

---


---

# Practical Java Design

## with Struts and JDO

---

---

<i>Version:</i>	1.1
<i>Author:</i>	Amir Firdus
<i>Contact:</i>	<a href="http://www.firdus.com">www.firdus.com</a>
<i>Date:</i>	October 2004
<i>Goal:</i>	To create practical and affordable web application architecture in Java.
<i>What is new:</i>	Version 1.1 is a major clean up. However, all the basic concepts have stayed the same.
<i>Copyright:</i>	© 2004 Amir Firdus (Firdus Software) 

## Table of Contents

Introduction.....	3
General.....	4
Development Entry Point.....	4
Java Landscape.....	5
Architecture.....	5
Solution Check.....	7
Domain.....	8
Domain Type.....	8
Implementation.....	8
Object Link.....	9
Object to Table Mapping.....	10
Object to Object Relationship.....	11
WebEmp Domain.....	12
Version 1.1 Notes.....	14
Application Services.....	15
Persist Service.....	15
Version 1.1 Notes.....	16
Domain Bean.....	17
As a Controller Element.....	17
As a Transaction Boundary.....	17
As a Data Detachment Point.....	18
Implementation .....	18
Web User Interface.....	19
Actions.....	19
Screens .....	20
History Bar .....	20
Custom Tags .....	20
Version 1.1 Notes.....	21
Data Flow.....	22
Conclusion.....	23
Literature.....	24

## INTRODUCTION

Many times we read excellent articles with refreshing ideas. Every so often there is a book that inspires us. From time to time we would like to build a prototype. Soon after we sit in front of our computers we face the same problem, days if not weeks to do anything.

Primary goal of this project is to put together practical and affordable environment for fast web application development.

Project consist of:

- Practical Java Design with Struts and JDO book
- WebEmp demo application
- Source code
- Complete working environment set up to work out-of-the-box

What makes this project unique is that we can read the book, play with the demo application, take a look at the source code and most importantly we can experiment and learn.

