8. Component Software

Overview

8.1 Component Frameworks: An Introduction

8.2 OSGi Component Framework
  8.2.1 Component Model and Bundles
  8.2.2 OSGi Container and Framework
  8.2.3 Further Features of the OSGi

8.3 Enterprise JavaBeans
  8.3.1 Component Model
  8.3.2 Component Infrastructure

Component frameworks support the composition of software systems from concrete software components.

Software components are well-defined entities of their own that can be provided by other development groups or companies.

In particular, component technology is important to create a market for prefabricated software.

Component frameworks have three main goals:

- Reuse of software components
- Providing the basis for a market of software components
- Role separation

Component frameworks enable to separate the responsibility of developing and managing a software system into five roles:

- component provider,
- framework provider,
- application provider
  (constructs an application from components),
- application deployer (installs an application on a platform),
- system administrator.

Role separation is important for division of labor, specialization, and better competition in the software market.

8.1 Component Frameworks: An Introduction

The “compound object pattern” provides the conceptual basis for component frameworks.

In this context, the term components refers to software components consisting of concrete “program parts and data”.

Components are the units of composition in the context of a component framework.

The goal of component frameworks is the **blackbox** reuse. Inspection of the component’s implementation by the user should be unnecessary.

Composition Techniques

A component framework provides the mechanisms to stick components together.

We can distinguish three kinds of *composition mechanisms:*

- Static *linking* of components, i.e. the components are linked together before they are executed.
- *Dynamic linking* of components, i.e. some components are linked at runtime into an executing process.
- *Connecting* of components that run as parts of different processes.

Component Frameworks

The glossary of ”Component Software: Beyond object-Oriented Programming” by Clemens Szyperski gives an explanation of the term “component framework”:

“A *component framework* is a collection of rules and interfaces (contracts) that govern the interaction of *components* plugged into the framework. A component framework typically enforces some of the more vital rules of interaction by encapsulating the required interaction mechanisms.”

As the general goal is to compose software systems from prefabricated components, a component framework has to give answers to the following three questions:

- Component model: What is a component?
- Component infrastructure: What are the infra-structure and the technical support for deploying and composing components?
- Further features: How are further requirements supported by the component framework?

The *component model* essentially consists of rules specifying

- what kinds of components are supported,
- what constitutes a component,
- and what the properties of components are.

Many of the rules are usually syntactic.
But, there may be as well semantic rules.

Components are described with the help of some languages (e.g. programming or assembly languages).

The *component infrastructure*

- explains how components are composed and deployed
- describes the rules
  - how a component finds other components by name,
  - how components are linked to each other,
  - and how they can communicate
- specifies interfaces that allow to inspect components at runtime
- may provide mechanisms to startup and initialize remote components

- may comprise library code that components need to connect to other components
- may define standardized platform components for composed systems
- may specify tools for composition and deployment

Component frameworks often support further features, like:

- support for persistence,
- security mechanisms.

Different Kinds of Frameworks

- Component frameworks can be independent of requirements of specific application domains.
- They can support general architectural patterns (e.g. the EJB framework primarily supports the construction of software systems according to the three-tier architectural pattern).
- They can be developed for more specific application areas. In that case, the component model and infrastructure are specialized to that area. Often predefined components are provided.

Discussion of „Frameworks“:

The RMI (remote method invocation) in Java is a communication infrastructure.

- It provides library classes, interfaces, and services to connect to remote objects and invoke their methods.
- It can be used to build up a component infrastructure, but it is **not** a component framework
- A component model is missing.

The AWT is a typical program framework.

The subclasses of class Component can be considered as components.

A component infrastructure providing rules and techniques for composition and deployment is missing.

Improvement:
JavaBeans framework building on AWT, which

- defines rules and conventions that specify component interfaces,
- provides an event-based composition mechanism.

8.2 OSGi: A Component Framework for Java

OSGi is a component framework for Java supporting:

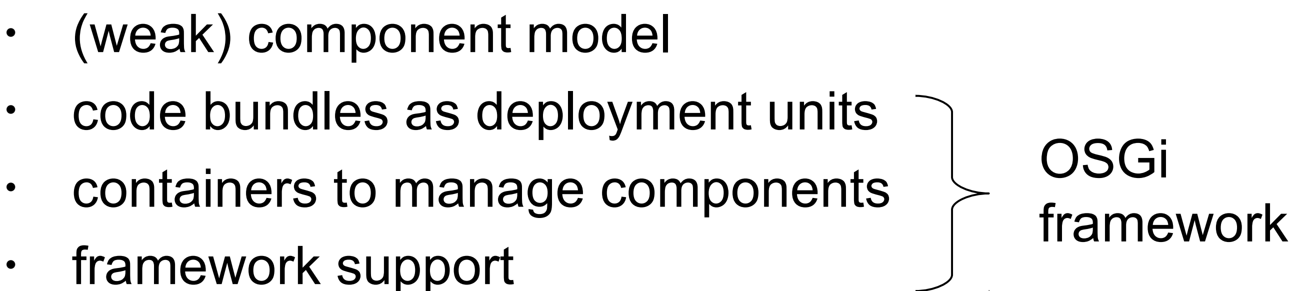
- dynamic installation, management, and runtime control of components (starting, stopping)
- deployment units called *bundles*
- versioning, security aspects, etc.

