J2EE and Web Services Overview

InferData

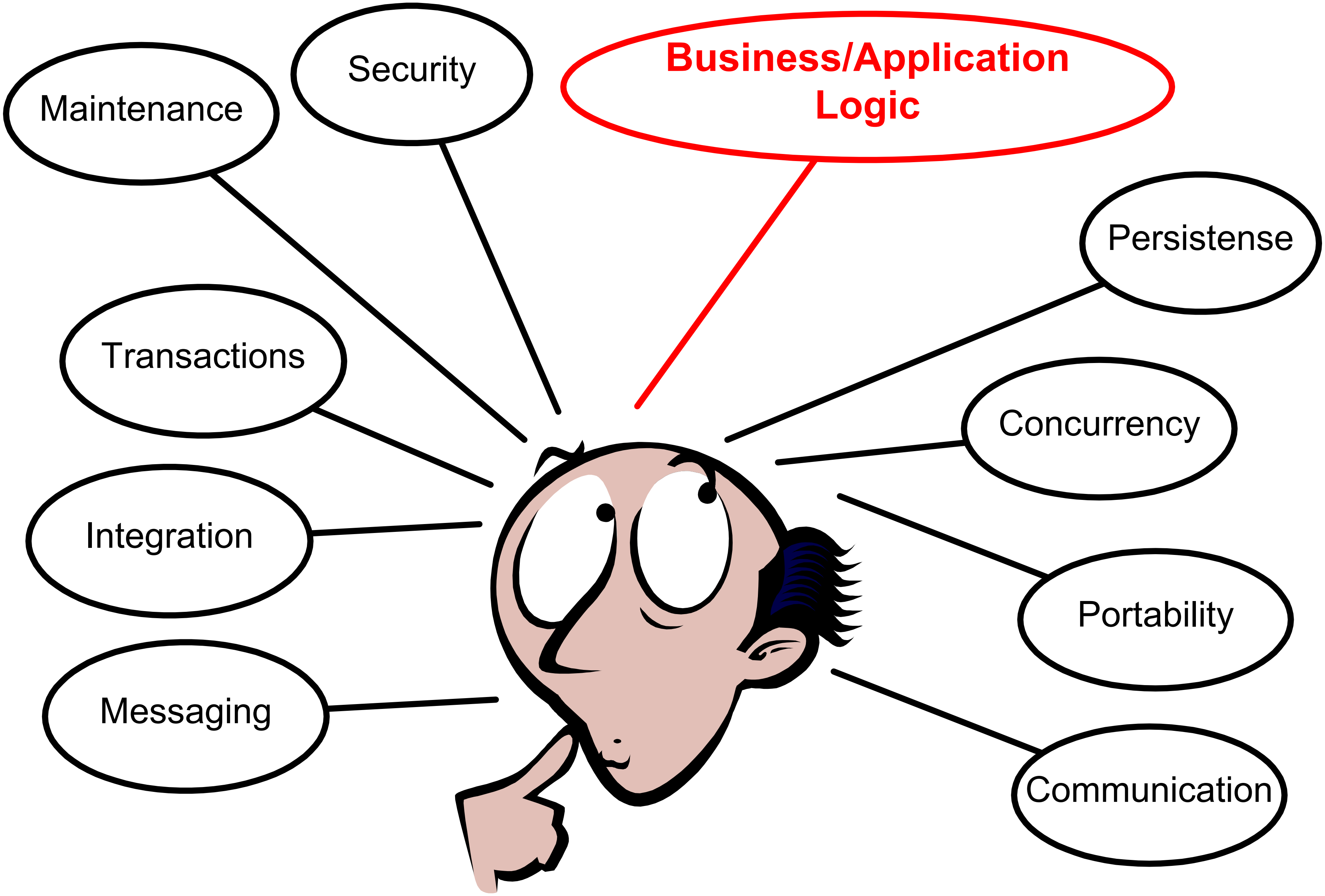
8200 NorthMoPac Expy. Suite 250
Austin, Texas 78759
Phone: (512) 306-8225

http://www.inferdata.com

Version 3.3

J2EE Architecture

Issues in Enterprise Application Development



Why J2EE?

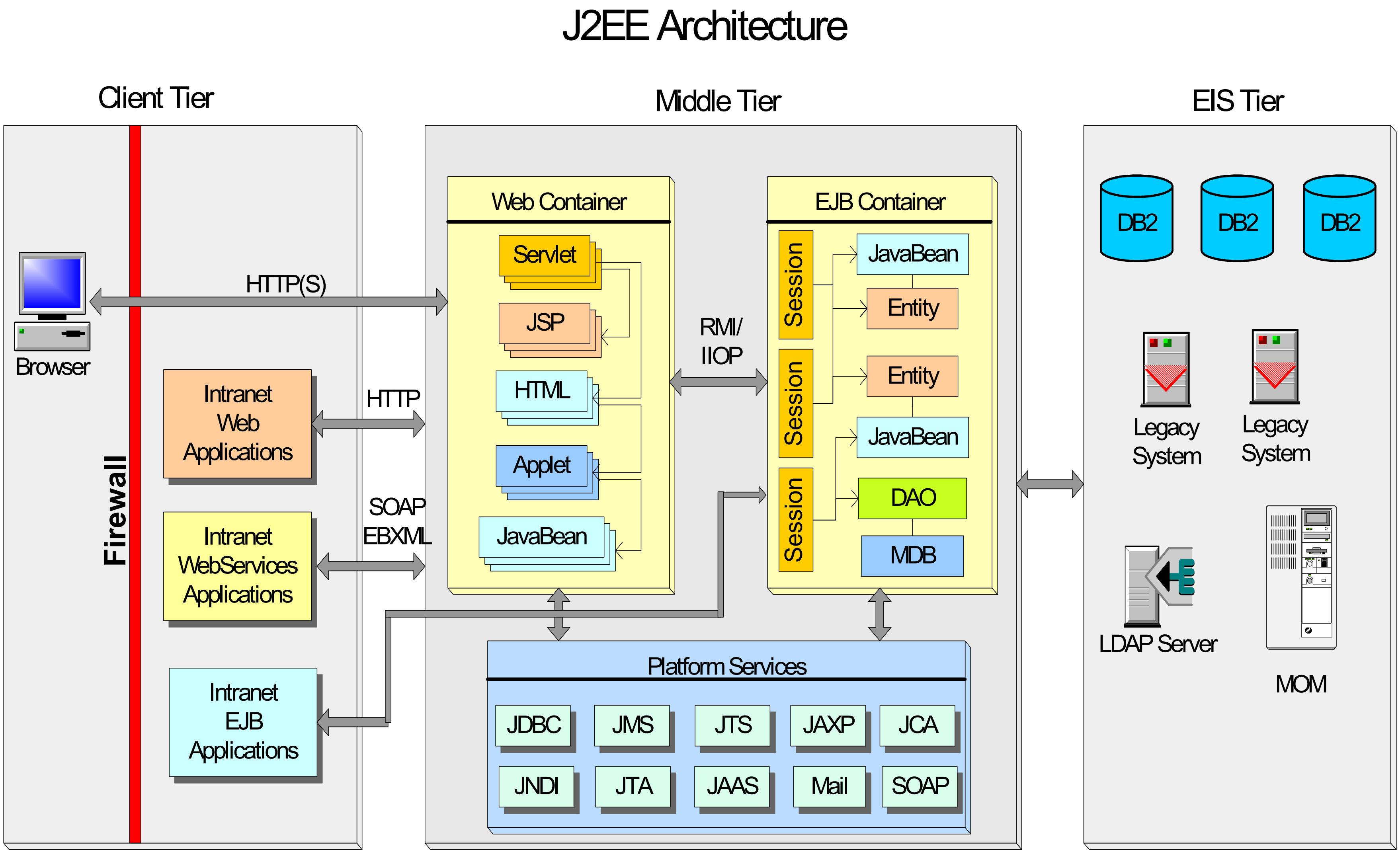
- The Java 2, Enterprise Edition (J2EE) platform enables the development of systems that are:
  - Transactional (Even Distributed!!)
  - Secure
  - Persistence-aware
  - Scalable
  - Distributed
  - Web-enabled
  - Component-based
  - Message oriented
  - **Portable**
  - Maintainable
  - Extensible
  - Inter-operable with legacy systems

What is J2EE?

- J2EE is introduced to reduce the cost and complexity of developing n-tiered enterprise applications.

- J2EE defines a complete platform that guarantees services in such areas as:
  - Security
  - Transaction Management
  - Resource management
  - Naming and directory
  - Messaging
  - Concurrency control
  - Interoperability

Java 2 Platform, Enterprise Edition



- Component based architecture for scalable, transactional, secure applications

Client Tier

- Allows the user to interact and enables presentation of information to the user
- Supports variety of client types:
  - Web Clients – communicate with the web tier using HTTP or HTTPS protocol
    - Web Browsers
    - Java Applets
    - Java Applications
    - Non-Java Clients
- EJB Client – communicates directly with the EJB tier using RMI/IIOP

Web Tier

- Web tier holds Web Applications
- A Web Application is a collection of HTML/XML documents, web components, and other resources in either a directory structure or archived format known as a Web ARchive (WAR) file.
- Web Components refer to servlets and JSP pages
- A Web Container is a runtime environment for Web Applications
  - A Web Container provides Web Components with a naming context and life cycle management. Some may provide more

EJB Tier

- EJB tier is where EJB container resides
- EJB components are deployed into EJB container
- EJB components model application specific business logic
- EJB container provides following services to its components:
  - Security
  - Transaction management
  - Resource management
  - Naming and directory
  - Concurrency control
  - Interoperability

J2EE Applications

- A J2EE application is a collection of software components that are engineered to be distributed across multiple tiers of an N-tier system.