## GENERAL

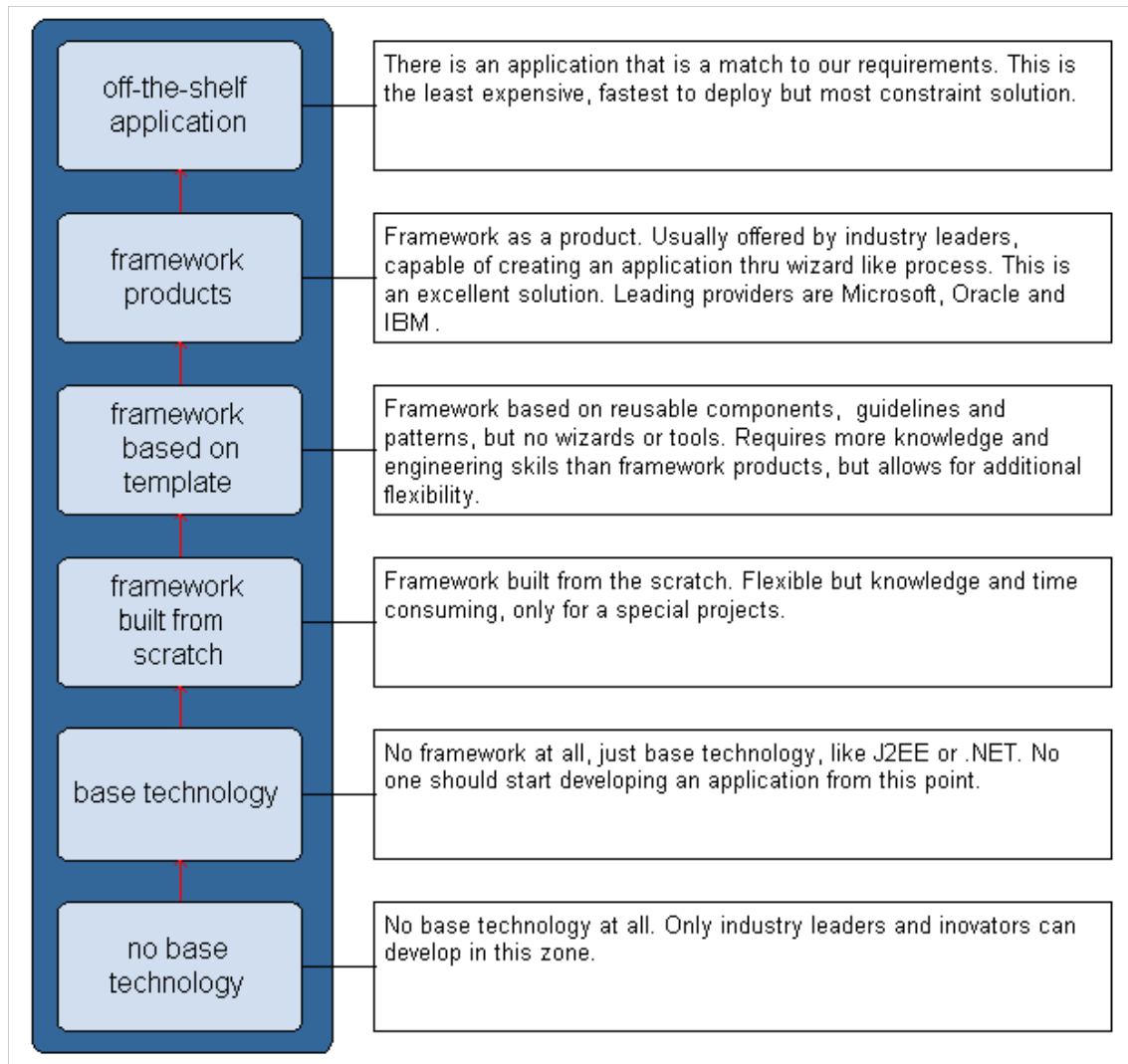
### ***Development Entry Point***

When the new application is needed, the first thing to do is to decide whether to buy or build one. Of course, if we are lucky enough that such application already exists, buying is probably the best thing to do. But, if the decision is to build one then next thing to do is to choose a development entry point.

There is a strategy that I have used many times so far and it has proved to be a good way to put things in the right perspective. It is very simple and it consists of identifying the extremes first, then all the rest falls in the place much easier. In this particular case, on one end of the scale we have core technologies like J2EE and on the other off-the-shelf applications. Everything in between can be considered as the framework zone with a higher or lower level of already provided code.

In general, the more complete the framework, the less effort is required. However, the solution is typically more locked in the framework provider technology.

Defining development entry point depends on many factors and mission at hand should be the final judge.



Development Entry Point Diagram

## Java Landscape

After finishing a complex J2EE application I felt, as there have to be a more practical and affordable way to build web applications in Java. When I looked at the Java landscape two new technologies have been emerging: Struts and JDO. It was very easy to recognize that the combination of the two is the right solution.

## Architecture

When designing applications we developers have to make a shift from the way we usually think when dealing with well defined and carefully designed core technologies to the world of patterns, object groupings, application frameworks,

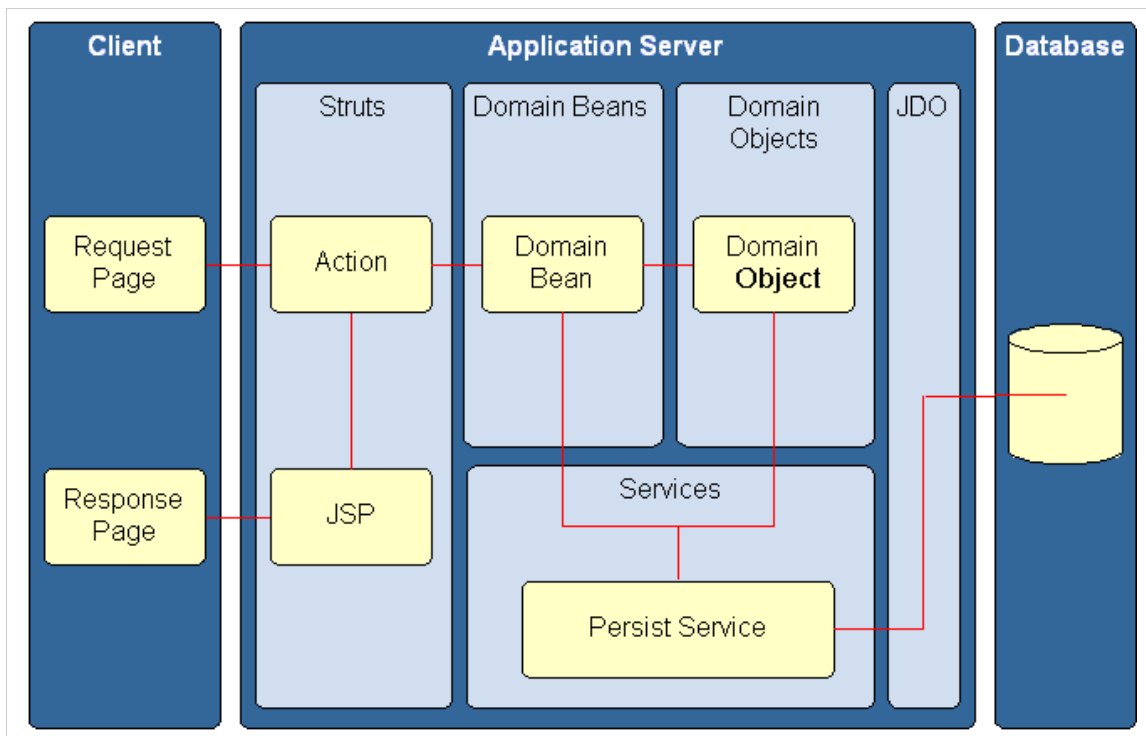
guidelines and even personal preferences.