Here, we give an introduction to OSGi in version 4.1.

OSGi is a framework specification (core specification has 288 pages) with different implementations (we used Eclipse Equinox).

Central concepts:

- (weak) component model
- code bundles as deployment units
- containers to manage components
- framework support



8.2.1 Component Model and Bundles

OSGi components:

- provide services via Java interfaces
- use services of other components
- have an *activator class* for starting/stopping/

managing the component.

The activator's start method

- creates and initializes the component instance(s)
- looks up required services
- registers service listeners
- registers the provided services

The stop method has to realize the corresponding tasks for deregistration and shutdown.

Component developer is responsible for

- connecting components
- multithreading
- transfer objects

(That is why we speak of a *weak* component model.)

Example: (OSGi component)

The service interface:

```
package randomnumberservice;

public interface RandomNumberService {
   // Returns a pseudorandom integer
   public int getRandomNumber();
}
```

Example: (OSGi component, continued)

The service implementation:

```
package randomnumberservice.impl;

import randomnumberservice.
                 RandomNumberService;
import java.util.Random;

public class RandomNumberServiceImpl
       implements RandomNumberService
{
  Random rand;
  /**
   * Initializes the random number generator
   * generator. The method must be called by
   * the bundle registering the service.
   */
  void initService() {
    rand = new Random();
  }

  public int getRandomNumber(){
    return rand.nextInt();
  }
}
```

Example: (OSGi component, continued)

The activator class:

```
package randomnumberservice.impl;

import java.util.Properties;
import org.osgi.framework.*;
import randomnumberservice.*;

public class Activator
       implements BundleActivator
{
  private ServiceRegistration reg = null;

  public void start(BundleContext bctxt)
               throws Exception
  {
    System.out.println(
      "randomNumberService starting");

    // Create & initialize service object
    RandomNumberServiceImpl rnsi =
      new RandomNumberServiceImpl();
    rnsi.initService();
    System.out.println(
      "randomNumberService initialized");

    // register service (see next slide)
```

```
    // register service
    reg = bctxt.registerService(
      RandomNumberService.class.getName(),
      rnsi, new Properties());

    System.out.println(
       "randomNumberService registered");
  }

  public void stop(BundleContext bctxt)
               throws Exception
  {
    System.out.println(
      "randomNumberService stopping");
    if (reg != null)  reg.unregister();
  }
}
```

Remark:

1. The above example shows the basic elements of components.

2. In the activator, the component can create its own thread of control (active component).

Example: (Client using a component)

Here is a simple client using the random number service:

```
package client.impl;
  import org.osgi.framework.*;
  import randomnumberservice.*;

public class Activator
       implements BundleActivator
{
  private ServiceReference ref = null;

  public void start(BundleContext bctxt)
              throws Exception
  {
    System.out.println("client: starting");
    // get ServiceReference object
    ref = bctxt.getServiceReference(
    "randomnumberservice.RandomNumberService");

    // test if this service exists (see next)
```

```
  // test if this service exists
    if (ref != null) {
      // get service object
      RandomNumberService rns =
        (RandomNumberService)
                       bctxt.getService(ref);
        // use the service
        System.out.println("random number:"
                      + rns.getRandomNumber());
  } }

  public void stop(BundleContext bctxt)
               throws Exception
  { // release the service
    bctxt.ungetService(ref);
    System.out.println(
               "client : service released");
    System.out.println("client: stopping");
  }
} // end of client
```

Remark:

1. A client can of course obtain and release services at any point in time.

2. A component requiring a service has to obtain it as shown above.

3. Component and application developer have to make sure that services are available when needed.

OSGi bundles:

- units of deployment
- realized as jar-archives
- consist of
  - manifest file
  - interfaces of provided services
  - implementation of services
  - activator class

(Bundles are similar to .NET assemblies.)

Bundles have two roles:

- manage codebase (import/export packages)
- provide services

In OSGi, a bundle can perform both roles at a time or it can perform only one of the roles.

Example: (Bundle manifest)