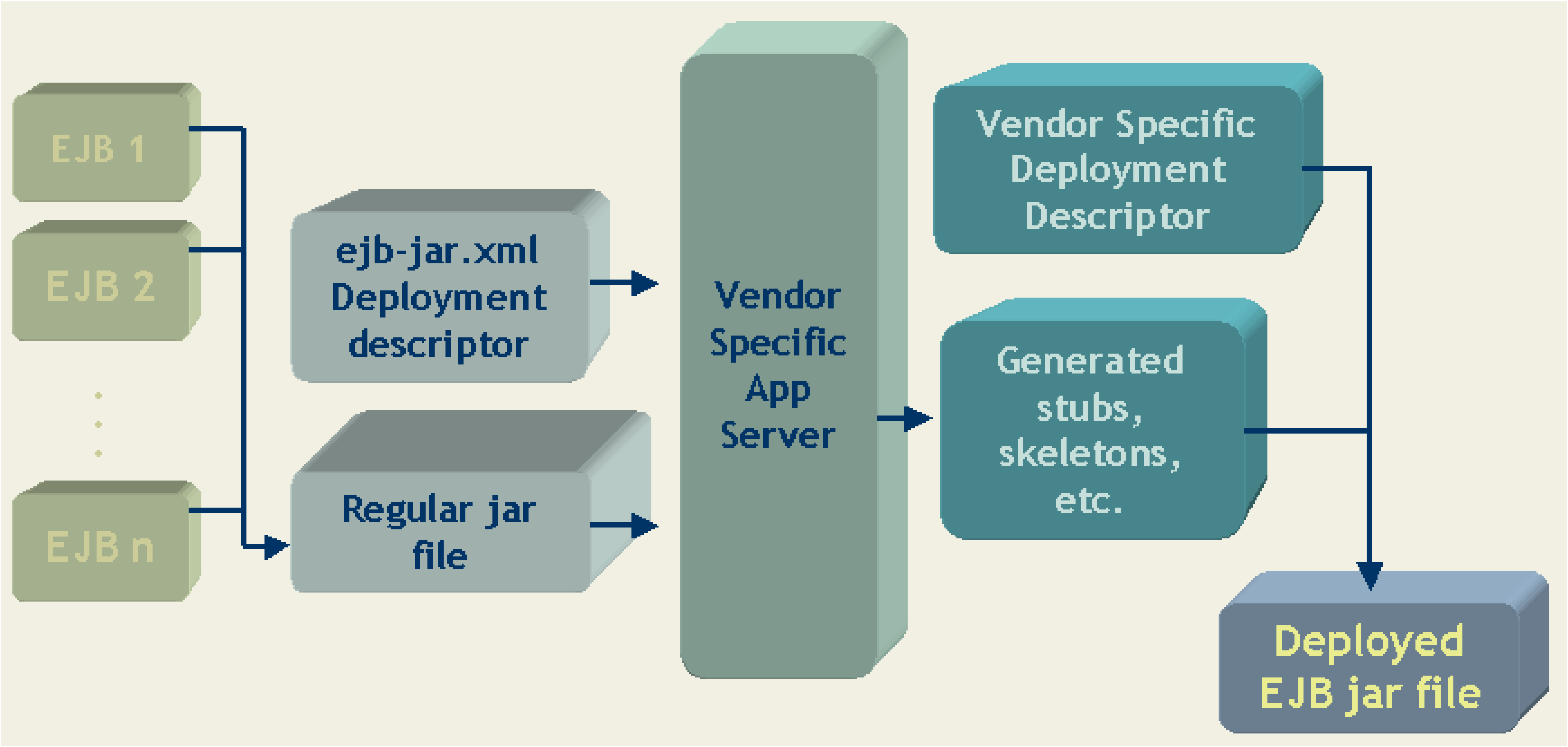
- J2EE applications may play the role of a client or a server depending on which tier they are deployed.

- J2EE applications are not required to be distributed, but they should be engineered so that they may be deployed on a distributed system.

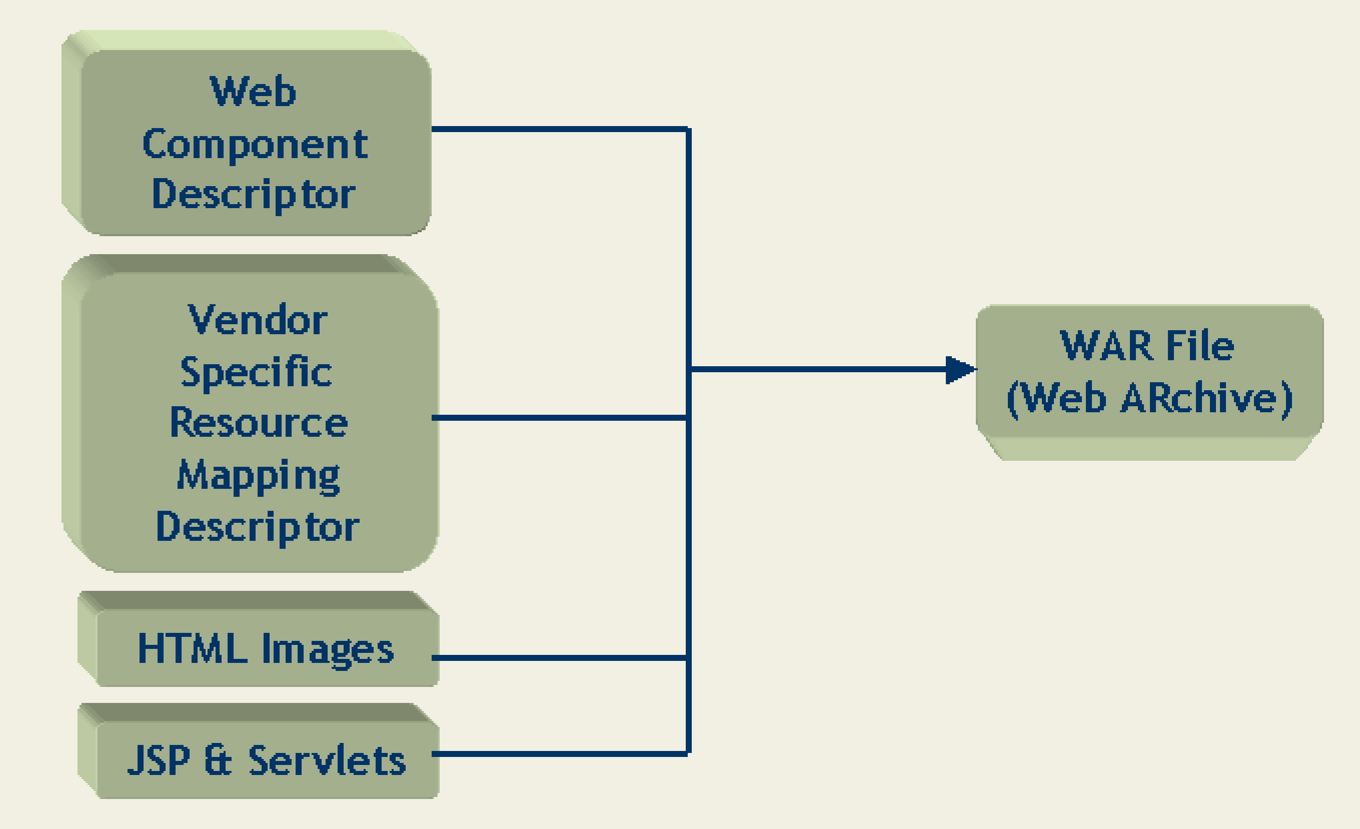
J2EE Packaging

- A J2EE application, packaged in an **E**nterprise **Ar**chive (EAR file), consists of:
  - One or more J2EE modules
    - EJB packaged in JAR files
    - Web components (JSP, Servlet, HTML etc.) packaged in WAR files
- One J2EE application deployment descriptor - *application.xml*
  - *application.xml* lists all the J2EE modules
- Each module consists of one or more J2EE components
  - An EJB JAR may contain:
    - One or more EJB components
    - A deployment descriptor - ejb-jar.xml
  - A WAR may contain:
    - Multiple Servlets, JSP, HTML WAR file
    - A deployment descriptor - web.xml