When the technology is young very often there is more talk about the technology itself then how it can be applied. This is obvious in our development resources where books and articles about J2EE, XML and SQL outnumber those about application design.

Few years ago there were very few blue prints for Java web applications. I remember two: Sun Java BluePrints and IBM Application Framework for e-business. But as the technology is maturing we witness new frameworks every day. Some are provided by industry leaders and some by independent developers.

Web applications do require knowledge of many technologies: HTML, JavaScript, CSS, Struts, JSP, Java, Object Oriented Design, JDO and Object to Table Mapping. It is not easy to learn them all. It is even more difficult to blend them into the practical and affordable design. However, once Struts and JDO were selected it actually boils down to two integration pieces: domain beans and object link. Domain beans are connectors between domain and web parts of the application. Object link is the proxy for domain object.

Of course we should not forget to integrate subsystems: exception handling, logging, security, license validation, message repository, configuration etc. Some are provided by Java programming language, some by application containers and some we have to build.



Application Architecture Diagram

## **Solution Check**

Once a solution starts to take shape, how can we tell that it is on the right track? Let's take a look at the extremes once again. On one end of the spectrum we can take someone's word for it and in certain cases it is a valid strategy. On the other end we can hire third party team for full evaluation, very valid and very expensive solution. So what to do in all the other cases? We should put in place as early as possible build process, test bed and test cases, both functional and load. At the beginning it is not that important how accurate or complete they are, what is important is to be in the position to test, analyze and correct the solution. Apart from having a correction mechanism in place this is the great way to measure progress of the project.

Another great thing about this is that we are forced to think about the solution as a whole. It is not just about the code. It is not just about the network traffic. It is not just about the user interface. It is not just about the domain. It is about the complete solution.

## DOMAIN

### *Domain Type*

Domain can be either table centric or object centric.

Table centric domain captures data and its relationships in the database tables. Functionality is at best defined as an API-like set of stored procedures and at worst SQL statements are all over the code. This brings us to the biggest issue I see with this design. Implementation quality can vary a lot depending on the development team experience and time constraints.

Object centric design brings control and discipline to the project, but it does require a quality persistence code. Before JDO this was out of reach for most web projects.

Just to be clear, JDO integrates SQL into Java and developers do not have to use SQL but database schema and database itself have to be designed and maintained. Even though most JDO implementations will generate database schema, in reality it is a starting point. For the best results team work between application architect and database administrator is recommended.

It is worth mentioning here that JDO does not have to be used only for domain objects. For example, JDO would work well with data transfer objects for reporting or data export purposes. This illustrates previous point how JDO integrates SQL in Java but it does not impose new database philosophy.

There are number of JDO implementations available. For this project I have decided to use KODO. Distribution package comes with KODO license adequate to run demo application, but for the development tasks either evaluation or development license have to be obtained from SolarMetric.

### *Implementation*

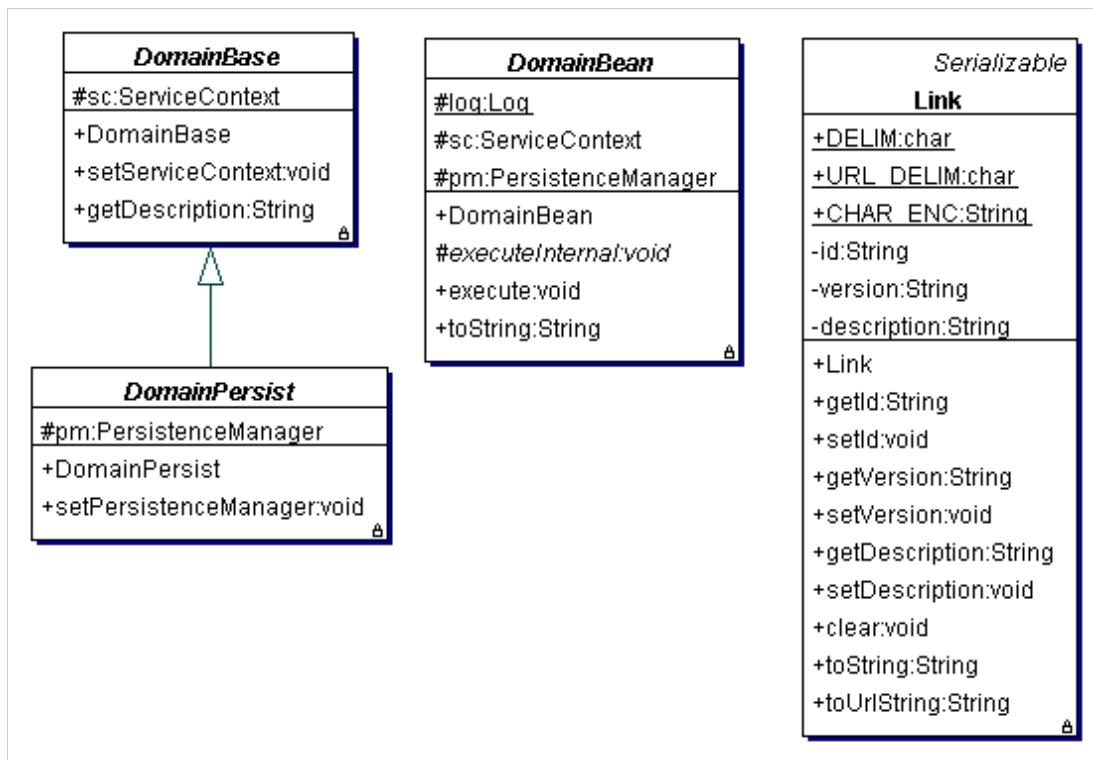
Domain objects are regular Java classes. This is what makes this solution so nice and practical. DomainBase is the base class that all domain classes inherit from.