A simple bundle manifest:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName:    randomNumberService
Bundle-Activator:
         randomnumberservice.impl.Activator
Import-Package:
         org.osgi.framework;version="1.3.0"
Export-Package:         randomnumberservice
```

8.2.2 OSGi Container and Framework

According to the OSGi terminology, the OSGi framework includes

- the container, that is the program managing the bundles, and
- the API for accessing the container and its services from bundles.

The container supports dynamic linking and provides the following bundle states:

INSTALLED – Bundle has been successfully installed.

RESOLVED – All Java classes that the bundle needs are available. Bundle is either ready to be started/has been stopped.

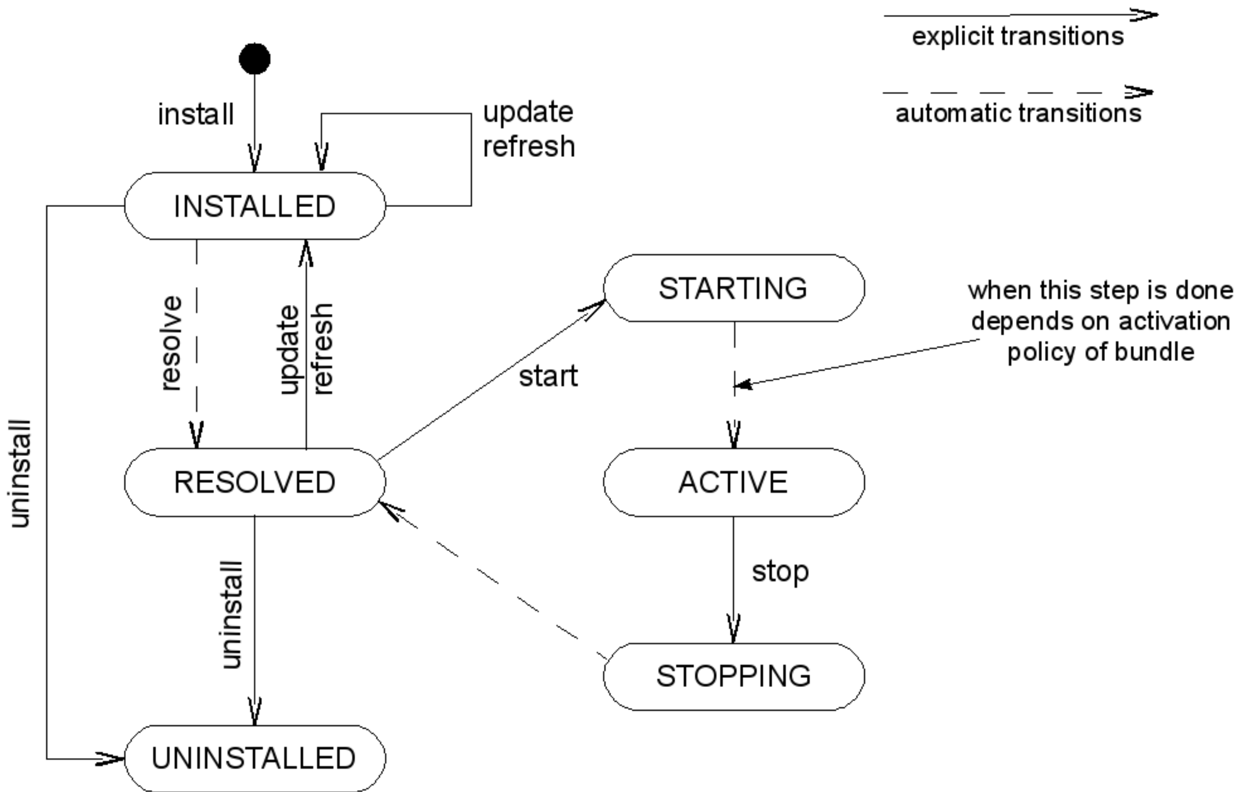
STARTING – The bundle is being started, the BundleActivator. start method will be called, and this method has not yet returned. When the bundle has a lazy activation policy (specified in the Manifest), the bundle will remain in the STARTING state until the bundle is first used.

ACTIVE – The bundle has been successfully activated and is running; its Bundle Activator start method has returned.

STOPPING – The bundle is being stopped. The BundleActivator.stop method has been called but yet returned.

UNINSTALLED – The bundle has been uninstalled. It cannot move into another state.

Transitions between bundle states:



Transitions can be triggered

- by other components via the framework
- from a shell:

