ARTHUR ANDERSEN

Arthur Andersen Java™ Architecture

Developing EJB applications and reusable components using the Software Factory

Martin Israelsen

Contents

Introduction

The requirement for bigger, faster and more scalable systems makes the development life cycle more complex than ever. The latest distributed technologies like Microsoft DCOM and Enterprise JavaBeans™ is becoming key technologies for creating scalable and reliable enterprise wide system in the global marketplace

DCOM and Enterprise JavaBeans are complex technologies that require significant knowledge and expertise from developers and project managers. Even with strong technological expertise it is often hard to meet the project budget and deadline constraints due to the steep learning curve.

The Software Factory/EJB enhances the development process by optimizing the modeling cycle, automating the object-to-relational persistence mapping and removing or hiding many of the low-level EJB requirements. This leaves the developer with more time to focus on the business domain.

Even with the Software Factory automation Enterprise JavaBeans still contains many pitfalls and gray areas. This document is written as a guide to EJB projects and will hopefully assist in avoiding most or all of the most common pitfalls and gray areas.

The document is based on Enterprise JavaBeans 1.0 using the BEA WebLogic EJB server. While the Enterprise JavaBeans specification leaves plenty of room for each EJB-server vendor to interpret the EJB specification differently, the overall picture is likely to be the same for most EJB implementations.

Please forward comments to Martin.Israelsen@us.arthurandersen.com

EJB architecture overview

The Enterprise JavaBeans architecture is build upon a number of different components. It will be hard to find two identical EJB implementations, but most implementations look like the one outlined below:

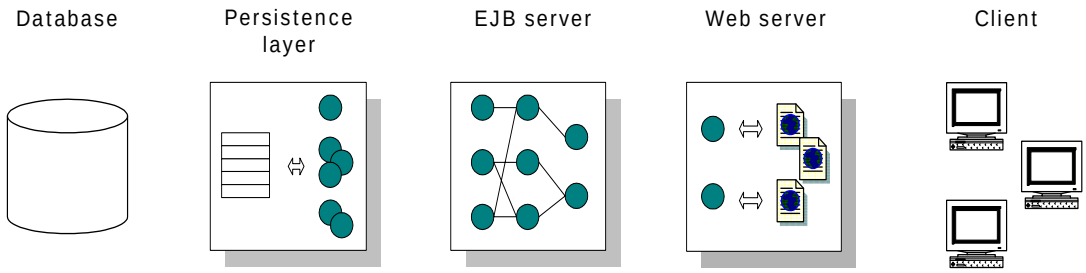


Figure 1 Typical EJB architecture

Database

The database can be any database that is supported directly by the EJB-server or any database that is supported though JDBC (if supported by the EJB server).

The database can either be an object-orientated databases e.g. DataStore, or relational as Oracle, DB2 or SQL-server.

Depending on the application and use of the entity beans, object-orientated databases can perform better than the relational counterpart, but can be harder to support due to the small percentage of resources with object-database experience.

Persistence layer

The persistence layer isolates the EJB server from the database and is responsible for storing the state of all EJB objects in the database. The persistence layer is also responsible for the object-to-relational mapping when using relational databases as the persistence mechanism.

Some EJB servers come with the object-to-relational mapping layer built-in. EJB servers with built-in object-to-relational mapping can connect directly to the database.

EJB server

The EJB server is the central component of the architecture. The key features of an EJB server are transaction management, resource management, workload distribution, fallover and security authentication and authorization.

To be EJB compliant, it is must support a number of required features as described in the EJB specification from Sun.

Web server

The web server is not mandatory in the EJB architecture, but has been included in the diagrams because many EJB implementations are inter- or intranet based.

Client

The client can be any client application, e.g. a Java program or a VB or C++ program. Since EJB is based on Java most client applications are likely to be Java based as well, but this is not a requirement. Non-Java applications can connect to EJB servers using CORBA or DCOM gateways.

EJB architecture features

Most Enterprise JavaBeans servers offer advanced features including distributed processing, load balancing and fall-over.

These built-in features simplifies the implementation significantly, however it is important to keep in mind that the EJB specification 1.0 not addresses these areas directly, and the features as a natural result are proprietary to the vendor.

Distributed processing/Load balancing

By adding more servers, service requests can be distributed among the servers and balance the workload on each server. The reason is that if one server can handle 10000 requests pr. minute, but the requirements are 20000 requests pr. minute, we can increate the performance by adding yet another server.

The performance improvement is seldom linear, as some computing power is used to manage the load balancing itself. Other factors like network bandwidth and database capacity will affect and limit the total throughput.

Although the EJB specification supports distributed processing, actual implementation is not addressed in the EJB specification, and implementation is usually proprietary of each vendor.

Fall-over

We can provide fall-over support by adding more servers to the architecture. If one server stops responding, the other servers can take over and handle the requests for the missing server. However, where infrastructure for load balancing usually dictates to locate the servers physically close to each other to improve performance, infrastructure for fall-over dictates that the fall-over servers should be located in geographically independent locations.

Software Factory overview

The Software Factory consists of a number of individual components:

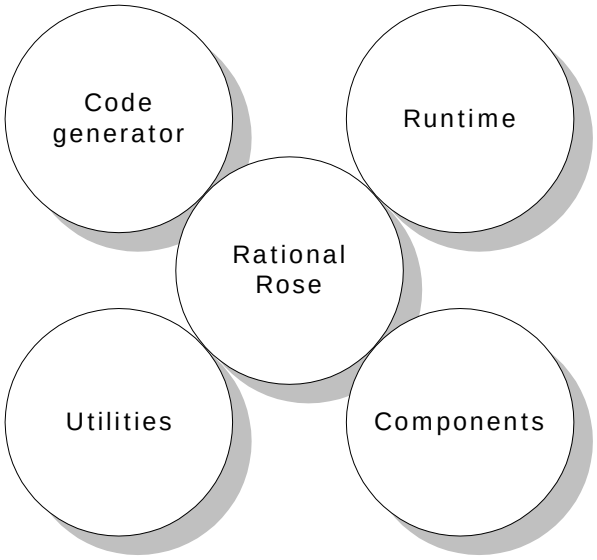


Figure 2 Software Factory components

Rational Rose

Rational Rose is the modeling tool. It is used to create and maintain the to-be object design diagrams. Rational Rose is not part of the Software Factory install and must be purchased and installed separately.

Code Generator

The Code Generator generates source code and database definition files from the Rational Rose models. The current version of the code generator supports Rational Rose models only, while the generated source code can be targeted to EJB, VB/ADO or JDBC.

Runtime

The Runtime component is a language specific extension layer that supports the code generator generated modules and provides often-used methods and classes. This layer is written specifically for each target platform.

Components

The Components group is a language specific extension layer that contains often used components. Aside from the code generator itself, the single most important feature of the software factory is ability of reusing already developed classes and modules on a number of engagements.

Examples of reusable components already developed or in development include a web-based user interface system that incorporates all features needed to enable user security and login controls. It can be used directly in any EJB/WEB based engagements where the need for user control exists, e.g. eBusiness applications.