It passes down to all domain objects functionality to hold a reference to the service context. Services are explained in more details in the next chapter.

It also introduces basic object description value. However, object description has to be defined for each specific domain object. Method getDescription should return single line object description but descriptive enough so the users can recognize the object by this description on the user interface.

DomainPersist is the base class that all persistent domain classes inherit from. By

doing so they gain functionality to access JDO persistence manager.



Base Class Diagram

## Object Link

Quite often we underestimate how important is to name things right. This is the reason why I have to start this very important topic with an explanation. Once I started writing this book I have realized that Link class can be easily confused with the web link (hyperlink), often called link for short as well. To be able to distinguish between the two, this book is using following terms: object link and web link. Object link is implemented as Link class and web link is implemented as HTML anchor element.

To manipulate objects inside Java environment we use object references that point to actual objects. However, in the case of web applications data has to be sent outside this environment to the web client. To be able to reference objects outside their natural environment we have to introduce a concept of object link.

Implementation wise we need a utility object that can act as a proxy for other objects. Additionally this object has to be able to serialize/deserialize itself when being exported/imported to and from the web client.

Link class is an object link implementation. It carries enough information for an object to be uniquely identified, recreated if required and recognized by the users on the web page.

<i>id</i>	Holds object unique identifier, value returned by JDO getObjectId method.
<i>version</i>	Holds version number, value required for optimistic locking.
<i>description</i>	Holds object description, value returned by getDescription method.

*Link Class Attributes*

There are two ways how Link object can be serialized/deserialized to a web client.

### **String Format**

Link object is serialized to a string using its toString method and value assigned to HTML Form element. Deserialization is done by LinkConverter class that takes string and creates link object.

### **Web Link Format**

Link object is transformed to a web link by using LinkTag, custom JSP tag. Link object data is stored as query parameters. When the request from web link hits the server ActionBase getLink method retrieves data from the query parameters and creates link object.

## **Object to Table Mapping**

Object to database table mapping is getting standardized and better documented. Here I just want to point out how it can be done in iterations and what the benefits are.

First iteration is to create initial database schema directly from the domain objects. This does not require much object to table mapping knowledge and it allows for the fast start.

Successive iterations should bring refinements in performance and data readability. This does require much more object to table mapping knowledge but it can be done without disturbing domain objects. Iterations can even continue after development phase in the production environment.

This is a huge advantage over the design where SQL penetrates deep in application layers and each database schema change can possibly break code all the way to the user interface.

## **Object to Object Relationship**

Objects can have relationships in which case they reference each other. Relationship references have to be maintained. It is a multi-object-transaction-like action where some references are cleared and some set. In essence this is an object oriented design issue, but driven by JDO community as it would significantly improve usability and quality of JDO based applications. Personally, I am looking forward to any progress in this area.

I have chosen to talk about this for another reason. This is a great example of one of the most difficult things in object oriented design I have experienced. That is, to decide how generic or how specific a solution should be?

On one hand it is very appealing to write a piece of code that is universally applicable and it works in any scenario. Object oriented design tends to put this temptation in front of us. On the other hand, it is the fact that it is much easier to understand and implement specific than generic solution. So how generic we can afford to be? Here are few solutions starting with the most specific:

### **Controller Level Implementation**

The most specific and easiest to implement solution. Object relationship maintenance code is done inside the controller. In our case domain bean is the controller. Code is not reusable. Object interface consists of standard accessor methods that are not relationship aware. When used outside the controller, in regard to object relationship, objects can be in the corrupted state.

### **External Method Implementation**

Very similar to the previous solution with the difference that object relationship maintenance code is moved to the separate, external to the object method and it can be reused.

### **Built-in Domain Object Implementation**

Object relationship maintenance code is built in the object itself. Accessor methods are relationship aware preventing objects to get in the corrupted state relationship wise. Object interface does not require additional coding instructions for object relationship maintenance. Requires serious testing. For basic implementation take a look in WebEmp source code.