```
$ equinox
osgi> install file:///osgi/plugins/randomNumberService.jar
Bundle id is 2
osgi> install file:///osgi/plugins/client.jar
Bundle id is 3
osgi> start 2
randomNumberService starting
randomNumberService initialized
randomNumberService registered
osgi> start 3
client : starting
client : got service
client : generating a random number: -1065829765
osgi> stop 3
.........
```

8.2.3 Further Features of OSGi

OSGi provides many more features.
Important examples are support for

- Event infrastructure:
  The framework supports events to communicate state changes of bundles to other bundles.
- Startup and shut down:
  There is a detailed procedure for shutting down and starting containers. In particular, all started bundles at shut down time are automatically restarted at start up.
- Security:
  An optional security layer supports authentication of bundles (based on the Java security architecture)
- Versioning:
  OSGi allows the use of different versions of bundles and packages. Version numbers and ranges can be used with bundle imports.

8.3 Enterprise JavaBeans

The Enterprise JavaBeans framework is a component framework developed by Sun Microsystems.
It supports the realization of application servers according to the three tier architectural pattern.

Explanation: (Application server)

An ***application server*** is a software system that provides services for (remote) clients.
The persistent data involved in such services is usually managed by databases.

The tasks of an application sever are:

- Accepting and maintaining the connection to clients (authorization and security mechanisms, control over the client sessions)
- Reading and writing data from and to the database
- Transaction mechanisms
- The core functionality of the supported services

In the following, we describe

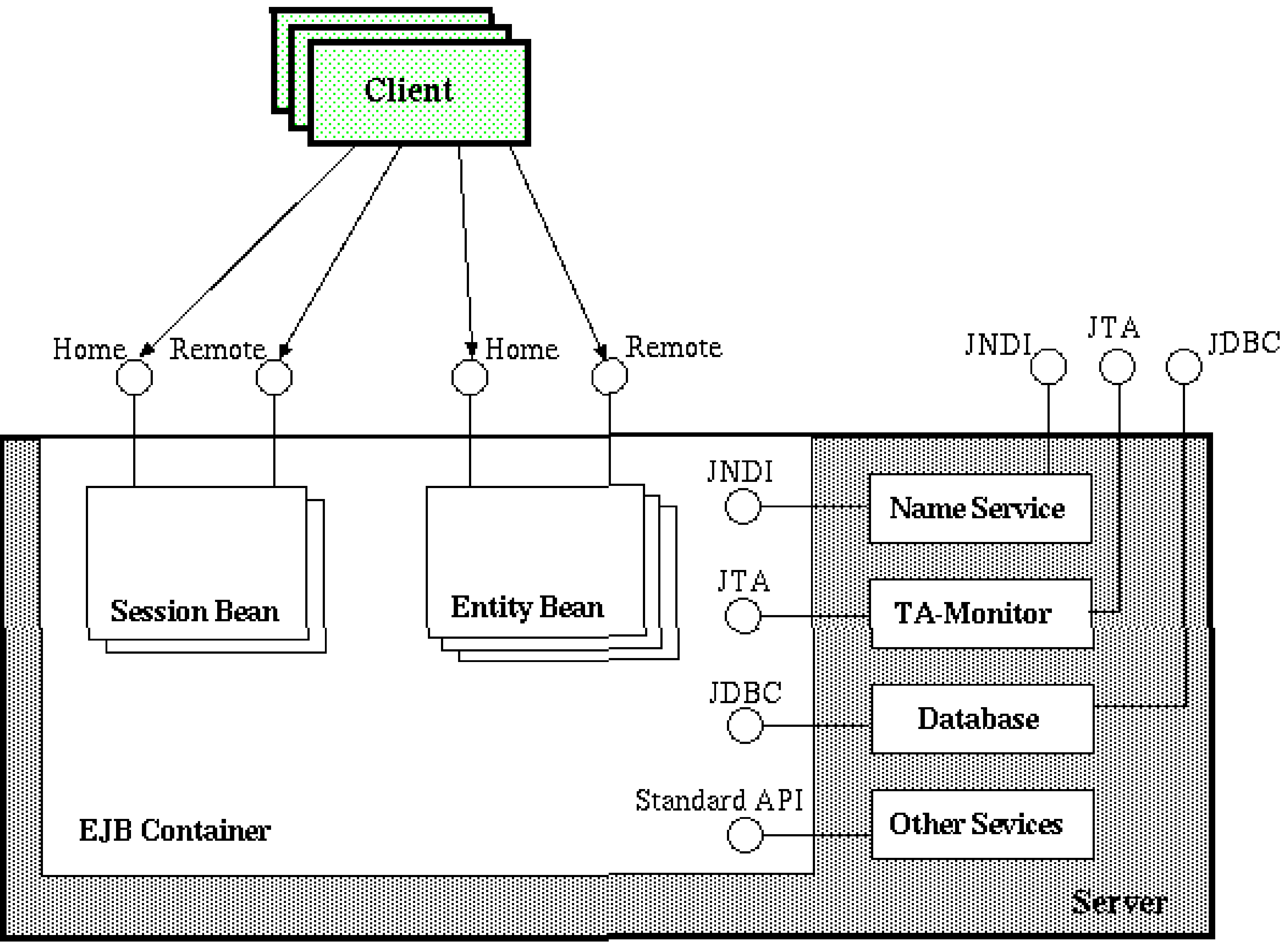
- the component model, i.e. what beans are and how they are described, and
- the infrastructure to deploy and work with beans.

Further reading:

Enterprise JavaBeans Specification

Remarks:

- EJB shows (like OSGi) that component frameworks do not provide the implementation of components to be used in the constructed systems (in contrast to program frameworks).
- A number of other component frameworks supporting the construction of application servers are available (e.g. the Spring Framework).


Client
Home
Remote
Home
Remote
JNDI
JTA
JDBC
Session Bean
Entity Bean
JNDI
JTA
JDBC
Standard API
Name Service
TA-Monitor
Database
Other Sevices
EJB Container
Server

8.3.1 Component Model

The Enterprise JavaBeans are the components from which a so-called application assembler constructs a server-side application.

The resulting application is deployed in an EJB container. After deployment, it is available to clients.

The application and thus the beans can use the mechanisms of the underlying container like:

- transaction mechanisms,
- control for persistence,
- security services,
- etc.

Bean Species and Parts

The beans are the units of composition in the EJB framework, i.e. the components. We speak of

- *bean instances:* These are the runtime objects. A bean instance might consist of more than one object.
- *bean components:* This is the software implementing their functionality. A bean component comprises more than one Java class.
- There are two different kinds of beans: ***Entity*** beans and ***session*** beans.

Entity beans essentially represent the data objects handled by the system (e.g. bank accounts, products, assurance policies, orders, or employees).

An entity bean instance usually corresponds to what is called an entity in the literature about entity-relationship modeling and data base systems.

Entity bean instances are usually persistent.
The programmer of the bean can choose between

- bean-managed persistence: persistence mechanisms implemented by programmer;
- container-managed persistence: persistence is managed automatically by the container.

An entity bean instance can be accessed by different clients communicating with the application sever.
The instance is logically identified by a so-called ***primary key***.

Session beans are used to model tasks, processes, or behavior of the application server.

A session bean corresponds to a service for a client.

A session bean instance is a private ressource of a client.

The framework distinguishes between

- ***stateless*** session beans and
- ***stateful session*** beans, which have a state that is maintained between method invocations.
  The state is not persistent.

A ***bean component*** consists of four or five parts:

- the remote interface,
- the home interface,
- the primary key class (only for entity beans),
- the bean class, and
- the deployment descriptor.

We will explain these parts in turn by a simple example of an entity bean representing a bank account.

The Remote Interface

The ***remote interface*** defines the methods that the entity bean provides to the client.
It reflects the functionality of the bean.

The bank account bean of our example allows one

- to ask for the number of the account,
- to query for the balance,
- and to change the balance by a positive or negative amount.