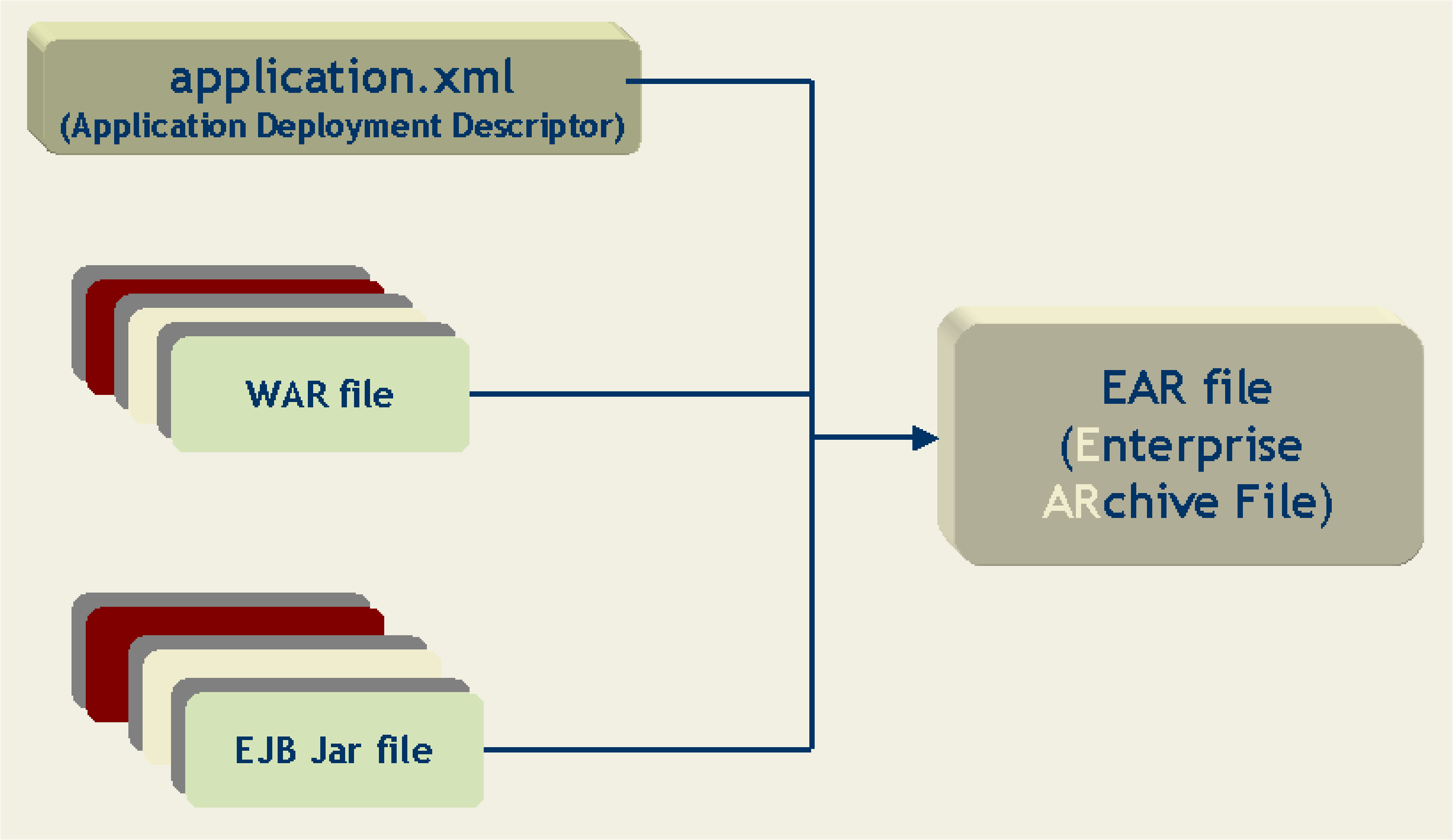
Packaging the EJB Tier


EJB 1
EJB 2
EJB n
ejb-jar.xml Deployment descriptor
Regular jar file
Vendor Specific App Server
Vendor Specific Deployment Descriptor
Generated stubs, skeletons, etc.
Deployed EJB jar file

Packaging the Web Tier



Packaging and Deploying J2EE Applications



- The J2EE standard also specifies a deployment descriptor for deploying client applications.

Overview of J2EE Technologies

- J2EE architecture is based on the following Java technologies:
  - Component Technologies: EJB, servlets, JSP
  - Service Technologies: Java Database Connectivity (JDBC), Java Transaction API (JTA) /Java Transaction Services (JTS), Java Naming Directory Interface (JNDI), connectors
  - Communication Technologies: Java IDL, RMI-IIOP, Java Messaging Service (JMS), JavaMail API

Enterprise JavaBeans Components

- EJB architecture – A server-side component model for the development and deployment of enterprise applications.
- Enterprise applications – Modeled as a collection of components (or enterprise beans) in which each component is a reusable software unit that implements a specific part of the business functionality and has a well-specified interface.
- EJB – Deployed on EJB servers and are hosted within containers. EJB depend on the container for certain services that can be declaratively customized at deployment time.

Types of EJBs

- There are three types of EJBs:
  - Session beans – Model task or process oriented aspects of a business application
  - Entity beans – Model persistent enterprise data
  - Message beans – Model asynchronous consuming of messages
- Components developed using EJB architecture are scalable, transactional, and concurrency safe.

Servlets

- Web components of a J2EE application that extend the functionality of a Web server. In J2EE applications, servlets are hosted by the Web container.
- Receive an HTTP request from a client and dynamically generate a response either in HTML or XML and send this response back to the client.
- May access enterprise resources such as databases or EJB in the EJB server.
- Maintain session information on behalf of the clients accessing them and may also interact with other servlets.

JavaServer Pages^{TM} (JSP)

- Enable the dynamic creation of Web pages. JSPs are also hosted by the Web container.
- Contain HTML or XML tags to format the Web document and Java technology code to generate the dynamic content. The Java technology code is executed each time the JSP is accessed.
- Use JavaBeans components to perform complex operations or to access EJB.
- The Web containers, hosting the JSP, typically compile the JSP into a servlet.

Java Database Connectivity (JDBC^{TM})

- ◆ The J2EE platform uses JDBC to access relational databases. JDBC contains classes and interfaces for:
  - Loading and configuring a database driver provided by database vendors
  - Connecting and authenticating a database server using URLs
  - Executing Standard Query Language (SQL) queries and updates, including compiled queries and stored procedures
  - Navigating query results in cursor based structures

Java Transaction API (JTA)

- Enables applications and application servers to access transactions in an implementation independent manner

- Specifies standard interfaces between a transaction manager and the modules involved in a distributed transaction:
  - Application
  - Application server
  - Manager that controls access to the shared resources affected by the transaction

Java Transaction Services (JTS)

- The JTS specifies the implementation of a transaction manager that supports JTA and implements the Java mapping of the OMG Object Transaction Services (OTS).
- A JTS transaction manager provides the services required to support demarcated transactions, resource management, synchronization, and transaction context propagation using IIOP.

Java Naming Directory Interface (JNDI)

- Provides the naming and directory services in an implementation independent manner
- Provides the enterprise applications with methods to:
  - Associate attributes with objects
  - Store the objects in name spaces
  - Retrieve objects by name or other attributes
- Is independent of the underlying service implementation

Java Interface Definition Language (Java IDL)

- Enables Java technology applications to invoke operations on remote objects whose interfaces are defined using the OMG Interface Definition Language (IDL).

- Java IDL contains:
  - The IDL compiler idlj
  - A Common Object Request Broker Architecture (CORBA) API
  - A CORBA-compliant Object Request Broker (ORB)

RMI-IIOP

- RMI-IIOP is an implementation of the Java Remote Method Invocation (RMI) protocol over the OMG Internet Inter-ORB Protocol (IIOP).
- RMI-IIOP specifies remote interfaces for business objects in Java technology, which can be converted to IDL.
- For remote interfaces defined using RMI, RMI-IIOP provides interoperability with CORBA objects implemented in any other language.

RMI-IIOP

- The J2EE platform provides a RMI interface compiler, rmic, that generates:
  - Client and server stubs that work with any CORBA compliant ORB
  - An IDL file that is compatible with the RMI interfaces

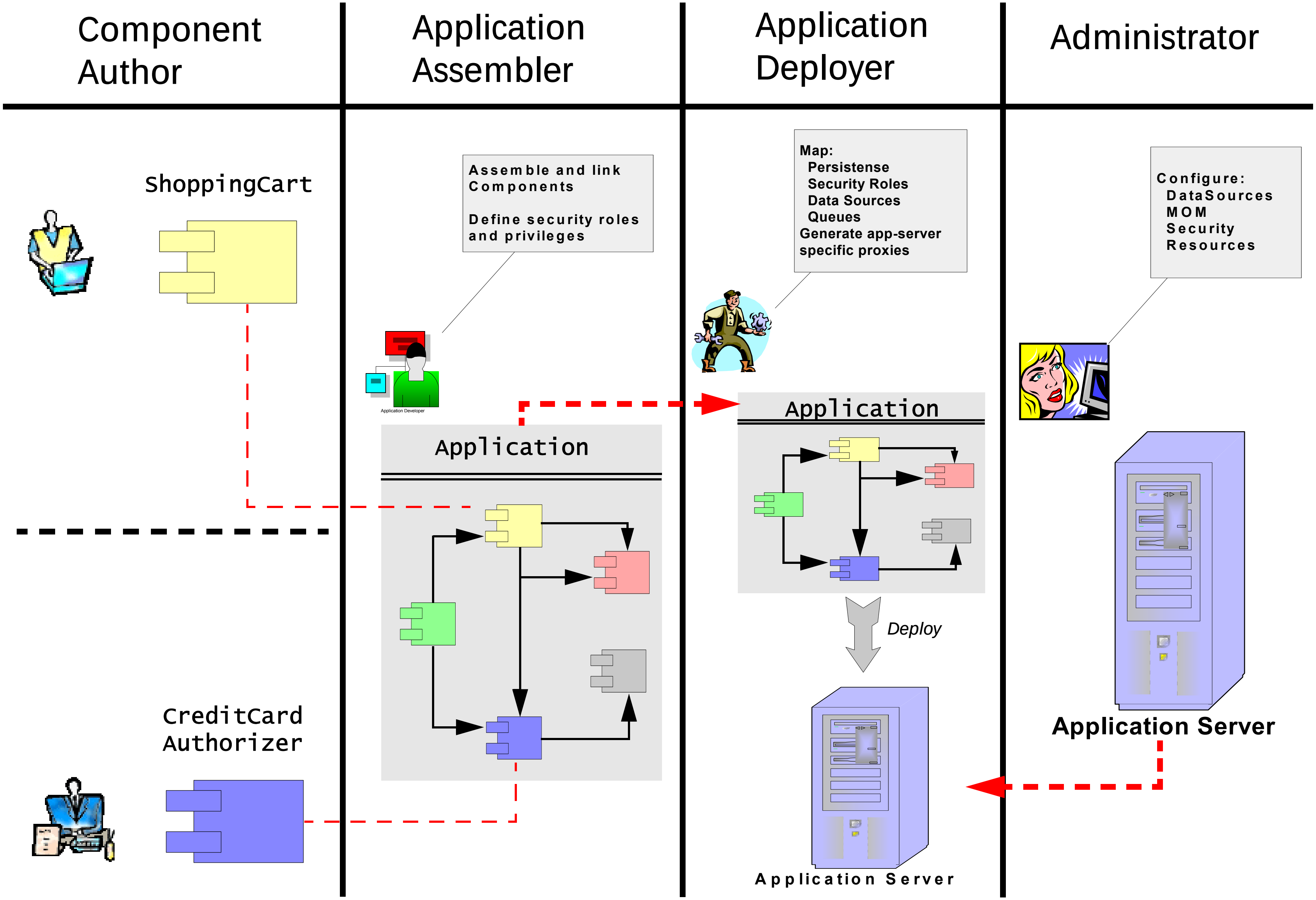
Java Messaging Service (JMS)

- JMS is an API for using enterprise messaging systems such as IBM MQSeries, and so on.
- JMS messages contain well-defined information that describe specific business actions.
- The JMS supports two messaging models
  - point-to-point
  - publish-subscribe

JavaMail API

- The JavaMail API:
  - Provides a set of abstract classes and interfaces that define the objects comprising an electronic mail system
  - Includes concrete implementations of commonly used Internet mail protocols
- The classes:
  - Support various mail system implementations
  - Can be extended to provide new protocols and functionality

J2EE Roles



Component Author

Application Assembler

Application Deployer

Administrator

Assemble and link Components

Define security roles and privileges

Map:
- Persistense
- Security Roles
- Data Sources
- Queues

Generate app-server specific proxies

Configure:
- DataSources
- MOM
- Security
- Resources

Why Enterprise JavaBeans (EJB)?