### **Built-in Domain Object Generic Implementation**

The best and most generic solution. A task for the expert group. Take a look at EJB 2.0 specification.

As a rule of thumb, in the case of serious problems we should go one level down to a more specific solution.

## **WebEmp Domain**

WebEmp is the companion demo application. It implements party pattern with roles in the familiar employment domain. This is a well documented design, here are few highlights that will make reading class and database schema diagrams easier.

Parties like person and organization can play many roles. In our case we need just two: employer and employee. As employers and employees, parties can enter in the employment relationship. One employer can have many employment relationships with employees and vice versa.

On the separate branch a party can use many addresses as well as one address can be used by many parties.

Employment and AddressUse classes are introduced because both relationships have its own attributes.

Objects from the same inheritance tree are mapped to the single table. Person and Organization to the PARTY table. Employer and Employee to the ROLE table.

Employment and AddressUse each have its own table. This would be the case even if the relationships did not have its own attributes. The reason for this is that in both cases relationship cardinality is many to many and this requires cross reference table.

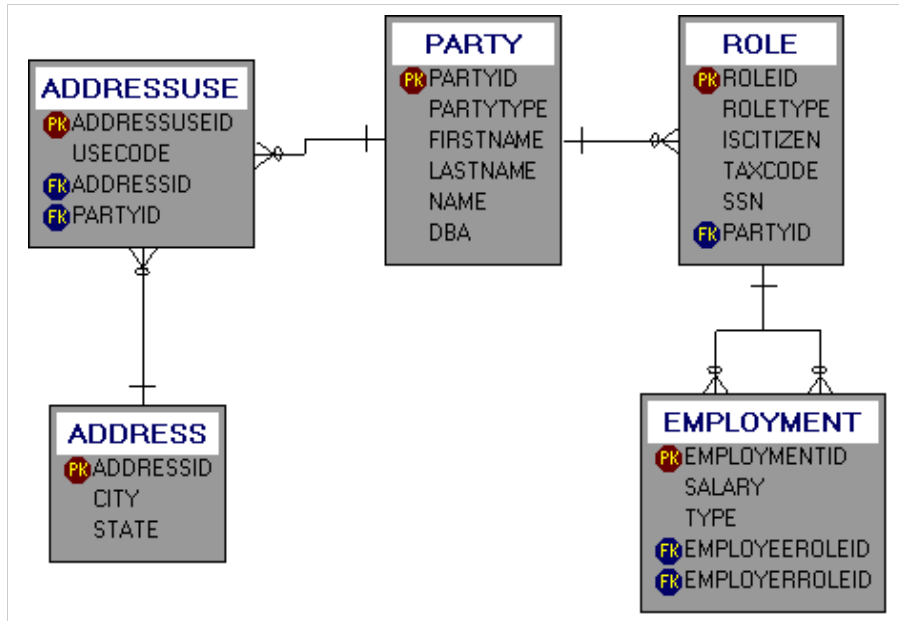
Let me finish with an example from this project how important load testing can be for the domain design. As we already know organization can play a role of employer and person can play a role of employee.

This is a great way to design the domain but not necessarily what application users want to see. Based on the application context they are presented with parties in their appropriate roles. In such case we need access to both party and role data.

For example to build a web link to an employer, organization data is required. This is what users use to visually identify employer. Everything works fine and all functional tests are OK. But during the load testing it is easy to spot a bottleneck when creating a list of all employers. When the volume of data increases it is about 10 times slower then it would be if the data were available on the employer role itself.

Because this is a demo application I had a luxury to ignore the issue.





Database Schema Diagram

### Version 1.1 Notes

Object link delimiter is changed from & to \$.

Object link *type* attribute containing the class name is removed because class name was already a part of KODO object identifier. This has reduced web link size quite a bit. The price for this gain is that to port WebEmp to another JDO implementation could require adjustment in the persist service based on object identifier format.

Version attribute has been added to object link.

All the functionality related to the JDO has been moved from DomainPersist class to the persist service. This has resulted in much cleaner code.

## APPLICATION SERVICES

Application services are utilitarian subsystems that all applications need. The list can be quite long: persistence, logging, message resource, configuration files, utility libraries, license checking etc. I like to think about them as equivalent of plumbing, electrical, heating installations in buildings. Once the structure is done all those services have to be put in place.

Service context is the control manager for the application services. To get access to the service, service context reference has to be obtained first.

Application servers may offer such services in which case server documentation have to be consulted and a decision about possible server dependency made.

Before we can implement our services let's identify possible service scopes:

- JNDI, across many JVMs
- JVM, one JVM
- Application session, one application
- User session, one user
- Thread, one thread

Because many web applications can be hosted in one application server it is very important to get the scope right. For example utility library is typically implemented as a static class giving it JVM scope.