Utilities

Utilities are a small group of programs to reduce often-repeated tasks. Among the tools are build and EJB compilation utilities, help-file builder and logging utilities.

A number of tools have been created to support the development process. It is outside the scope of this document to throughly describe all the tools, but there is a few worth mentioning.

*The Build utility***.** Compilation is the most performed process of the development cycle. The Build utility assists the developer in the compilation end EJB wrapping phase. The wrapping is necessary before the components can be deployed to the EJB server, whether it's a deployment server or a local server.

When performing a total build, wrapping consumes more than 95% of the overall time spent. For the same reason, this build utility has been developed and deployed into the Java architecture. The build utility analyses the existing sounds and class files, and compiles and wraps only when needed.

Build Help utility. The build help utility generates Javadoc-based helpfiles. It eliminates the recursion problem with Javadoc.

*EJBPing***.** The EJBPing is a simple utility that checks the connection to any EJB server running the EJB architecture, by establishing a connection to IDServerBean. EJBPing can be used to quickly verify if the server is running and whether it is possible to connect using the architecture.

Designing using the EJB architecture

It is outside the scope of this document to describe the analysis and design phases using RADFrame and RADframe/OO, instead this chapter will focus on designing a model and generating the model into the EJB world.

Rational rose diagrams

A prerequisite for generating anything using the code generator, is a Rational Rose diagram. If you're already familiar with UML or ERD diagrams, the Rational Rose UML diagrams will not come as a big surprise.

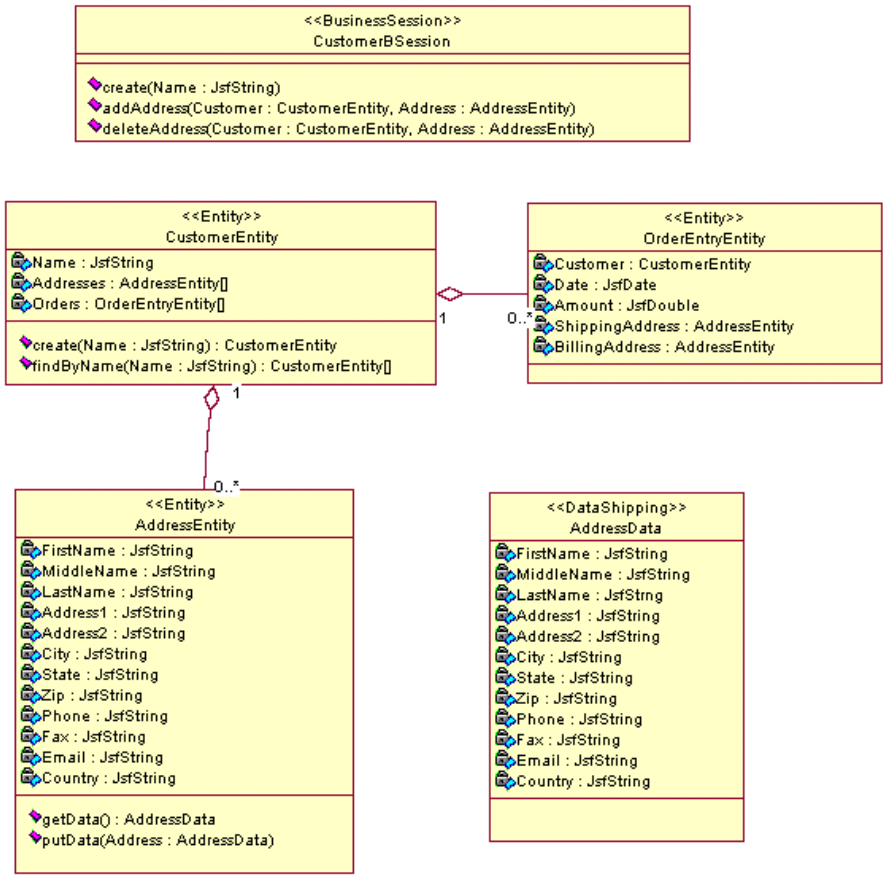


Figure 3 Rational Rose diagram

The Rational Rose diagram represents all classes and relationships that we want the system to contain. The diagram is the only input needed by the code generator to

generate the database scheme, the object-to-rational mapping layer and the actual classes.

To successfully generate a model, there is a number of syntax and naming conventions that must be adhered to. These rules are thoroughly described in the code generator documentation.

In addition to the syntax rules, a model should adhere the to architecture layer model as described in the next section. This will ensure the optimal performance

Architecture layers

This section describes the areas of the architecture framework which must be addressed when object modeling at the design phase.

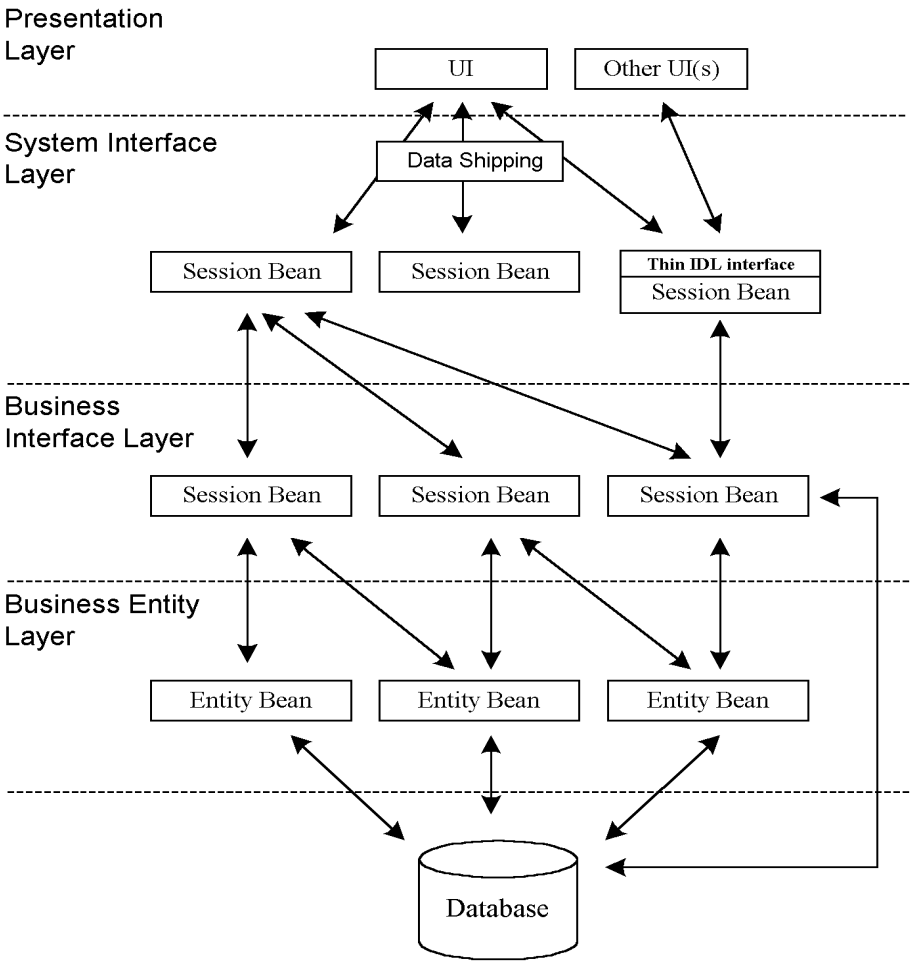


Figure 4 Architecture layers

The EJB architecture consists of a number of logical layers. The layers each serve a distinct purpose:

Business Entity

This Business Entity Layer contains entity beans that directly represent the majority of business entity types in the system, and tend to be analogous to real world objects. Operations on these beans should be focused on maintaining the consistency of state of the bean itself, and less on controlling behavior of graphs of objects.