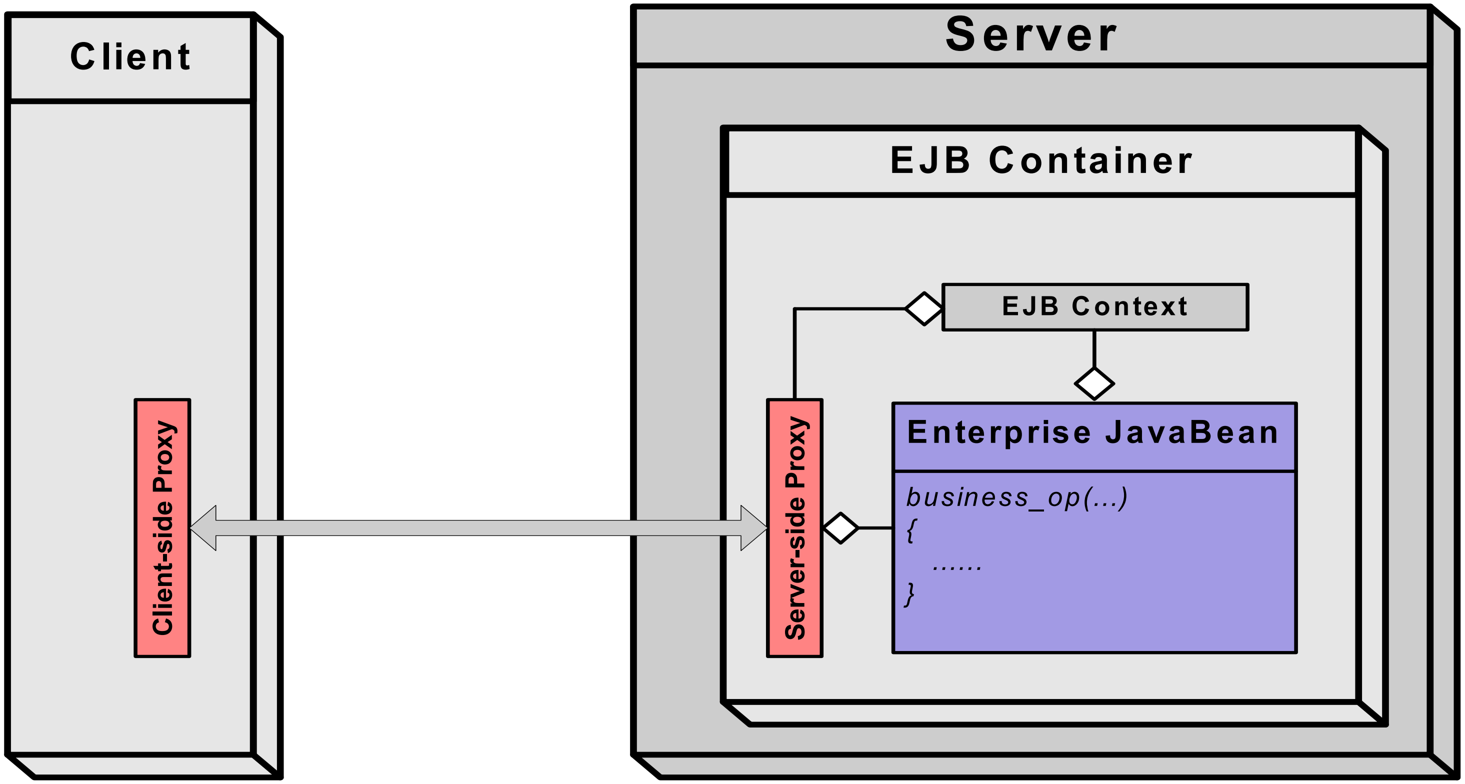
This section covers topics that address:

- The reasons for using Enterprise JavaBeans (EJB)
- Component Transaction Monitors (CTM)
- What are EJBs ?

Why EJB?

- Enterprise Java Beans enable the development of component-based, enterprise applications where the execution environment provides the following services:
  - Transactions (even distributed transactions!!)
  - Security
  - Persistence
  - Resource management
    - Objects (memory)
    - Connections (Database, MOM, Legacy systems etc.)
    - Threads
  - Object distribution
  - Concurrency
- The application developer is free to focus on the business application logic
- Portability
  - Vendor independence from providers of:
    - Application servers
    - Database systems
    - Message Oriented Middleware
    - Authentication & Authorization systems

What are Enterprise JavaBeans ?



- Enterprise JavaBeans (EJB) are server-side components that are hosted and execute in the context of an EJB container

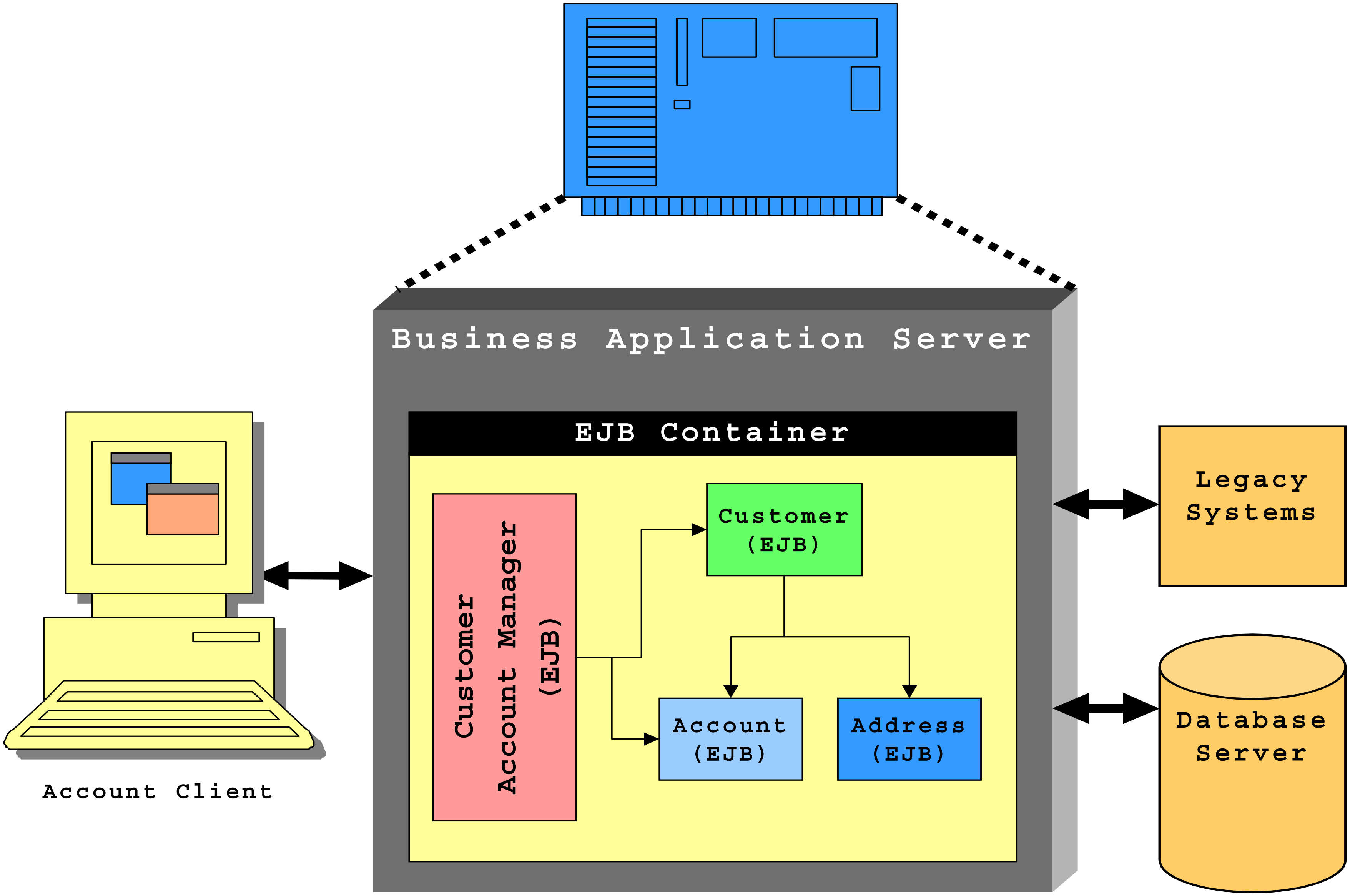
- Remote clients access the EJB using a proxy (aka *remote reference*)

- EJB is an implementation of a *Component Transaction Monitor* (CTM)