```
public interface BankAccount
                extends EJBObject {
  String getAccountNumber()
                throws RemoteException;
  float getBalance()
                throws RemoteException;
  void changeBalance(float amount)
                throws RemoteException;
}
```

EJBObject is a supertype of all bean instances.
In order to support remote method invocation, it is a subtype of type Remote.

The Home Interface and Primary Key Class

The ***home interface*** defines the methods concerning the lifecycle of the bean.

The client uses this interface

- to create bean instances,
- to find them,
- and to remove them.

In our example, the home interface defines two methods to create and find an account:

```
public interface BankAccountHome
                 extends EJBHome {
  BankAccount create(String accNo,
                     float initBal)
             throws CreateException,
                    RemoteException;

  BankAccount findByPrimaryKey(
                   BankAccountPK accPK)
             throws FinderException,
                    RemoteException;
}
```

Home interfaces have to extend interface EJBHome which contains a method to delete a bean and, as a subtype of Remote, the functionality for remote method invocation.

The method *findByPrimaryKey* plays a special role in the EJB framework. It has to take a primary key as parameter. Its return type has to be the type of the corresponding remote interface.

Here, we use the account number as primary key:

```
public class BankAccountPK
     implements java.io.Serializable {

  public String accountNumber;

  public BankAccountPK(String accNo){
     accountNumber = accNo; }

  public BankAccountPK() {
     accountNumber = new String(); }

  public int hashCode() {
     return accountNumber.hashCode(); }

  public boolean equals(Object obj) {
     return accountNumber.equals(obj); }

  public String toString() {
     return accountNumber; }
}
```

The Bean Class

The bean class provides implementations for the methods of the home and remote interface. The only exception is the method findByPrimaryKey which is provided in the deployment process.

The bean class must not be a subtype of the home and remote interfaces, i.e. it does not “implement” the interfaces in the sense of Java.

It has to provide implementations for their methods. The connection between the bean class and the interfaces will be established during deployment.

```
public class BankAccountBean
             implements EntityBean {

   public String accountNumber;
   public float accountBalance;
   private EntityContext theContext;

// continued on following slides
```

```
// corresponds to create method of
// home interface: ejb+create
   public BankAccountPK ejbCreate(
            String accNo, float initBal)
         throws CreateException,
                RemoteException {
       accountNumber = accNo;
       accountBalance = initBal;
       return null;
    }

// methods implementing the remote
// interface
   public String getAccountNumber()
         throws RemoteException {
      return accountNumber;
   }
   public float getBalance()
         throws RemoteException {
      return accountBalance;
   }
   public void changeBalance(
                       float amount)
         throws RemoteException {
      accountBalance += amount;
   }
```

```
// the methods of interface
// javax.ejb.EntityBean
   public void setEntityContext(
                     EntityContext ctx)
            throws RemoteException {
      theContext = ctx;
   }
   public void unsetEntityContext()
            throws RemoteException {
      theContext = null;
   }

   public void ejbRemove()
            throws RemoteException,
                   RemoveException {}
   public void ejbActivate()
            throws RemoteException {}
   public void ejbPassivate()
            throws RemoteException {}
   public void ejbLoad()
            throws RemoteException {}
   public void ejbStore()
            throws RemoteException {}
}
```

Bean classes have to implement the interface EntityBean of the EJB framework.
The methods of this interface manage the communication between the bean and the container.

For example the method setEntityContext allows the container to store so-called context information into the bean. This context information is used by the container to manage the identity of bean instances.

The Deployment Descriptor

The ***deployment descriptor***

- describes the different parts of a bean and the runtime properties like e.g. whether persistence is container- or bean-managed,
- allows to combine several beans into one unit,
- describes external dependency (e.g. to databases or external beans),
- provides deployment information (e.g. access rights specifying who has the permission to invoke a method).
- Deployment descriptors are defined in an XML format.

Remark:

EJB framework provides a very practical example of a description language with formal syntax to capture architectural relations and properties.

The deployment descriptors are used to deploy the software systems in a semi-automatic way.