Business Session

The Business Session Layer is the computing part of the EJB architecture. This layer, implemented as stateless session beans, provides a generic interface to the business domain and in combination with the Business Entity Layer, can be viewed as providing reusable business components that are independent of the presentation of a particular application. An individual business session bean controls the behavior and state of a number of entity beans. As a result, the majority of the business logic is contained in this layer and the beans are considered coarse grained or "heavy".

System Session

The System Session Layer is the last layer within the application server. The system session layer consolidates a number of Business Session Layer calls to provide data to the presentation layer or client. As a guide, a single session bean can control a single screen or a set of related screens. In this way, if a screen requirement changes, the impact of this change can be easily identified in the screen or process specific System Session Bean.

Data Shipping

The Data Shipping Classes are used to transfer data around the system and between the EJB server and the client. Packaging data in shipping classes improves performance significantly compared to multiple accessor method calls.

Architecture considerations

Despite the ever-increasing performance and faster computers, designing and implementing Enterprise JavaBeans systems requires performance and scalability considerations as with any other implementations.

Most Enterprise JavaBeans server do offer fall-over and distribution automatically, but these features are only available if the system has been designed to take advantage of them.

Performance

The object model must be modeled to reduce cascading object access. Because the model resembles an SQL-database, the same rule for denormalization and normalization applies (denormalize till it hurts, normalize till it works).

Software Factory systems can take advantage of the 2-way navigation built into the architecture. The 2-way navigation is slower than 1-way navigation when inserting and deleting objects because the association has to be updated two places, but greatly improves navigation and searching.

Performance tests should be conducted throughout the project to verify that the performance is within the defined range.

Reuse

As with all other system development projects, efficient reuse of already created components has a significant impact on budget and schedule. Not only is it significantly faster to do the initial development but testing and rollout will also be affected because the component comes fully developed and tested.

EJB project teams are encouraged to take advantage of the Software Factory component library and submit new components to the GSECE team.

Platform independence

Despite the "write-one, run everywhere" slogan, the developer must take care to avoid platform dependencies. The most common platform dependencies are within a small group:

Filenames. Filenames and paths are represented differently on different platforms. Use the file pathSeparator and separator fields to create filenames that are valid on all platforms.

Threads/locking. Different platforms utilize different threading mechanisms. The windows platform uses pre-emptive multitasking where Sun Solaris does not. The threading mechanism can impact internal synchronization and can lead to system deadlocks if not handled properly.

Capitalization. Some platforms are dependent on capitalization of filenames and parameters while others are not. Sun Solaris, for example, require a filename to be capitalized the exact same way as the file itself, whereas the Windows platform allows any capitalization when opening a file.

Database. The container-managed persistence in EJB eliminates most queries and other database operations. Still, there might be valid reasons for direct database access, e.g. when optimizing search methods. When doing direct database access, care should be taken to utilize the built-in database drivers. This makes the application less database dependent, because databases can be switched just by changing the database driver.

Architecture independence

In all levels of an application, it is important to avoid dependencies on inner the functionality of any component.

A typical example is the entity key generated by the Code Generator. The serialized entity key consists of class and primary key, e.g.

com.arthurandersen.sfcl.customer.CustomerHome:452

For the developer, it is tempting to rely on this format to get either the owner class or the primary key value. Doing so will introduce dependency on the inner workings of the EJB architecture, and will break if the entity key format is changed.

Creating the model

Designing an EJB model from scratch is a significant task and it is not possible to cover all aspects in this document. Instead I will recommend that you peer with someone who has prior EJB modeling experience. Don't worry about the additional time used - Getting the model right the first time will save many hours later.

Getting started

Assuming that the analysis model already has been created, it is time to create the design (technical) model. This process is described in the document Design process in Rational Rose.doc. Some additional tips are provided below.

Start with a small model. Create a small throwaway model with just a few beans, generate, deploy and execute it. Modify it, generate, deploy and execute it a number of times. This one-day activity will give you a better understanding of the modeling, coding and deployment phases.

Inherit Software Factory components. Inherit from Software Factory components as much as possible to take advantage of the already built and tested components.

Add mandatory fields to create methods. Add mandatory fields to the create methods, e.g. pass name in a customer entity create method. This ensures that the entity is created with valid data. Create methods without any data can potentially lead to empty and invalid entities in the database.

Avoid findAll(). The entity should have designated finder methods for all possible search types. Designated finder methods takes advantage of the database query functionality and will perform much quicker than using findAll(). FindAll() returns all instances of the particular entity and can bring down the entire EJB server on large implementations.

The Code Generator

The code generator generates a number of different output files, depending on the configuration. Some of the output files are only created once while others have to be regenerated whenever the model has been changed.