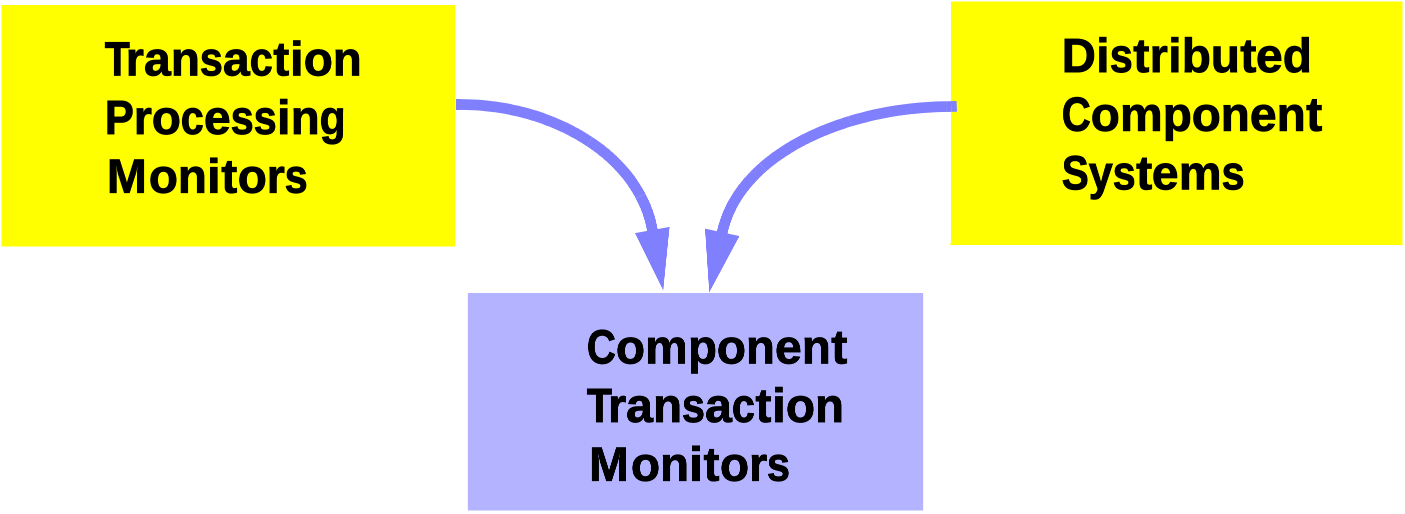
Enterprise JavaBeans

- EJB is a standard server-side component model for Component Transaction Monitors (CTM)
- EJB architecture enables the development and deployment of distributed, object-oriented, enterprise-class applications
- EJB based applications are:
  - Transactional
  - Persistence-aware
  - Secure
  - Scalable

A Typical EJB Example

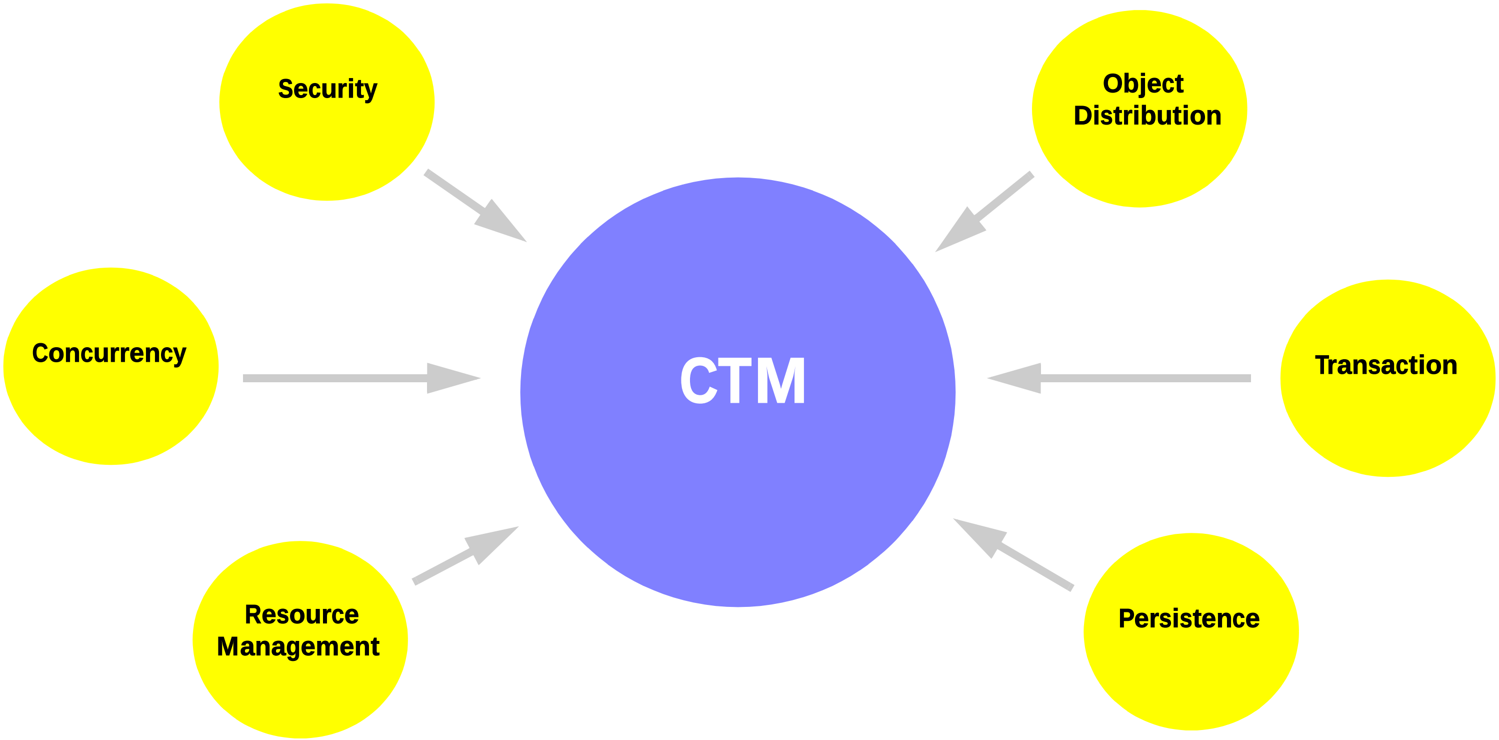


Component Transaction Monitors (CTM)


Transaction Processing Monitors
Distributed Component Systems
Component Transaction Monitors

- Transaction Processing (TP) Monitors
  - Robust and scalable transaction processing
  - Typically lack a real component model
  - CICS, Encina, Tuxedo
- Distributed component frameworks
  - Developer responsible for managing concurrency, scalability, persistence etc.
  - CORBA, DCOM
- Component Transaction Monitors combine the benefits of the above two
  - MTS, EJB

Component Transaction Monitor (CTM)



- A CTM provides the infrastructure for:
  - Transaction Management (even distributed !!)
  - Object Distribution
  - Concurrency
  - Security
  - Persistence
  - Resource Management

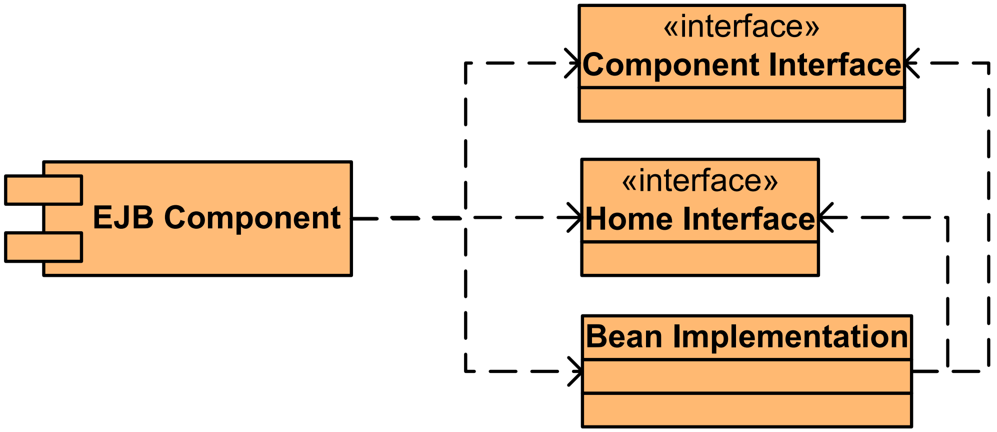
EJB Fundamentals

This section covers the following topics:

- Artifacts of an EJB component
- Deployment Descriptor
- Types of EJB
- EJB Containers

EJB Artifacts

- An EJB is a coarse-grained server-side Java object hosted in an EJB container



- Typically*, associated with each EJB component, we have the following:
  - Component interface
    - Declares all the business methods that users of this bean may invoke
  - Home interface
    - Declares all the life-cycle and *factory* methods
  - Bean implementation class containing all the implementation code
    - Needs to implement all the *Component Interface* methods
    - Needs to "support" the *Home Interface* methods

- Component and Home interfaces may be either *Local* or *Remote*

(* Message-Driven Beans are an exception)