Implementation wise it comes down to how a reference to the service context can be obtained anywhere in the code. For example application servers that have a control over thread creation can use thread context. This way there is no need for each class to have a object reference access mechanism. Also access to the static class is built in Java itself, it does not require reference at all. In our case we have to pass service context reference through the layers. First the service context instance is created and stored in the user session. Then it is passed to the domain bean and optionally to the domain objects.

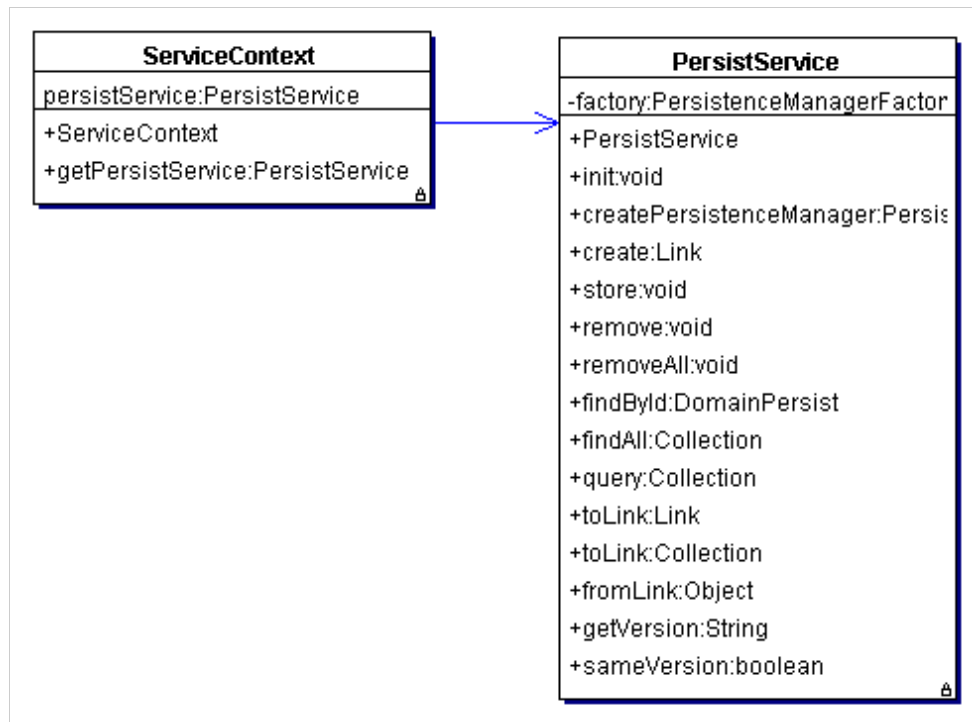
### *Persist Service*

Persist service does not have a goal to put yet another wrapper around an excellent JDO interface. By the way, it is a bad practice to introduce unnecessary wrappers, unfortunately often used by some developers.

In the persist service case this is where application specifics are introduced and basic JDO functionality centralized. Access to the JDO persistence manager is available and it should be used whenever its full power is required.

To persist an object with JDO we need access to the JDO persistence manager

factory and JDO persistence manager. Persist service creates persistence manager factory during its initialization at which point connection and configuration data is loaded. To actually persist objects, persistence manager instance has to be obtained from the factory and made available to all the code involved in the transaction. Persistence manager reference access mechanism is built in DomainPersist class.



Service Class Diagram

### Version 1.1 Notes

In the version 1.1 all persistent methods are moved from the DomainPersist class to the persist service. This way code is much cleaner.

## DOMAIN BEAN

Domain beans are connectors between the domain and user interface. On the domain side they have access to all domain objects. On the user interface side they provide all the data required by user interface.

### *As a Controller Element*

Domain beans are what is known as controller element. Their purpose is to navigate domain objects until fulfilling its goal. Domain beans do not represent reusable code and cannot call each other. It took me a while to figure this one out, maybe because I was influenced by EJB session bean design. Anyway, once I have realized this everything become much more simple. If we notice that certain portions of the code can be reused, such code should be moved to the domain objects where it belongs.

### *As a Transaction Boundary*

Most importantly domain beans define transaction boundaries. Each call to the domain bean represents one transaction.

To understand this issue better let's take a look at web application environment. On one side, web server is multi user environment possibly resulting in many concurrent requests. Each request possibly making multiple database calls. On the other side, data is unique and must not have multiple versions.

To manage this, database calls have to be grouped and executed as a set, one set at the time. This is what transaction mechanism does. Transaction mechanism is already implemented in the database and it will do its job well. The real responsibility lays in the hand of application designer and the way in which transaction mechanism is engaged.