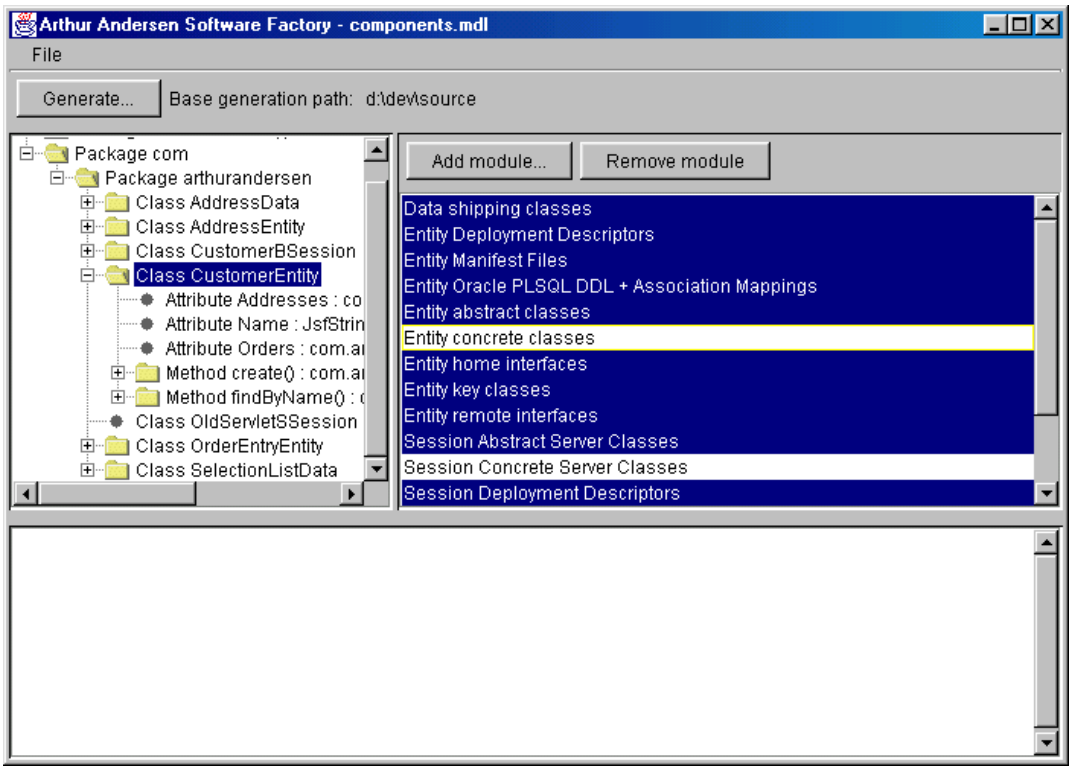


Figure 5 The Software Factory code generator

Data Shipping Classes

The Data shipping classes is regular Java classes used to store and transfer data between the server and client(s). The Data Shipping Classes should be maintained fully by the code generator and regenerated when the class has changed.

Abstract and concrete model classes

The most important of generated files are the abstract and concrete class files.

The **abstract** class file is created and maintained by the code generator. The abstract class contains all logic necessary to create, store and load entities, maintain associations and general housekeeping. The abstract class should never be modified, as it will be overridden next time the code generator is run.

The **concrete** class file is the placeholder for all business logic. The code generator initially generates the concrete file, but during subsequent generation, only the abstract class should be generated.

It is the developer's responsibility to maintain consistency between the abstract and the concrete class. When adding methods to the model, the concrete class has to be updated with the new methods manually. Likewise, when deleting methods from the model, the methods in the concrete class must be deleted manually.

Having an abstract and concrete class is slightly more cumbersome than having a two-way generation tool that automatically updates a single source file. The approach however, is fairly secure against accidental deletion of methods and/or members, and has proven to work well.

Entity Oracle PLSQL DDL + Association Mappings

The database scheme file contains the database-specific SQL DDL script to generate the selected entities. The database scheme file should be run in a database manipulation tool, like ISQL for SQL-server or SQL-worksheet for Oracle. The database scheme has to be regenerated every time an entity has been changed in the model. Failure to update the database scheme will result in a malfunctioning entity or an entity that cannot be deployed.

The association-mapping file describes the relationship between entities. It is used at runtime when linking entities with multiple associations. This file is automatically generated whenever the database scheme is generated.

> Note for Oracle users: The SQLDDL file contains characters that are not correctly understood by SQL-Plus. As a result SQL-worksheet version 8.0.60 or higher should be used instead of SQL-Plus when executing the SQLDDL file.

Deployment descriptor

All Enterprise JavaBeans have a deployment descriptor. The deployment descriptor contains various settings regarding database mapping, timeouts, number of cached beans, time-out settings and finder method implementation.

The deployment descriptor should be regenerated whenever the bean has been changed.

Note that the deployment descriptor defines the actual implementation of the finder methods. Because the code generator in the current version doesn't know the details of the finder method, the actual finder method has to be manually updated after generation of the deployment descriptor.

Home Interfaces

The home interfaces contains the creator and finder methods described for each bean.

The home interface is managed completely by the code generator and should not be changed manually.

Manifest files

The manifest files are used when the bean is added to a .jar file prior to deployment.

The manifest file is managed completely by the code generator and should not be changed manually.

Remote Interfaces

The remote interface contains all methods except the creator and finders methods described for each bean.

The remote interface is managed completely by the code generator and should not be changed manually.

Generating the model

When generating a model the first time, all modules in the right pane should be selected. After generating the model, run the SQL script and build and deploy all beans. Obviously there is nothing to run at this point in time, because we haven't written any code yet, but the build and deployment will verify that the model and environment has been set up correct.

After the initial model generation, make it a standard always to exclude the concrete class generation. The concrete classes should only be generated when adding new beans.

Adding beans

When adding beans to the model, the most efficient approach is to fully generate the new beans. This approach will ensure that all necessary files are created, including the concrete classes. Care should be taken not to generate any beans that already have been developed.

Modifying beans

When methods or member fields have been changed, but no beans have been added or deleted in the model, the most common approach is to re-generate the modified beans with all options selected **except** the concrete classes. This will ensure that the deployment descriptor, remote- and home interfaces and all other files have been updated correctly.

If any of the modified beans are entity beans, it will also be necessary to run the generated SQL script. It should be noted that the generated SQL script doesn't update the existing tables, but delete and recreate them. All data in the affected entities will

therefore be deleted. If the modifications are minor and you want to preserve the table contents, you might consider applying the changes manually using ALTER TABLE.

Deleting beans

The code generator will not delete classes for deleted beans and it will not generate SQL code to delete the entity tables. Housecleaning has to be done manually.

Before running the code generator, whether you're adding, modifying or deleting beans, you must be aware that the association mappings and SQL script are generated only for the selected beans. The resulting SQL script is very useful for performing incremental updates on the database. The developer can update the personal tablespace and test the modifications before submitting the changes to the central repository.

However, the generated association-mappings will also be limited to contain only the generated beans and is practically useless. To overcome this problem, generate the SQL script and association-mapping file for the entire model, and use these files when deploying the EJB server.

Writing EJB applications

This section contains various tips for the EJB application developer.

The Runtime classes

The Software Factory comes with a number of pre-developed modules named the Runtime files. Some of the modules are used solely by the architecture and should not be used by the application, while others are available.

A brief description of the most commonly used classes is listed below.

com.arthurandersen.common.EJBSystem

EJBSystem has a number of methods to setup and establish connectivity to the EJB server. Login information can be stored in the resourcebundle com.arthurandersen.commonproperties.EJBSystem.properties or it can be set directly in EJBSystem.

```
//
// How to use EJBSystem
//

public void testConnection()
  throws RemoteException, ResolveException
{
  // Override default values taken from com.arthurandersen.commonproperties.EJBSystem
  EJBSystem.setUsername("sysadm");
  EJBSystem.setPassword("password");
  EJBSystem.setURL("t3://localhost:7001");

  // Connect to server
  AASystem.resolveHomeInterface(IDServerHome.class);

  ..
}
```

com.arthurandersen.ejb.EJBEntity

The EJBEntity is the ultimate ancestor of all Software Factory generated entity beans. EJBEntity inherits from `javax.ejb.EntityBean`.

The EJBEntity contains the required EJB methods (`ejbRemove()`, `ejbActivate()` and `ejbPassivate()`) as functionality to store dirty (modified) beans and a few helper methods (`isModified()`, `getHomeInterfaceName()` and `getUserID()`).

com.arthurandersen.ejb.EJBSession

The EJBSession is the ultimate ancestor of all Software Factory generated session beans. EJBSession inherits directly from `javax.ejb.SessionBean`.

