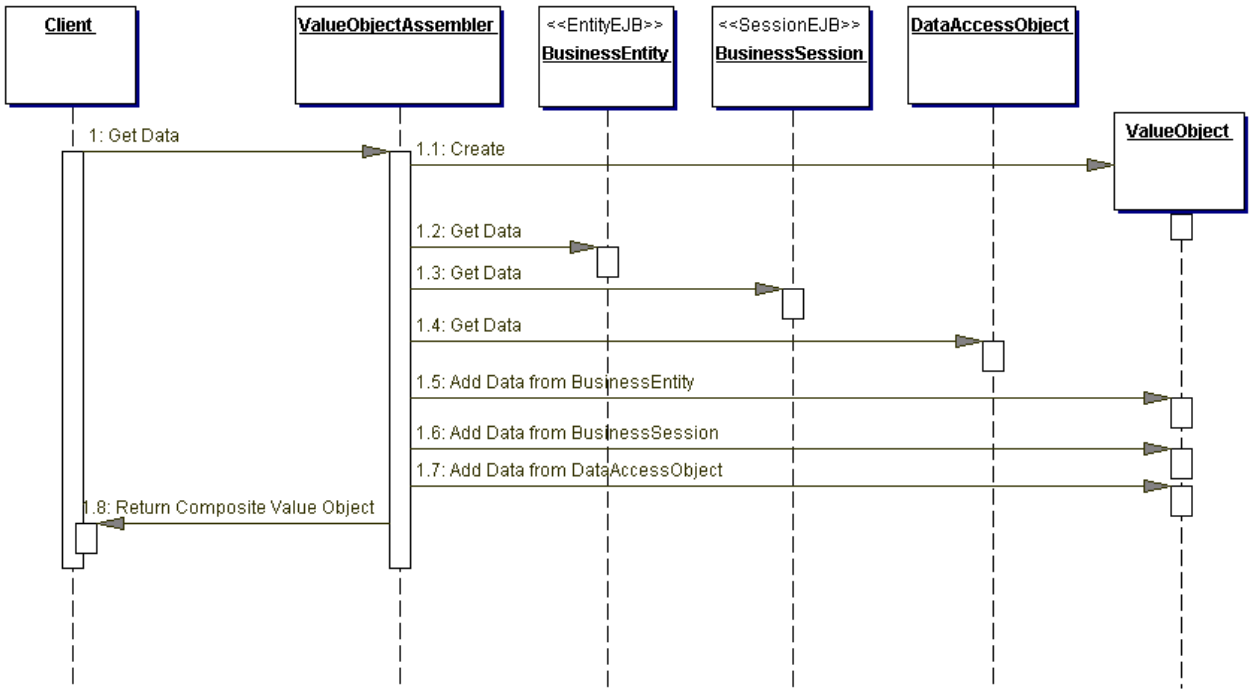


Splitting up the service layer into a public API and private part

As you can see here, the Service API has been made accessible for the presentation layer to use. The presentation layer doesn't have a dependency to the actual implementation of the service layer.

6.2.5.3 Transfer Object Assembler

The transfer object assembler pattern introduces a component that is solely responsible for creating transfer objects. Composite transfer objects can be created by a Transfer Object Assembler, in order to return the best possible model representing the data that is offered by the service.



Transfer object assembler sequence diagram

As you can see from the sequence diagram above, the transfer object assembler creates value objects based on the values residing in the business components. That way, the client can obtain a rich model, representing the data offered by the service, without exposing the internal components that were needed to implement the service.

6.3 Transactions

...

6.4 Exception handling

...

6.5 Managing State

....

7 Managing the code base

7.1 Introduction

Managing the code base of an enterprise application is not an easy task. Although we have clearly defined specific layers within the architecture, we need to ensure that the same layers are also visible within the code base.
Too often, applications are written by creating a single software project where all the code is stored in a single tree. After some time, it becomes unmanageable, and very tricky to see the dependencies between the various layers in the architecture, since from a developers point of view, everything is stored in the same source tree.

7.2 Best practices

7.2.1 Setup sub projects in the development environment

The splitting up of code into several sub projects, or sub components will help you to bring some structure into the development environment. It will allow for different sub modules to be developed independently of the other sub modules.

7.2.2 Standardize the development environment

It will also introduce some level of structure into the project , so that new , and less experienced developers can clearly target what they are intended to do , without losing sight of things in a monolithically source tree. Each sub module can have its own set of documentation, scripts and unit tests. Developers can concentrate working on a single module, without being impacted by other sub modules.