Domain bean brings following benefits in this regard:

- Transaction boundaries are crystal clear
- Code is centralized
- Room for errors minimized

Another aspect that has to be considered is the possibility of users stepping on each other data. Popular solution to this problem is to label the data on retrieval with the version number. Each user gets the latest data version number. When it comes to applying the possible data changes then it is on the first come first serve basis. Changes done by the first user are committed and other users are asked to refresh the data. I have not implemented data version check in WebEmp

application, however complete infrastructure is in place. Take a look at Link and PersitService classes.

## ***As a Data Detachment Point***

Users need access to the domain data and functionality. This is done through the user interface.

Traditionally user interface consists of a series of screens with the form-like data structures. This means that translation between domain objects and user interface has to take place in both directions.

There are attempts to avoid this translation process and give users access to the domain objects directly. Take a look at Naked Objects project. If we think about this, objects are already well defined interfaces. Do we really need to create another layer above? The best thing about this approach is potential for automation, which means savings. Whether this concept will gain substantial user support, I guess time will tell.

In the meantime this is where domain object to user interface translation takes place in our design. Also this is the point where data required by user interface is detached from the domain. Domain itself is a maze of interconnected data. We need a way to detach some. Primitive data is detached by assigning domain object attribute values to the domain bean attributes. Domain objects are detached by assigning their object link.

Because domain beans are Java Beans we can use bean utility library for data manipulation. This produces compact and clean code, but prevents use of primitive data types for domain bean attributes. Instead wrapper classes are required.

## ***Implementation***

Domain beans are regular Java classes. DomainBean is the base class that all domain beans inherit from. It is a place to implement code common to all domain beans:

- Creating persistence manager
- Starting the transaction
- Committing the transaction or rolling back in the case of an error
- Domain level exception handling
- Domain level logging

For the specific domain bean we have to define a list of attributes and implement executeInternal method.

## WEB USER INTERFACE

Web user interface has its own distinct characteristics. In particular, the way application is navigated. It is very important for both designers and application users to understand and agree on this, as it can be quite different from previous user experiences.

When applied appropriately web user interface works great. Let's see what *applied appropriately* meant over the past decade.

### **Read Only**

At the beginning HTML was read only.

### **HTML Forms**

The basic interaction features were added.

### **Rise of the Web Browsers**

Web browsers power and popularity grows to that point that they were portrayed as ultimate application containers.

### **Web Application Frameworks**

Finally, things shifted again moving much of the presentation handling back to the web server. As a result, number of web application frameworks were created. Struts is one of them.

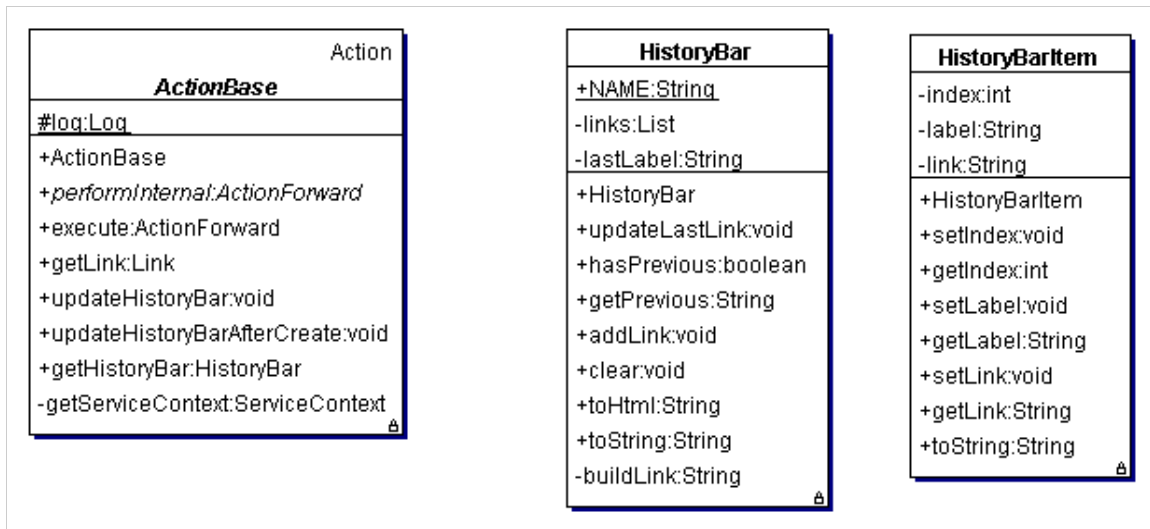
## *Actions*

Action is a user interface controller. It should be used to navigate domain beans and control user interface flow. Domain related code does not belong here.

ActionBase is a base class that all specific action classes inherit from. It is a place to implement code common to all Actions:

- Service context retrieval
- Web level exception handling
- Web level logging
- History bar functionality
- Utility code

For the specific action we have to implement performInternal method.



Web Base Class Diagram

## Screens

User screens are template based and are assembled from number of pieces.

There are two types of screens: single-request and multi-request.

Single-request screens are the screens where all the data that exists on the screen can be retrieved in the single request, for example employee screen.