The EJBSession contains a few housekeeping and helper methods (`getEJBContext() and getUserID()`).

com.arthurandersen.ejb.idserverbean.IDServer

The IDServerBean is a database specific primary key generator. It generates and maintains sequence numbers for the actual database implementation. It is invoked by the create methods in the abstract entity classes.

> The current implementation of IDServerBean is written for Oracle databases. It is necessary to change the SQL in IDServerBean when generating code for other databases.

com.arthurandersen.system.AASystem

The somewhat funny named class AASystem contains various helper classes to resolve interfaces, home interfaces and entitykeys. The AASystem class reduces the usual 10-20 lines of code needed to instantiate a Enterprise JavaBean to just two lines.

```
//
// How to use resolveHomeInterface
//

public void createCustomer()
  throws RemoteException, ResolveException
{
  CustomerHome lCustomerHome = null;
  Customer lCustomer = null;

  // Instantiate the home interface
  lCustomerHome = (CustomerHome) AASystem.resolveHomeInterface(CustomerHome.class)

  // Create a CustomerEntity using the home interface
  lCustomer = lCustomerHome.create("John Doe");

  ..
}
```

com.arthurandersen.system.EntityKey

The EntityKey is the primary key in the EJB architecture. The EntityKey is a unique reference to all entities in the entire system, and all entities generated using the code generator will have an entity key.

Unlike a regular client/server primary key, the EntityKey consists of a reference to the bean class that is the EJB representation of the data as well as a unique number. The EntityKey is partly generated by the IDServerBean.

Having just the EntityKey, it is possible to instantiate the corresponding entity regardless of the class type.

```
//
// How to use ResolveKey
//

public void getCustomer(EntityKey lCustomerEK)
  throws RemoteException, ResolveException
```

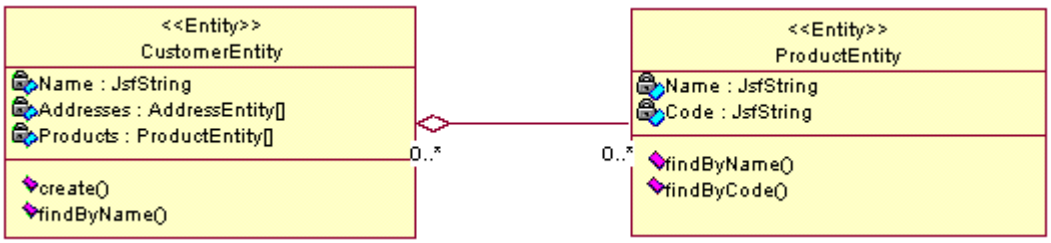
```
{
  Customer lCustomer = null;

  // Get the customer entity using the EntityKey
  lCustomer = (Customer) AASystem.resolveKey(lCustomerEK);

  ..
}
```

Association navigation

Because Enterprise JavaBeans is linked through objects rather than primary- and foreign keys, navigation are slightly more complicated and limited compared to 2-tier client/server implementations.

Objects are linked using associations. The architecture provides the necessary functionality to store and retrieve the association information, but it is the developer's responsibility to add and remove associations.

Associations can be linked as one-way to two-way associations. An example of one-way associations is when an object has a list of associated objects, but the associated objects don't have a link back.



.

Figure 6 One-way association (UML)

As shown in the example above, it is possible to navigate from the CustomerEntity to the ProductEntity, but it is not possible to navigate from the ProductEntity to the CustomerEntity.

When implementing two-way associations both objects will have a list to each other, thus providing navigation in both directions.

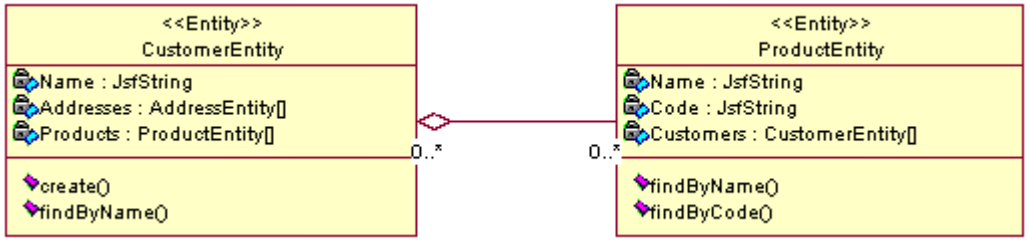


Figure 7 Two-way association (UML)

Even though two-way association requires more code than one-way associations, two-way associations are preferred whenever two-way navigation *might* be needed. Two-way navigation is preferred because it's more or less impossible to perform efficient navigation without association links. It is not possible to add a few SQL queries, as it would be in a normal 2-tier client/server application. Two-way associations should therefore be implemented if there is just the slightest chance that two-way navigation is required.

Exceptions

EJB exceptions have the same functionality as regular Java exceptions. Note that exceptions not declared in the method signature will be wrapped in a java.rmi.RemoteException.

Using Inheritance

Object orientated programming can be a huge time saver – but only when applied correctly. While object orientated languages is used in most projects today, only a small percentage utilities all the benefits of objects.

To recap from the Java books there are three levels of objects: Encapsulation, inheritance and polymorphism.

Encapsulation is the first and easiest level. The term refers to blending code with data. In older procedural languages, e.g. Basic, variables could either be local (within a method) or global (common for all methods). By encapsulating data, we store the data within the object itself, therefore making the data part of the code. When the object goes away our data goes away. When we instantiate (create) more than one version of the object each version will have it's own data.

```
class Account
{
  // encapsulated data members
  public float mBalance = 0.0;

  public void calculateRevenue()
  {
    // calculation code here
```

```
  }
}
```

When encapsulating data, we can benefit from making the data private and only provide access through accessor methods. The accesssor methods enable us to implement functionality that will be executed when data is modified.

```
class Account
{
  private float mBalance = 0.0;
  public float getBalance()
  {
    return mBalance;
  }

  public void setBalance(float pBalance)
  {
    mBalance = pBalance;
    calculateRevenue();
  }

  public void calculateRevenue()
  {
    // calculation code here
  }
}
```

Inheritance is the second level. When we inherit an object, we are basically copying the ancestors data members and functionality, but without actually copying the code. Since we don't copy the code, maintenance becomes significantly easier because the code only needs to me maintained one place. In addition to the features we get from the ancestor we can extend the object by adding new methods and/or data members, and we can change the functionality of the ancestor by overriding certain methods.

```
class InvestorAccount
extends Account
{
  private String mInvestorID = null;

  public String getInvestorID()
  {
    return mInvestorID;
  }

    public void setInvestorID(String pInvestorID)
  {
    mInvestorID = pInvestorID;
  }

  // Override calculateRevenue to provide investor specific revenue calculation
  public void calculateRevenue()
  {
    // calculation code here
  }
}
```

Polymorphism is the third and most complex part of object orientated development. Polymorphism can be translated into "something that has many shapes". By

polymorphism we are create a number of objects that have similar interfaces but different functionality.