7.2.3 Promote reuse through the sub projects

Sub modules also promote reuse, as developers will clearly see who is using what sub module. Developers will realize that changes made to a sub module will have an impact on all components that have a dependency with that sub module. Also, changes are that you'll see a 'Commons' sub module popping up in the source tree at some point in time, containing all common , and non-application specific routines. These Common modules could potentially be reused by other projects.

7.2.4 Manage dependencies between the sub projects

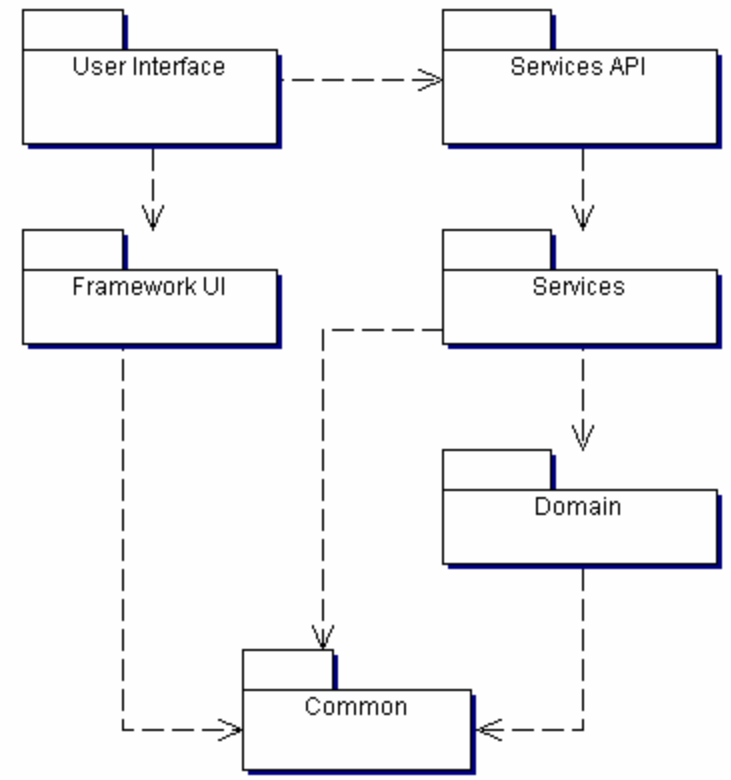
Splitting up the source tree into different sub modules also make it very easy to spot the different dependencies within the software project. For example, the presentation layer should never access domain objects directly.
When all source code is stored in a single source tree, it becomes very difficult to separate these types of things. By introducing a UI project alongside a Domain project, dependencies can be easily managed and enforced by the development environment (for example, by not allowing the UI project to have a dependency on the domain project.) This will make it impossible for developers to use domain objects in the presentation tier.

7.2.5 Apply a Modular approach to the development environment

When something goes wrong, we need to be able to fix the problem as soon as possible. Sometimes, redeploying the entire application just isn't an option. By introducing several sub modules into the development environment, we can quickly pinpoint the exact cause of a problem (by identifying the module), and possibly deploy a patch that deals with that specific issue in the sub module.

In the picture below, we see the main modules that an enterprise application should contain, as well as the dependencies between them.

- **User Interface**
  Code related to the user interface. These include user interface components, presentation framework code, web pages…
- **Framework UI**
  Framework code that's used by the user interface project. These include custom user interface components, helper classes…
- **Services Domain API**
  The public interface of our service layer. Contains the business interfaces that the client is allowed to use, as well as the transfer objects that can be used by the clients.
  The User Interface module has a dependency with the Services API module, as it will use the interfaces to call service methods , and use the value objects to have a representation of the data returned by those services.
- **Services**
  This module contains the actual implementation of our service layer. Contains the actual implementation of the business services, transfer object assemblers, …
  As you can see , the user interface doesn't have a dependency with the Services module, as it uses a strict API to access the business logic. The Services API module has a dependency to the actual services implementation.
- **Domain**
  This module contains the domain objects used by the application. These objects should have very little dependencies. Depicted here is a dependency with a common module, that can contain application exception classes, enumeration types , ….
  As you can see, the only module that is able to access the domain objects is the services module.
- **Common**
  This module contains all reusable artifacts that can be used by all layers. This module typically contains the application exception classes, enumeration types, …..



Modular approach to the source tree

7.2.6 Conclusion

In conclusion, it's safe to say that applying a modular approach to the code base adds many advantages to the overall development process. It will make the overall project more manageable, and can avoid many headaches as the project evolves.