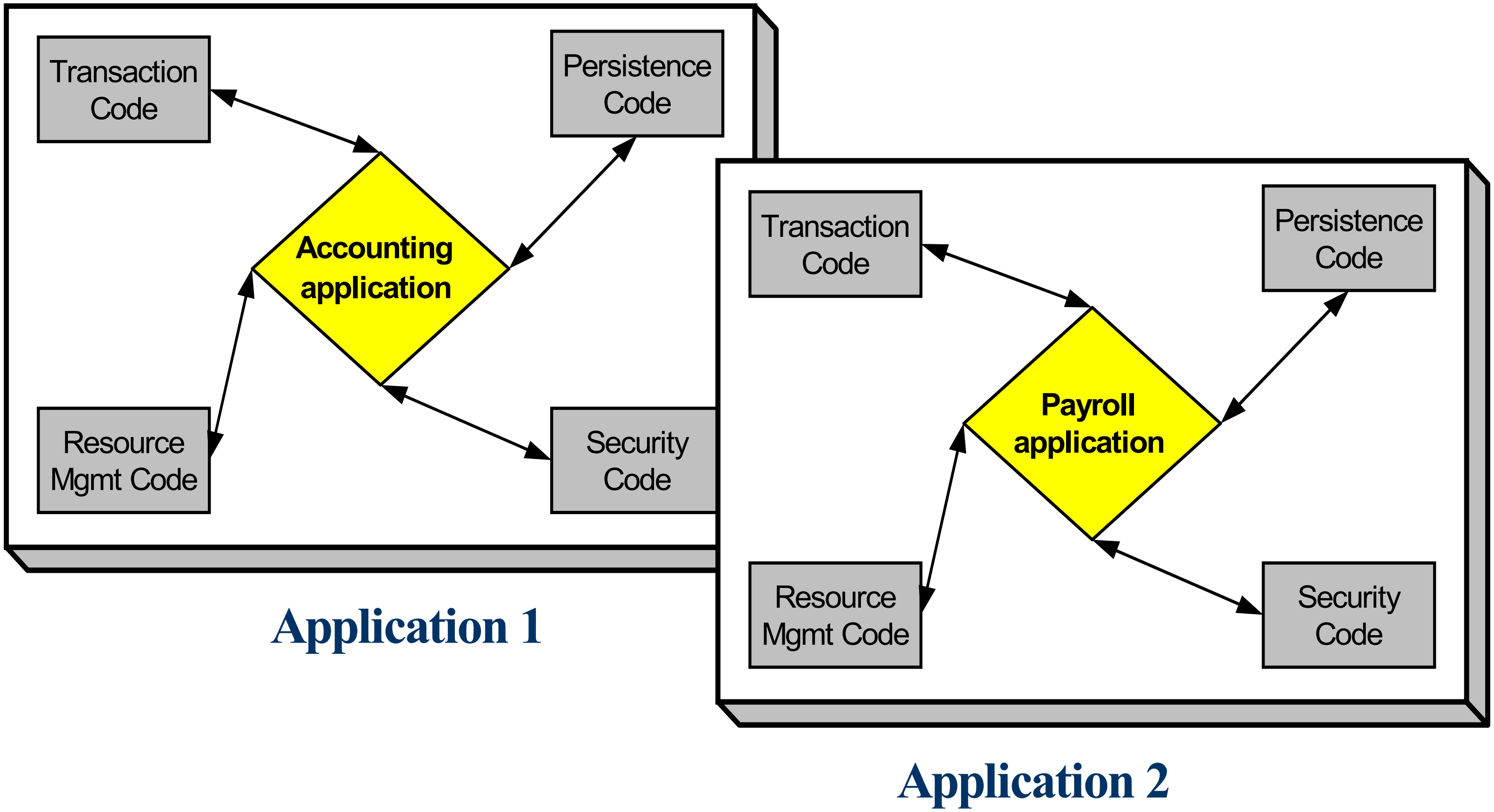
EJB Deployment Descriptor

- The deployment descriptor is an XML document that contains:
  - The declaration of an EJB
    - Home interface
    - Component interface
    - Bean implementation class
  - Transaction attributes
  - Security policies
  - Persistence attributes
  - Concurrency attributes (Isolation)
  - Resource management attributes
    - Size of connection and instance pools
  - External resource references
    - DataSources
    - Messaging Systems
  - Definitions of relationships among EJB components
  - Names of EJB home objects to be used to register in a JNDI-based naming service
- The standard name of this file is **ejb-jar.xml**
  - EJB container vendor’s extensions are specified using additional files

EJB Containers

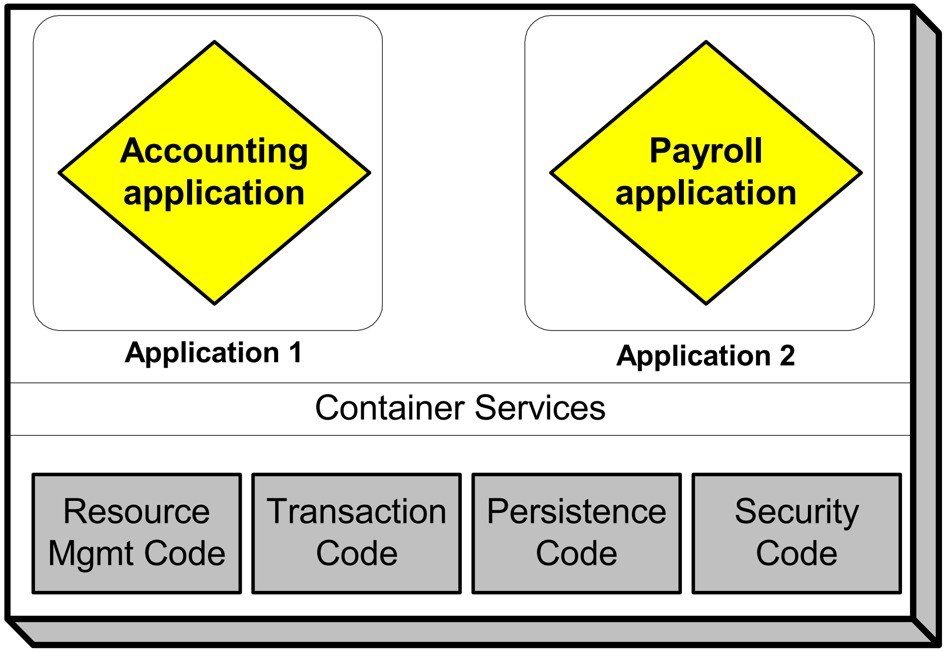
- Provides runtime support for EJB components
- Interpose between EJB components and services
  - Transparently inject services that are specified and configured in the deployment descriptor
- Must provide a Java™ technology compatible runtime environment
  - Based on Java™ 2 Platform, Standard Edition, v1.3 (J2SE™)
- Must support all of the J2EE standard services
  - Can be extended using connectors to external resources
- Provides federated view of underlying services
  - Transaction management
  - Life cycle management
  - Naming services (JNDI)
  - Resource pooling for database connections, objects
  - Instantiate enterprise beans on behalf of the client
  - Manage the enterprise bean storage
  - Manage the security context

Why Containers ?



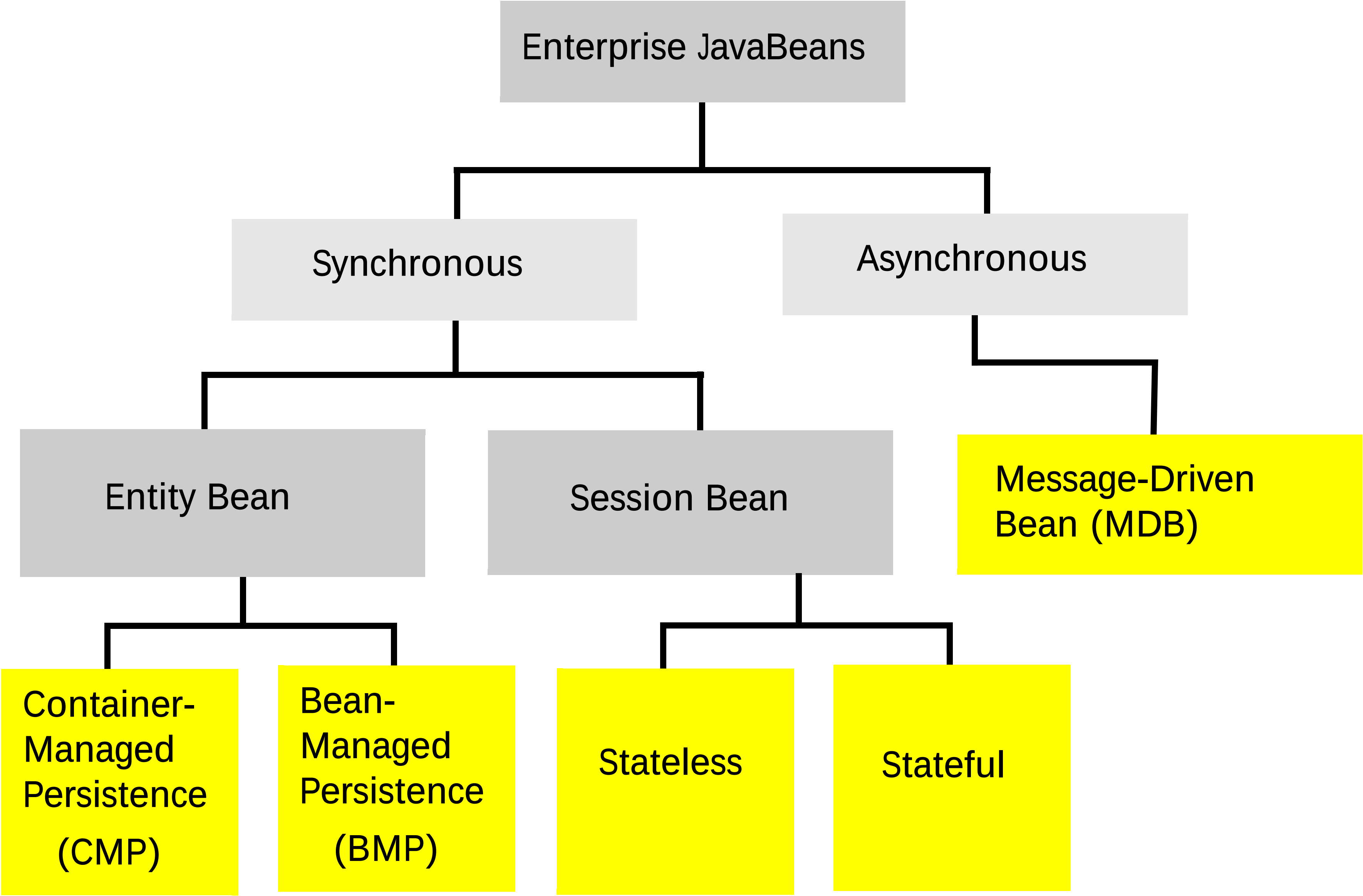
- Without containers, every enterprise application has to implement :
  - The application logic
  - Support code for transactions, security, persistence etc.

With Containers



- Containers provide a common substrate of services
- Enterprise programmers can now concentrate on implementing the business logic
- The services are now reused across multiple applications
- The services, implemented by the container, are typically developed by experts in those areas

Different Kinds of EJBs


Enterprise JavaBeans
Synchronous
Asynchronous
Entity Bean
Session Bean
Message-Driven Bean (MDB)
Container-Managed Persistence (CMP)
Bean-Managed Persistence (BMP)
Stateless
Stateful

EJB Types

- EJBs fall into three broad categories
  - Entity Beans
  - Session Beans
  - Message-Driven Beans (MDB)

- Entity Beans are used to model data objects or business concepts that correspond to ***types*** (persistent) from the ***type model***
  - They fall in to two subcategories
    - Container Managed Persistence (CMP)
    - Bean Managed Persistence (BMP)

- Session Beans are used to model application services
  - Used to model a set of *use cases* for an *actor*
  - Methods correspond to *use cases*
  - They are further divided into:
    - Stateful Session Beans
    - Stateless Session Beans

- Message-Driven Beans are used to process asynchronous messages sent by messaging clients

Local and Remote Enterprise JavaBeans

- Session and Entity beans can further be classified as:
  - Remote
  - Local

- A ***remote*** EJB can be accessed by:
  - Remote callers
    - i.e. clients that exist outside the EJB container containing the EJB
  - Local clients
    - i.e. Callers that co-exist in the same EJB container with the EJB

- All parameters and return values are returned by **value** during the method invocation of methods on a ***remote*** EJB
  - Even when the callers are in the same EJB Container !!