In the example below, we create an account depending on the customer type. It can either be an investor account, a broker account or a regular account. When performing operations on the account, we don't need to specify how to calculate the revenue; the account subclass (either InvestorAccount or Account) is handling the calculation.

```
class calculate
{
  private Account mAccount = null;

  public void createAccount(int pCustomerType)
  {
    if (pCustomerType == INVESTOR)
    {
      mAccount = new InvestorAccount();
    } else
    if (pCustomerType == BROKER)
    {
      mAccount = new BrokerAccount();
    } else
    {
      mAccount = new Account();
    }

    // Calculate revenue, regardless of accounttype.
    mAccount.calculateRevenue();
  }
}
```

> Watch out for the use of instanceof in class hierarchies. instanceof usually signals class dependant actions outside the class itself and can be a sign of a poorly designed model.

Inheritance in EJB

In general the EJB inheritance model works like regular Java classes. However, there are a few issues that the EJB specification does not address.

Specifically, it is not possible to inherit and change the return type of a method. This is a problem when inheriting home interfaces as the create method(s) return the bean itself as the return value.

This problem is addressed in the code generator by generating separate home interfaces for each bean while inheriting the bean and remote interface.

Because the home interface isn't inherited, developers should be aware that instanceof only works with the remote interface and not the home interface.

Inheritance versus Interface

While several languages have supported inheritance for many years, interfaces are relatively new for many developers. An interface is used to apply a certain behavior to an object. When we implement an interface, we are committing to implement one or more methods with predefined signatures.

By implementing the interface, we enable other objects to interact with our object, regardless of the inner workings of our own object.

In the example below, we have defined an interface called Searchable. By implementing the Searchable interface in the customer object, we have created the behavior that is understood by the search mechanism, and we can now search in the customers using the existing search screen.

```
public interface Searchable
{
  public Vector search(String pSearchKey);
}


public class Customer
  extends Address
  implements Searchable
{
  public Vector search(String pSearchKey)
  {
    // Here goes code to search for customer.
  }
}
```

> Confused about when to use interface and when to use inheritance? Use inheritance to extend the functionality of an object. Use interface to enable another object to interact with your object.

Interfaces in EJB

When using interfaces in EJB, the implements keyword and must be declared in both the remote interface and the concrete class.

Creating and using Components

Components are the key technology for creating truly reusable code. The term component really refers to a complex class, and the differentiation between classes and components is somewhat blurry. I generally prefer to use the term component for classes that have high complexity and usually a multitude of functions. In other words: A class that can do various string operations is a class, while a class that handles accounting transactions is likely to be a component.

The most important feature of a component is that is can be adapted to specific needs in various occasions. A well-written component should be able to handle somewhere between 80-100 percent of the required features either directly or through minimal overriding.

It is our goal to extend the number of components in the Software Factory architecture, by creating and adding components that is likely to be reused on a number of projects.

Obviously it is not a small task to create a truly reusable component, and it is recommend that you contact GSECE for assistance when creating components that potentially can be part of the Software Factory architecture.

Using components

Because components is bigger than classes the use is also different. In the GSECoE architecture, you are likely to find components consisting of two parts: A Rational Rose representation and one or more corresponding source files.

The Rational Rose model will provide the interface for the component. Do not modify the Rational Rose representation directly as it prohibits you from using future updates. Instead you should use the component as the ancestor for the actual implementation.

Wrap components and classes

When using components, whether it's Software factory components, native Java components or third party components, it is worthwhile to consider creating a wrapper or layer class.

The typical wrapper class is often small or even empty in the beginning of the project, but as the project progresses the wrapper class can be used to implement central client specific coding and work-around when the source code isn't available.

An example of a wrapper class is the client logger class. The wrapper class was created to enable us to implement an email-based notification system so the helpdesk would receive an email every time the system encountered a fatal exception.

```
Class com.client.util.logger extends com.arthurandersen.util.logger
{

  public error(String pMessage, Exception ex)
  {
    // send email to development team regarding this bug
  }
}
```

Creating reusable components

Creating reusable code has always been an important requirement, but often hard to accomplish due to different architectures, platforms or requirements. Platform and architecture extensions like ActiveX, DCOM and Enterprise JavaBeans are making it easier to create cross-language or cross-platform components, but the requirement for

Before creating a component it is important to plan for the usability.

- When is this component to be used?
- Under what circumstances is this component to be used?
- What future requirements might be added later?
- Should the component be added to the Software Factory library?

When doing the planning exercise for components development, it is important to plan for future enhancements. Technology has a tendency to change, often during the development phase, and although scope creep needs to be avoided as much as possible, user requirements can influence development significantly.

While in the planning stage, it is advisable to avoid words like *never* and *not needed*. Since the component, like all components, essentially becomes a part of the architecture, it must be designed in a way that it is truly extendable.

The Software Factory components

The EJB architecture contains a number of components built to ease EJB implementations using the Software Factory architecture. While it is outside the scope of this document to thoroughly describe all components, there is a few worth mentioning.

Troubleshooting EJB implementations

This section describes common errors and solutions.

Development

The development section describes problems or mistakes related to the development phase. Note that some of the problems or mistakes might show up during runtime.

Overriding deployment descriptors

When regenerating your entities and classes, be aware that the deployment descriptor is overridden as well. The consequence is fatal if the deployment descriptor contains finder logic, because the finder methods will stop working properly.

Deployment descriptor finder methods before regenerating:

```
"findAll()" "(= 1 1)"
"findBySSN(String pSsn)" "(= mSocialSecurityNumber $pSsn)"
"findByAssociateCode(String pCode)" "(= mCode $pCode)"
"findByUserName(String pUserName)" "(= mUserName $pUserName)"
```

and after:

```
"findAll()" "(= 1 1)"
"findBySSN(String pSsn)" "(= 1 1)"
"findByAssociateCode(String pCode)" "(= 1 1)"
"findByUserName(String pUserName)" "(= 1 1)"
```

It is obvious that all methods except for the findAll() will stop working properly and unless the database only has 1 record, it is unlikely that the finder methods will return the correct entity.

Logic

Dirty reads (unable to read data created in the same transaction)

The problem described in this section occurs when using container managed persistence and transactions with WebLogic. It is not known if any other EJB servers have the same problems.

Reading entities that has been created in the same transaction will not be found by the findermethods. Example

```
lCustomer = lCustomerHome.create();
lCustomer.setCustomerCode("JDO");
lCustomer.setName("John Doe");

lEnumeration = lCustomerHome.findByCustomerCode("JDO");
```

```
while (lEnumeration.hasMoreElements())
{
  lCustomer = (Customer) lEnumeration.nextElement();
  System.out.println("Customer found: " + lCustomer.getName());
}
```

The code shown above will *not* retrieve John Doe the first time it is run. However, it will show up next time the application server is called. It looks like we just have to leave the WebLogic for a split second and our changes will go through.

This ghostly behavior is otherwise known as “dirty reads” or “uncommitted reads”. This is not something that is particular for EJB – In fact, all Client/Server and n-tier applications needs to address the dirty read problem. However, the WebLogic/EJB implementation is slightly different from regular Client/Server implementations, because changes will be stored internally and not applied to the database before the transaction ends. Because the changes are stored internally, we will not be able to use the finder methods to locate the newly created beans.

One way to solve the problem is by changing the transaction type from container managed to bean managed. This means that we manually have to start and stop the transactions.