```
<?xml version="1.0" ?>
<!DOCTYPE ejb-jar PUBLIC
"-//Sun Microsystems, Inc.
    //DTD Enterprise JavaBeans 1.2//EN"
"http://java.sun.com/j2eedtds/ejb-jar_1_2.dtd>"

<ejb-jar>
<description>
deployment descriptor of entity bean BankAccount
</description>
```

(continued on next three slides)

```xml
<enterprise-beans>
  <entity>
  <description>
  BankAccount represents a bank account.
  </description>
<!-- name of the bean in JNDI -->
  <ejb-name>BankAccount</ejb-name>
<!-- name of home interface -->
  <home>ejb.samples.accounts.BankAccountHome</home>
<!-- name of remote interface -->
  <remote>ejb.samples.accounts.BankAccount</remote>
<!-- name of bean class -->
  <ejb-class>ejb.samples.accounts.BankAccountBean
  </ejb-class>
<!-- bean uses container-managed persistence -->
  <persistence-type>Container</persistence-type>
<!-- class of primary key -->
  <prim-key-class>ejb.samples.accounts.AccountPK
  </prim-key-class>
```

```
<!-- field in bean corresponding to key -->
  <primkey-field>accountNumber</prim-key-field>
<!-- bean is not reentrant -->
  <reentrant>False</reentrant>
<!-- fields of bean that should be persistent -->
  <cmp-field>
    <field-name>accountNumber</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>accountBalance</field-name>
  </cmp-field>
  </entity>
</enterprise-beans>

<assembly-descriptor>
  <security-role>
    <description>The role of bank employees</description>
    <role-name>banker</role-name>
  </security-role>
```