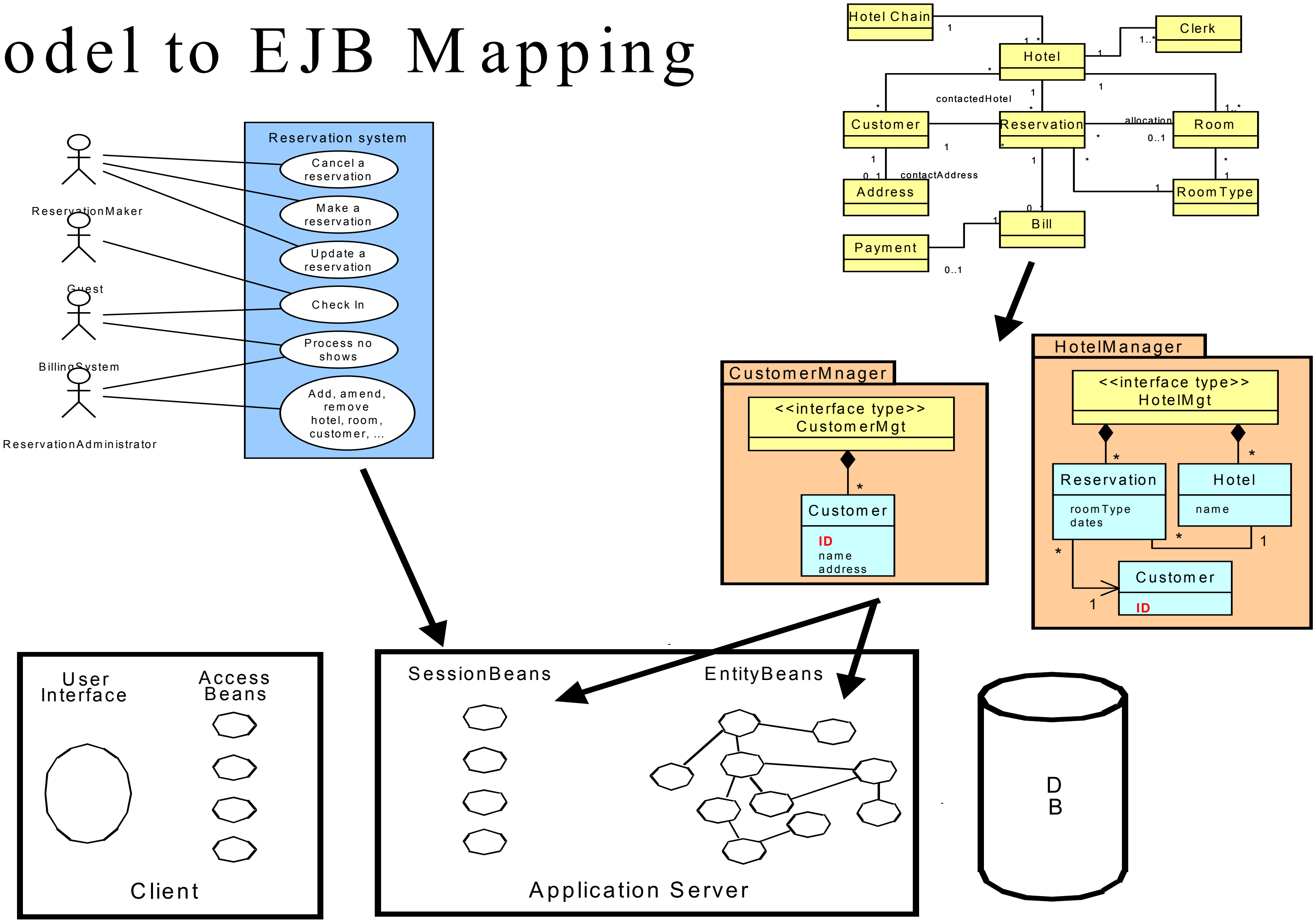
- A local EJB can **only** be accessed by *local* callers

- All parameters and return values are returned by **reference** during the method invocation of methods on a ***local*** EJB

- Entity beans should normally be modelled as *local* objects

Which EJB types should I use?

Model to EJB Mapping



Proxy Pattern

The proxy design pattern is fundamental to the EJB framework. In this section, we shall cover the following patterns:

- Basic Proxy
- Persistence Proxy
- Transaction Proxy
- Security Proxy
- Concurrency Proxy
- Virtual Object Proxy
- Communication Proxy

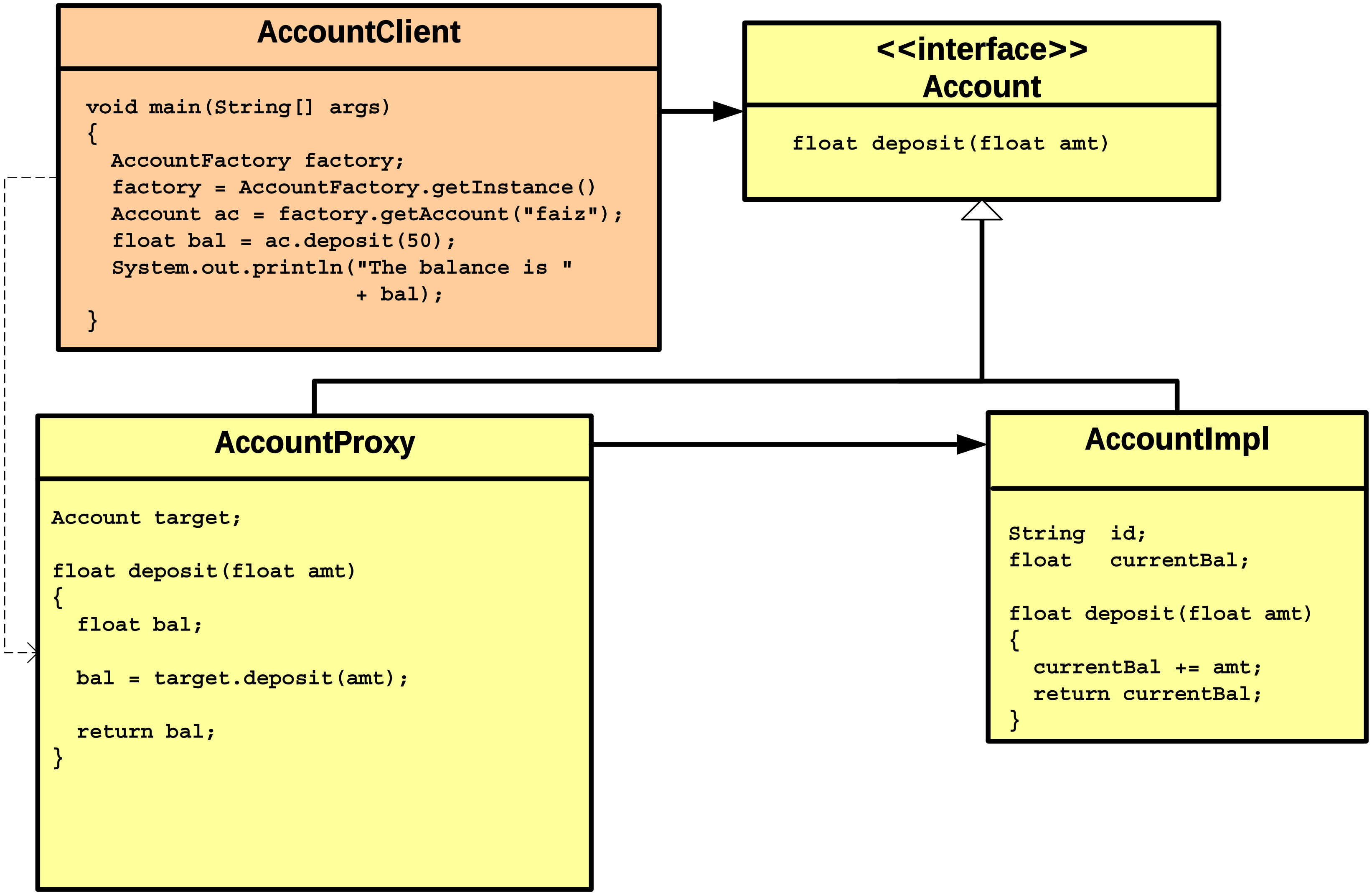
Basic Proxy

- Problem:
  - We want to decouple the user (client) of a service (method) from the class and object implementing that service (method)

- Reasons:
  - Security
  - The class implementing the method might change in the future

- Description:
  - Specify the service using an interface, e.g.
    - Interface: `Account`
    - Service method : `float deposit(float amount);`
  - Build the user (client) code in terms of this interface
  - Define a pair of classes that implement this interface
    - One class that implements the logic, `AccountImpl`
    - One class that delegates the work to the implemention object, `AccountProxy`
  - At run time:
  - The user (client) uses an instance of the proxy class, `AccountProxy`
  - The proxy uses an instance of the implementation class, `AccountImpl`

Basic Proxy



AccountClient

```
void main(String[] args)
{
  AccountFactory factory;
  factory = AccountFactory.getInstance()
  Account ac = factory.getAccount("faiz");
  float bal = ac.deposit(50);
  System.out.println("The balance is "
                     + bal);
}
```

<<interface>>
Account

```
float deposit(float amt)
```

AccountProxy

```
Account target;

float deposit(float amt)
{
  float bal;

  bal = target.deposit(amt);

  return bal;
}
```