Change the transactionAttribute to TX_BEAN_MANAGED in the deployment descriptor.

```
       (controlDescriptors
              (DEFAULT
                     isolationLevel             TRANSACTION_SERIALIZABLE
                     transactionAttribute       TX_BEAN_MANAGED
                     runAsMode                  CLIENT_IDENTITY
              ); end DEFAULT
       ); end controlDescriptors
```

In the source code file, import the following classes:

```
import javax.jts.UserTransaction;
import javax.ejb.EJBContext;
```

Add the following code lines in the commit data method:

```
public void commitData()
    throws RemoteException
{
  EJBContext ec = null;
  UserTransaction ut =  null;

  ..

  // start the transaction
  ec = getEJBContext();
  ut = ec.getUserTransaction();

  try
  {
    ut.begin();
```

```
    // Put update code here
    ..

    // End transaction
    ut.commit();
  }
  catch (Exception ex)
  {
    ut.rollback();
    ..
  }

  ..
}
```

Performance degradation due to remote calls

An important feature of EJB is the ability to distribute the application over a number of servers. This neat feature, however, also impacts the application performance. In a distributed environment we cannot know where the bean is located. It can be on the same box and in the same VM, or, it can be on the other side of the globe. The developer cannot know where the bean is instantiated and should assume that all calls potentially could go over the wire.

Performance degradation in loops

Any kind of loop is a potential system killer as the loop count can be hard or even impossible to predict. Always move as much code as possible outside the loop.

Example:

```
for (int i = 0; i < lPhasesEKs.length; i++)
{
  for (int j = 0; j < lBuildingEKs.length; j++)
  {
    ..
    try
    {
      ..
      lXlateEK = lAnyLocationSetUpB.getHousingType((AnyLocationEK) lBuildingEK);

      XlateB lXlateB = (XlateB) AASystem.resolveInterface(XlateB.class) ;
      String lBuildingType =  lXlateB.getDescription(lXlateEK);
      ..
    } catch (Exception ex)
    {
      ..
    }
  }
}
```

The AASystem.resolveInterface(XlateB.class) lookup is done (lPhasesEK * lBuildingsEK) times. Move the resolveInterface outside the main loop, and keep the actual data lookup (lXlateB.getDescription) inside the loop. This will improve the overall speed.

```
XlateB lXlateB = (XlateB) AASystem.resolveInterface(XlateB.class) ;

for (int i = 0; i < lPhasesEKs.length; i++)
{
  for (int j = 0; j < lBuildingEKs.length; j++)
  {
    ..
    try
    {
      ..
      lXlateEK = lAnyLocationSetUpB.getHousingType((AnyLocationEK) lBuildingEK);
      String lBuildingType =  lXlateB.getDescription(lXlateEK);
      ..
    } catch (Exception ex)
    {
      ..
    }
  }
}
```

Exception handling

A number of rules apply when dealing with exceptions:

- Handle only exceptions that can be handled
- Do not handle exceptions that can’t be handled
- Always display user friendly messages to the user
- Avoid dying exceptions

“Handling” implies various operations depending on the context. Handling an exception can include writing debug information to a log file, showing an error message to the user or throwing the exception again.

It is important to watch out for dying exceptions. A dying exception refers to an exception that is caught but no action is taken. Take a look at the example below – no messages will be shown or captured if the customer load failed.

If we call getCustomerName and get a blank name back, we cannot determine if determine if the customer name was not found, if there was a problem loading the entity bean or if the customer name actually is blank.

```
public void dyingException
{
  public void getCustomerName()
  {
    String lResult = “”;

    try
    {
    }
    catch (Exception ex)
    {
      // Ignore exceptions
    }
```

```
    return lResult;
  }
}
```

In this other example, the exception is caught, but is not handled correctly. If an exception is thrown (ie. if one of the parameters are null) the exception will be caught but the method will return true.

```
public boolean doTypesMatch (Class [] c, Object [] o)       {
  int i;

  try {
    for (i = 0; i < c.length; i++)    {
      if (!c[i].isInstance (o[i])) {
        return false;
      }
    }
  }
  catch (Exception e) {
    System.out.println ("Exception checking types matching " + e);
  }

  return true;
}
```

The exception must at least be logged. If we don't log it, helpdesk will have a very hard time explaining how come no customer names are found and no error messages are being displayed (System.out.println is not the best way to do this, see the discussion in the System.out.println section).

Below is another example of Exception handling. In this example we let the exception die, because we really don't care about the FinderException. All we care about is that lCustomerEntity is null if no Customer was found. If an FinderException is thrown, lCustomerEntity will still be null and we can ignore the exception and look only at lCustomerEntity only.

```
public void DyingExceptionExample
{
  public void test()
  {
    CustomerEntity lCustomerEntity = null;
    ..
    try
    {
      ..
    }
    catch (FinderException ex)
    {
      // No handling implemented, we can ignore the finderexception because
      // lCustomerEntity will be null. If the exception was not thrown,
      // lCustomerEntity will be != null.
    }

    ..

    if (lCustomerEntity != null)
    {
      ..
    }
```

```
  }
}
```

We can improve the readability by explicitly setting lCustomerEntity to null if the finderException is thrown.

```
public void BetterDyingExample
{
  public void test()
  {
    CustomerEntity lCustomerEntity = null;
    ..
    try
    {
      ..
    }
    catch (FinderException ex)
    {
      // Set lCustomerEntity to null. That will enable the rest of the code to only
      // have to check lCustomerEntity and nothing else.

      lCustomerEntity == null;
    }

    ..

    if (lCustomerEntity != null)
    {
      ..
    }
  }
}
```

Writing useful error messages

When writing debug information, it is important not only to write that something went wrong, but also why it went wrong. An error message such as "Internal error. Cannot write data" is not helpful for the user nor is it helpful for the developer. A well-designed system will present one type of error message to the user, while a more thorough description is logged in the system log files.

Not-so-good message:

User message: **"Unable to write data"**
System error message: **"Exception while saving data"**

Notice in the above example that none of the error messages indicate where the error occurred, nor do they indicate what can be done to correct the error.

Here's a better version:

User message: **"Unable to save the customer information. The customer order number could not be found. Please re-try the operation and contact global customer support at 1-888-ITBROKE if you get this error again. (Error code #1401)".**

System error message: **"Exception 1401 while saving customer information in CustomerEntity.saveData(). Customer = com.andersen.CustomerEntity@4354, Connection = java.sql.JDBCConnection@12, OrderID = 0, OrderEntity =**

Stacktrace: "

Exception java.lang.NullPointerException at …"

Managing EJB projects

Planning

Development Environment

The development environment should be established before development begins. A well-designed development environment will improve the development speed while a poorly designed development environment can reduce the development speed.

There are three rules in setting up development environments

- Keep it simple
- Keep it consistent
- Keep it identical

Keeping it simple should not need any explanation. If it's simple it's more likely to work. But keeping it simple also means "make it simple when needed". Avoid repetitive keystrokes and processes. Make scripts to compile, deploy and run instead of typing the same commands time after time.