```xml
<!-- Bank employees may invoke all methods -->
  <method-permission>
    <role-name>banker</role-name>
    <method>
      <ejb-name>BankAccount</ejb-name>
      <method-name>*</method-name>
    </method>
  </method-permission>
<!-- transactional behavior of method changeBalance -->
  <container-transaction>
  <!-- method is only invoked as part of transaction -->
    <method>
      <ejb-name>BankAccount</ejb-name>
      <method-name>changeBalance</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
</ejb-jar>
```

Further Aspects

In summary, the component model of the EJB framework is based on

- Java interfaces and classes
- and an XML document describing the relationship and properties of the parts.

I.e. it is a programming language dependent model.

The EJB specification contains a fairly large number of conventions and rules which

- make it possible that the parts can work together seamlessly,
- allow for the communication with the underlying container,
- make the beans to a large extent independent of the special implementation of the container.

Starting with version 3.0, EJB supports Java annotations to simplify the coding and to realize automated composition by *dependency injection*.

In the following, we provide some more rule examples:

- A bean must not return the implicit parameter “this” as result of a method or pass it as a parameter.
- Static variables are forbidden in bean classes.
- Use of threads and synchronization mechanisms is not allowed.
- Use of GUI facilities as well as input- and output-operations are forbidden.
- Use of introspection and reflection must be avoided.
- Beans should not influence class loading and security management.

The basic reason underlying these restrictions is to create a well-defined interface between the components and the container.

This interface has to take behavioral and security aspects into account. E.g.

- deadlocks have to be avoided,
- the access rights of beans have to be restricted so that they cannot manipulate the data of other beans.

8.3.2 Component Infrastructure

The backbone of the EJB component infrastructure is the ***EJB container***.
A bean instance can only exist within a container.
The container is the runtime environment for bean instances and provides the “necessary services”, in particular:

- a naming service,
- a transaction monitor,
- access to a database,
- and Java Mail service

Managing Beans

The major task of the EJB container is to control the life cycle of bean instances.

It creates and removes instances.
In addition, it is responsible for an efficient management of bean instances:

- Many containers provide mechanisms to passivate and reactivate instances by temporarily storing them on disk.
- Pooling of instances allows to reuse an instance thereby avoiding removing an instance and afterwards creating an equal instance again.

- Support for load balancing between different containers: If one container is very busy, use another less frequented container.
- Mechanism to access beans by name. Containers use the Java Naming and Directory Interface which is an extension to the naming mechanism underlying RMI.

Persistence, Transactions, Security

EJB containers have to support a mechanism that is called container-managed persistence.

An entity bean with container-managed persistence relies on the container to perform persistent data access on behalf of the bean instances.

The container has to

- transfer the data between the bean instance and the underlying database,
- implement the creation, removal, and lookup of the entity object in database,
- manage the mapping between primary key and EJBObject.

Transactions ensure data consistency, taking into account failure recovery and multi-user programming.

The EJB framework supports distributed transactions: An application can automatically update data in multiple databases which may be distributed across multiple sites.