Multi-request screens are the screens where multiple requests are needed in order to retrieve all the data, for example employment screen. They are very close to workflow or wizard implementation. Typically there is a base screen but in order to gather all the data other screens have to be visited. Because of this data has to be kept in session instead request. This is done by giving Struts forms session scope.

## History Bar

History bar is a list of web links that we can use to go back to the starting point. We can think about it as a kind of dynamic menu. Apart from its primary navigation purpose it is an indicator where in the screen hierarchy we are. This is a very important feature specific to web applications. A change from desktop application experience where similar effect is achieved by use of pop-up windows.

## Custom Tags

Link tag transforms object link to web link. Web links created in this way are used to navigate the application.

When creating a web link HTML anchor HREF attribute value has to be URL encoded and description HTML escaped, meaning that characters which can cause HTML parsing to fail have to be replaced with their codes.

### **Version 1.1 Notes**

All Struts action forms are replaced with Struts dynamic forms.

To make Java web application performant from a get-go, java server pages have to be compiled. This used to be a part of build process. It worked fine but target server has to be known in advance. To make application easily portable between application servers I have replaced build-time with run-time compilation option. This option is based on Precompilation Protocol, according to which application server can be asked to compile java server page. To compile all WebEmp java server pages, please use web link available on the application home page. For implementation details, take a look at JspPrecompile class and jsp.properties file.

## DATA FLOW

One of the great advantages of object oriented design is that data and behavior are tied together but it is hard to imagine an application where data is not moved around on its own. I believe that understanding the data flow greatly helps in understanding the application architecture. Following table shows data flow and format through all the layers in our design. Vertical columns represent two ways to exchange data with the HTML client: HTML Form and web link.

<i>HTML Client</i>	<u>HTML Form</u> Data is stored in HTML Form elements like text boxes, select boxes etc. Those elements contain string representation of Struts form and/or java bean attributes.	<u>Web Link</u> Data is stored as a series of tag=value pairs called query parameters.
<i>Struts</i>	<p><u>On the way out</u> &gt; First, data is moved from domain bean to Struts form and/or other java beans, then it is moved to HTML form elements. In the case of object link Struts is using its toString method.</p> <p><u>On the way in</u> &gt; First, Struts moves data from the request to the Struts form then the data can be moved to domain bean. Struts uses LinkConverter class to instantiate object link from its string representation.</p>	<p><u>On the way out</u> &gt; Data has to be built in a web link. Data can originate from the domain bean but it does not have to. Custom Link tag converts object link to web link.</p> <p><u>On the way in</u> &gt; Query parameters have to be retrieved from the request. Data can be passed to the domain bean but it does not have to. ActionBase getLink method retrieves data from query parameters and creates an object link instance.</p>
<i>Domain Beans</i>	Data is moved between domain objects and domain bean. Domain bean represents a well defined flat data structure that can be moved outside domain. Domain objects have to be replaced with object links at this point.	
<i>Domain Objects</i>	Data is moved between domain objects and database by JDO. Domain objects could have complex relationships but JDO only loads data when it is needed.	
<i>Database</i>	Data is stored in database tables. Database holds all the data and it is a System of Record.	

Data Flow Table

## CONCLUSION

There are so many options and ways to do this, ranging from guidelines to model driven application building engines.

However two new technologies Struts and JDO allow for web applications to be built in practical and affordable way. They bring many benefits as well as few difficulties.

I believe this is a well balanced solution. It is heavier than the usual web application but much lighter than an EJB solution. Amount of required infrastructure code is minimized, with the vendor independence preserved.

By now I want to thank you all, in hope that this book, companion demo application and source code have provided both playground and the recipe how to build web application in Java.

If you would like to know more about the project or you have suggestions please, do not hesitate to contact me at [www.firdus.com](http://www.firdus.com).

## LITERATURE

### **Books**

- Information System Architecture, Martin Fowler
- Java Modeling in Color with UML, Peter Coad, Eric Lefebvre and Jeff De Luca
- Java Design Patterns, James W. Cooper
- Enterprise Modeling with UML, Chris Marshall
- The Art of Objects, Yun-Tung Lau
- EJB Design Patterns, Floyd Marinescu

### **Articles**

- IBM Application Framework for e-business, IBM
- Sun Java BluePrints
- Mapping Objects to Relational Databases, Scott W. Ambler
- Dealing with Roles, Martin Fowler
- Passport to Intranet Architectures and Performances, TechMetric Research
- David Jordan, Managing Your Relationships
- George Reese, Java Database Best Practices - JDO Persistence
- Scott Ambler, Implementing Referential Integrity and Shared Business Logic
- Andy Bulka, Relationship Manager Pattern

### **Specifications**

- JDO
- EJB

### **Product Documentation**

- KODO documentation, Solarmetric
- Struts documentation, Apache Software Foundation
- Tomcat documentation, Apache Software Foundation