A consistent development environment is an environment that enables logically thinking. We might decide to install all programs under D:\Program Files\Developer, to make it easy for the developer to find the development tools used. If some applications is installed on C: and others on D: our environment is no longer consistent. Install all workstation programs on the same drive, preferably in the same root directory.

Identical environments eases workstation setup, team development and sharing across workstations. If the entire team have the same files installed in the same directories, it is possible to share scripts, property files and compiled modules.

The shared library environment

The most practical environment model is the shared-library model. In this model all developers share a common set of library files. In the Java world, the library files equals the compiled class files.

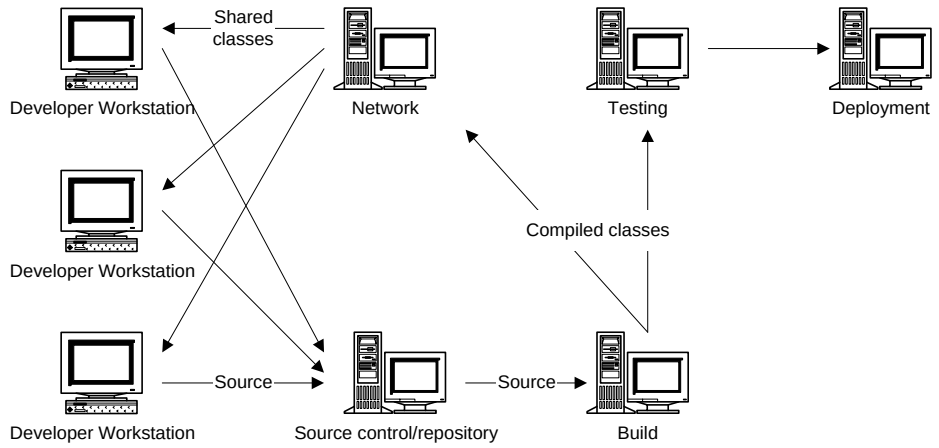
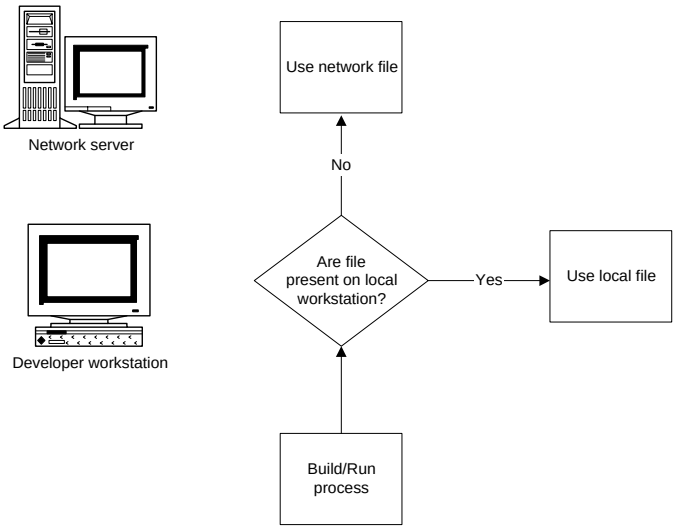


Figure 8 Shared library environment

The shared library is compiled on a frequent basis, daily or hourly or whenever it is most appropriate. All source code changes is compiled on the build computer and will be deployed to the shared library if compiled without errors.

During the development phase, each developer will have a local copy of all source files that is being changed. The source code is compiled and deployed locally. As soon as the change has been implemented the source code is signed back into the repository and is deleted from the local drive.

All workstations will setup so that search will be performed on the local drive before searching on the network drive. Whether it's a class file or a property file, the local drive will always be searched first during build and run. Only if the file can't be found locally will search continue on the network.



The shared library model has two distinct advantages:

- The developers are shielded from potentially bad code that has been signed into source code. Only compiled, workable code will go into the shared library.
- Only local source files has to be compiled. This will reduce compiling time on the local workstation.

The drawback of using shared library is the delay between implementing changes and having the changes propagated to the shared library. Enabling the developers to build on request can minimize the delay.

An alternative to the suggested environment is the working copy environment. Keeping a working copy of the entire network repository locally can reduce network latency and network dependency. Performance is usually higher compared to the network-based environment.

In the working copy model all source files exists locally. During sign-in and sign-out the entire network repository is copied to the developer workstation. All files, except files that have been signed-out, are marked read-only in order to avoid modifying files that haven't been signed out.

The drawback of using the working copy model is that it is possible to modify local files (by resetting the read-only attribute) and that the local version never can be guaranteed to reflect the latest changes – except immediately after local files have been refreshed from the network.

Configuring the shared environment

Because the class path always is searched from left to right, we can overload classes and property files on the network with local classes and property files. The local directory structure effectively becomes a copy of the network directory structure.

The minimal class path should have the following entries.

Directory	Description
.	Current directory
[local]\properties	Local property files
[local]\source	Local source files
[local]\classes	Local compiled classes
[server]\properties	Global property files
[server]\classes	Global compiled classes (daily build).
[server]\libs\SoftwareFactory.jar	Software Factory Code Generator
[server]\libs\EJBModules.jar	Software Factory EJB runtime modules
[jdk]\lib\classes.zip	Java classes

If using C: as the local drive,H: as the network drive and \dev as the development directory, the class path will look like this:

```
Classpath=.;c:\dev\properties;c:\dev\source;c:\dev\classes;h:\dev\properties;h:\de
v\classes;h:\dev\libs\SoftwareFactory.jar;h:\dev\libs\EJBModules.jar;c:\jdk1.1.7\l
ib\classes.zip
```

The actual class path is usually significantly larger than the class path shown above. An example is the following class path that includes WebLogic EJB server classes, the JRun Servlet engine and Swing 1.1 classes:

```
Classpath=.;c:\dev\properties;c:\dev\source;c:\dev\classes;h:\dev\properties;h:\de
v\classes;h:\dev\libs\SoftwareFactory.jar;h:\dev\libs\EJBModules.jar;c:\jdk1.1.7\l
ib\classes.zip;d:\swing-1.1\swingall.jar;d:\dev\libs\weblogic\lib\weblogicaux.jar;
d:\dev\libs\weblogic\classes;d:\jdk1.1.7\lib\classes.zip;d:\dev\libs\jsdk.jar;d:\d
ev\libs\jrun.jar;d:\dev\libs\servlet.jar
```

> **Important** Packages and .jar files can contain classes that conflict with other classes. An example of this is weblogicaux.jar in the class path shown above. This jar file contains Sun's swing classes, but in a different version than swing 1.1. To avoid the problem, Swing 1.1 has to be declared before the weblogicaux.jar declaration.

In Windows NT the class path is modified from **'Start | Settings | Control Panel | System | Environment'**

In Windows 95/98 the class path is modified in the **Autoexec.bat** file, located in the root of the boot drive.

Directory	Description
D:\dev	Framework architecture directory
D:\dev\jdk1.1.7	Sun JDK 1.1.7
D:\dev\jbuilder3	JBuilder 3
D:\dev\swing1-1	Swing extensions 1.1
D:\dev\tools\…	Other required project tools