In case of container-managed transaction demarcation, the container demarcates transactions according to instructions provided by the application assembler in the deployment descriptor.

These instructions describe e.g. whether the container should run a method

- in a client's transaction,
- in no transaction,
- or in a new transaction that the container has to start.

In order to support such mechanisms, the EJB framework comprises a transaction model.

The EJB framework provides several mechanisms to secure the applications assembled from the beans. The container provides the implementation of a security infrastructure that can be controlled by so-called security policies.

It is important to have a clear separation between

- application code, i.e. beans,
- and security code, i.e. the security infrastructure of the container and the security policies.

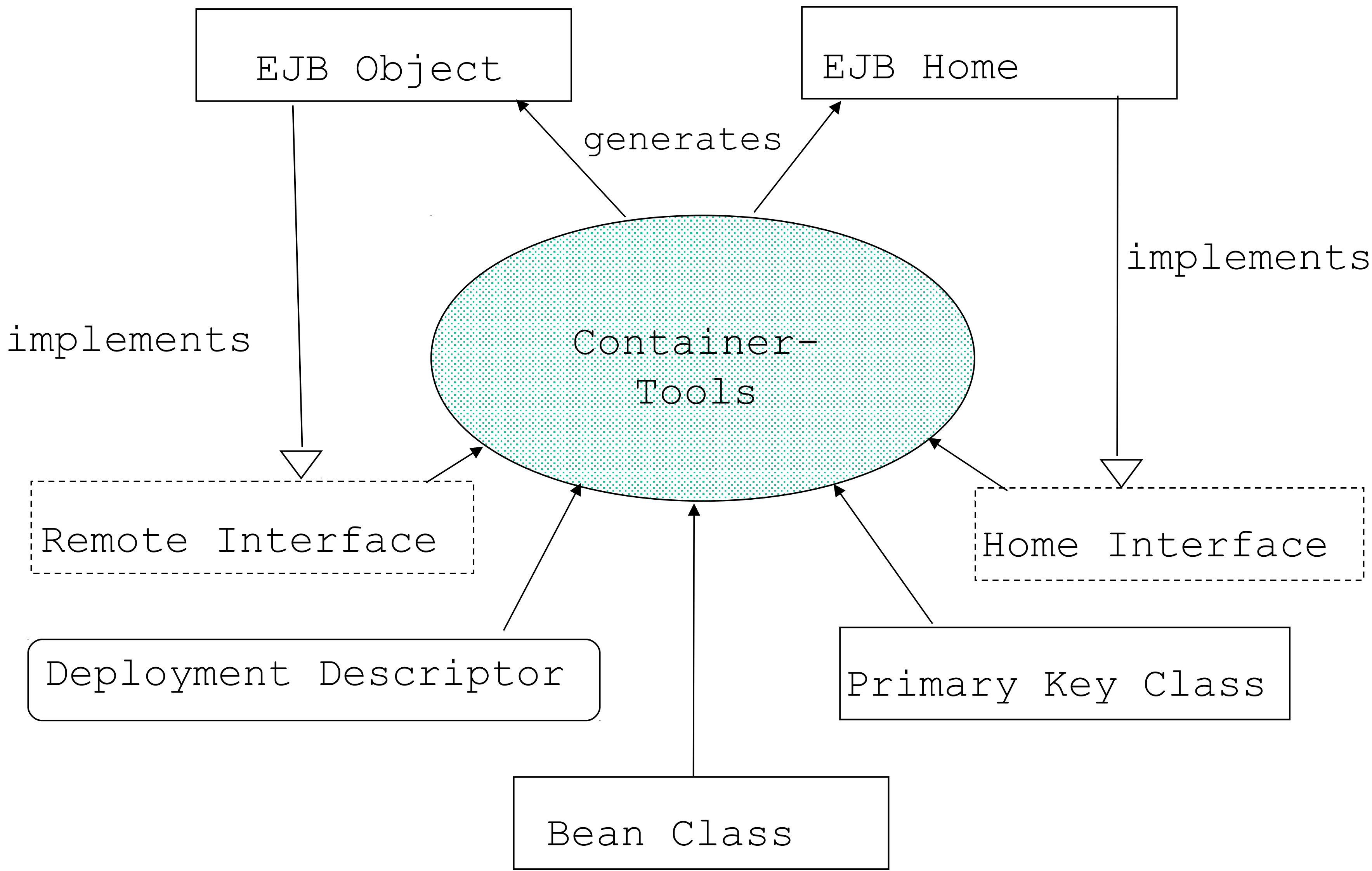
These parts are connected only at deployment time.

Deployment of Beans

We consider the question of how beans are deployed in containers.
The container developers have to provide tools for the deployment of beans.

For each bean X, the tools take the bean sources as input and generate two classes:

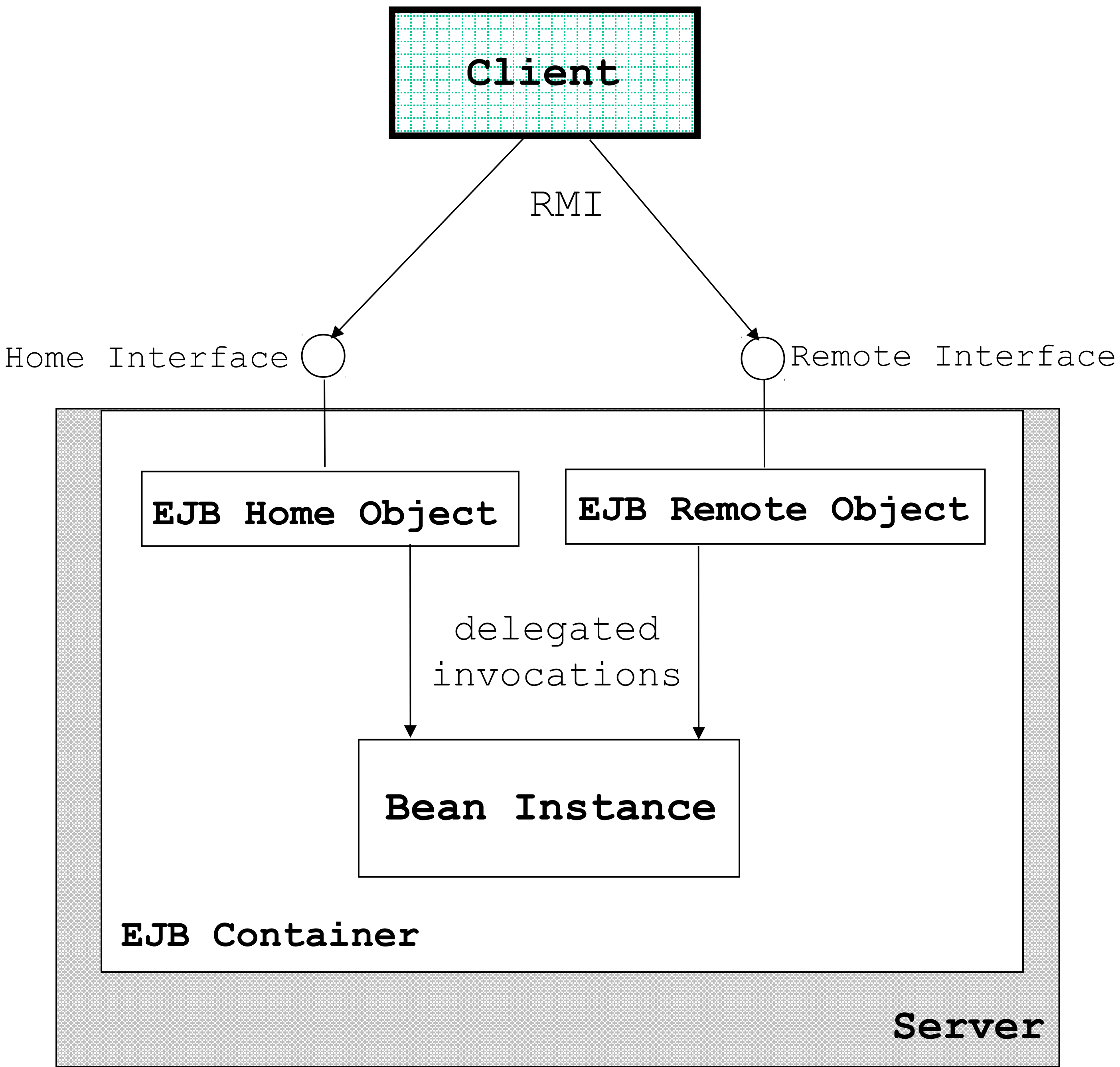
- a class XHome implementing the home interface (EJB home objects)
- and a class XObject implementing the remote interface (EJB remote objects).


EJB Object
generates
EJB Home
implements
implements
Container-
Tools
Remote Interface
Home Interface
Deployment Descriptor
Primary Key Class
Bean Class

The EJB home and remote objects

- encapsulate the bean instance:
  - A client can only access a bean by invoking methods of the home and remote objects.
  - These objects delegate the client's method invocations to the bean instance.

- perform operations realizing important tasks of the container:
  - ressource management,
  - security mechanisms,
  - transactions handling,
  - persistence management.


Client
RMI
Home Interface
Remote Interface
EJB Home Object
EJB Remote Object
delegated
invocations
Bean Instance
EJB Container
Server

It is important that the EJB framework supports such a delegation and encapsulation technique:

- This way, the container has control over bean instances and can prevent incorrect or malicious access to beans.
- It is the basis to relate the component infrastructure to the components.
- It frees the bean developers to a large extent from taking infrastructure issues into account.

A bean instance is always accessed through its home and remote object. This also holds for the communication among beans.

Using Beans

To illustrate the usage of beans let us consider the example of the bank account.

A client wishes to access an instance of the BankAccount-bean.

He has to know the name by which the bean is made accessible in the container.

Usually, this is the name of the bean as given in the deployment descriptor. In our example, the name was “BankAccount”, and we assume that the bean is accessible under the name “java:comp/env/ejb/BankAccount”.

The following fragment shows the essential steps to access and use a bean:

```
String BANK_ACCOUNT =
        "java:comp/env/ejb/BankAccount";
InitialContext ctx = new InitialContext();
Object o = ctx.lookup(BANK_ACCOUNT);
BankAccountHome bh = (BankAccountHome)
           PortableRemoteObject.narrow(
                o,BankAccountHome.class);
BankAccount ba = bh.create("2345735",0.0);
ba.changeBalance(32.00);
```

The creation of a new empty bank account leads to the creation of

- a bank account instance,
- an entity in the underlying database,
- a primary key object, associated with the bean instance (access with method `getPrimaryKey`)

In summary, we can see that the use of beans is very similar to the use of remote objects. The management of beans by the container is hidden to the client.