AccountImpl

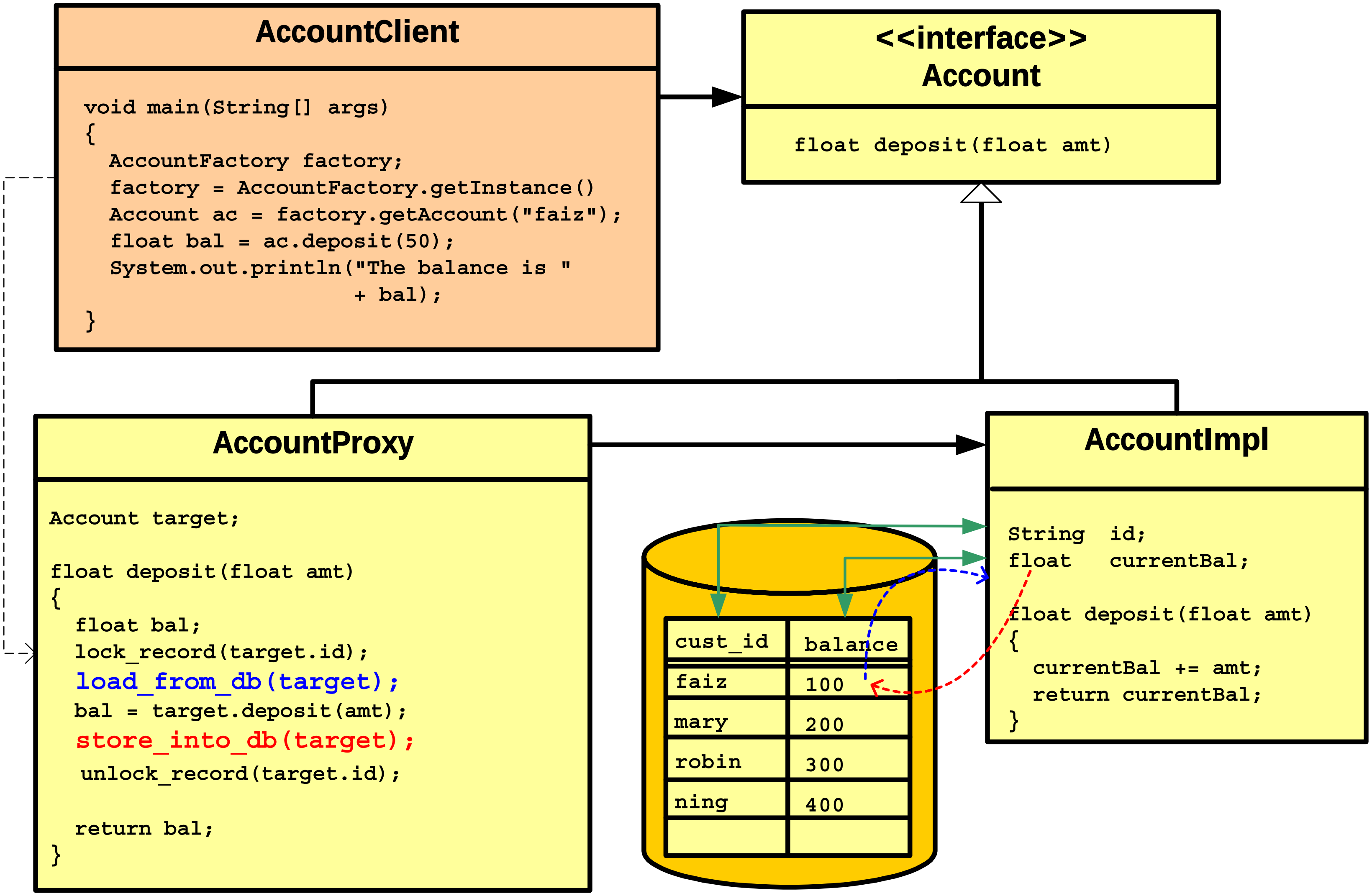
```
String  id;
float   currentBal;

float deposit(float amt)
{
  currentBal += amt;
  return currentBal;
}
```

Persistence Proxy

- Problem:
  - The implementation object is persistent
    - i.e. it represents data/record from a database system
  - It is possible that the state of the in-memory component may be different from the corresponding state in the database
    - Another application may have modified the data on the database system
- Reasons:
  - Neither the client nor the component implementor should be responsible for synchronizing the in-memory component with the data on the database
    - Neither of them should be required to have the expertise in data access
- Description:
  - In the proxy, perform the following steps in the business method (`deposit`)

1. Take a lock on the corresponding record(s) in the database
2. Load the data from the database into the implementation object
3. Invoke the business method (`deposit`) on the implementation object
4. Store the data from the implementation object into the database
5. Release the record lock(s)

Persistence Proxy



```
void main(String[] args)
{
  AccountFactory factory;
  factory = AccountFactory.getInstance()
  Account ac = factory.getAccount("faiz");
  float bal = ac.deposit(50);
  System.out.println("The balance is "
                      + bal);
}
```

```
float deposit(float amt)
```

```
Account target;

float deposit(float amt)
{
  float bal;
  lock_record(target.id);
  load_from_db(target);
  bal = target.deposit(amt);
  store_into_db(target);
  unlock_record(target.id);

  return bal;
}
```

cust_id	balance
faiz	100
mary	200
robin	300
ning	400

```
String  id;
float   currentBal;

float deposit(float amt)
{
  currentBal += amt;
  return currentBal;
}
```

Transaction Proxy

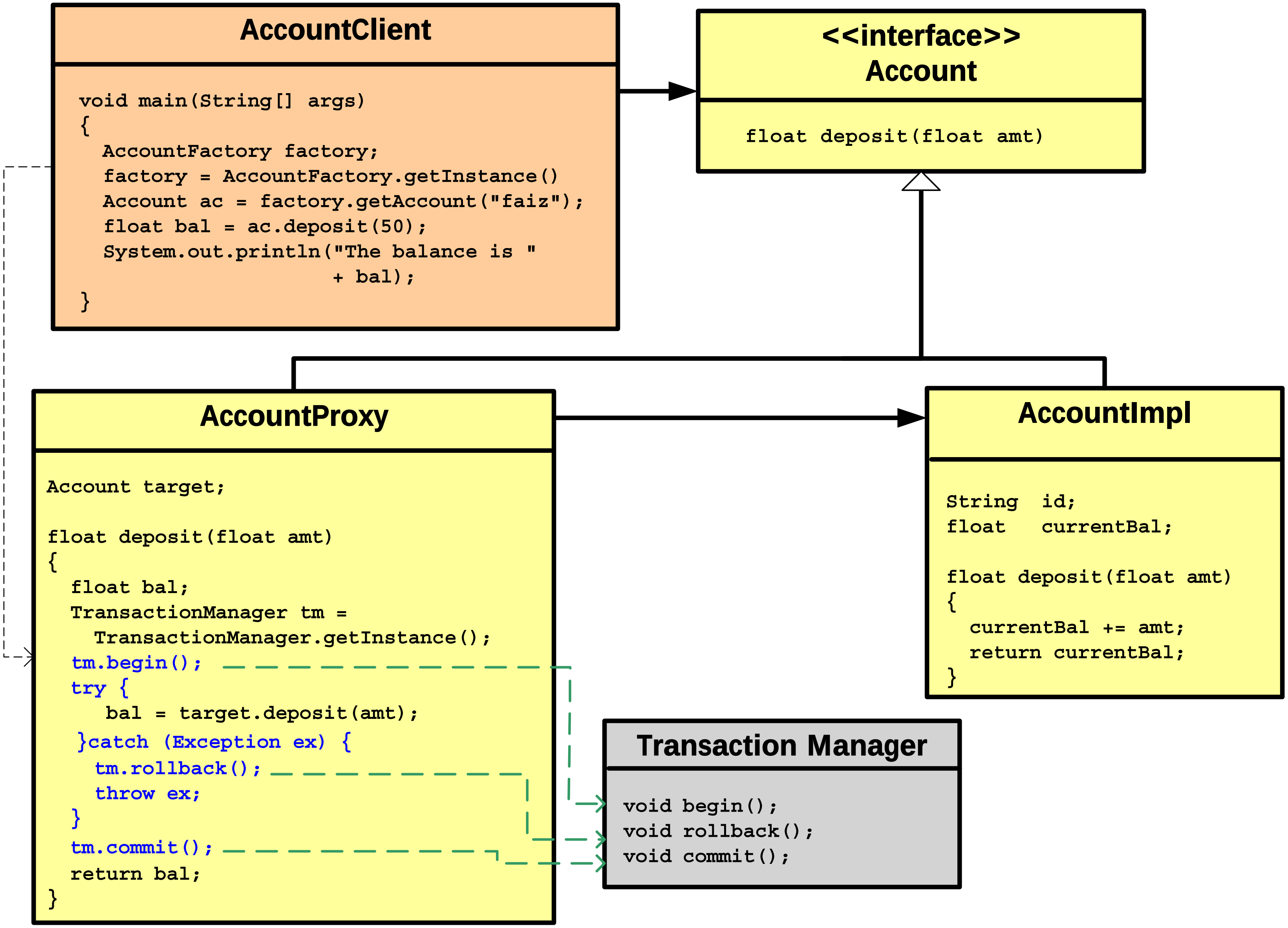
- Problem:
  - The service method of the component may need to be invoked in the context of a transaction
  - There may be multiple components each connected to their respective transactional resources (databases, MOM etc.)
    - Distributed transactions, two-phase commits etc.

- Reasons:
  - Neither the client nor the component implementor should be responsible for managing transactions
  - Neither of them should be required to have the expertise in transaction processing

- Description:
  - In the proxy, perform the following steps in the business method (`deposit`)

1. Start a transaction
2. Invoke the business method (`deposit`) on the implementation object
3. If the operation is succesful, ***commit*** the transaction
4. If the operation is unsuccesful, ***rollback*** the transaction

Transaction Proxy



AccountClient

```
void main(String[] args)
{
  AccountFactory factory;
  factory = AccountFactory.getInstance()
  Account ac = factory.getAccount("faiz");
  float bal = ac.deposit(50);
  System.out.println("The balance is "
                     + bal);
}
```

<<interface>>
Account

```
float deposit(float amt)
```

AccountProxy

```
Account target;

float deposit(float amt)
{
  float bal;
  TransactionManager tm =
    TransactionManager.getInstance();
  tm.begin();
  try {
     bal = target.deposit(amt);
   }catch (Exception ex) {
    tm.rollback();
    throw ex;
  }
  tm.commit();
  return bal;
}
```

AccountImpl

```
String  id;
float   currentBal;

float deposit(float amt)
{
  currentBal += amt;
  return currentBal;
}
```

Transaction Manager

```
void begin();
void rollback();
void commit();
```

Security Proxy

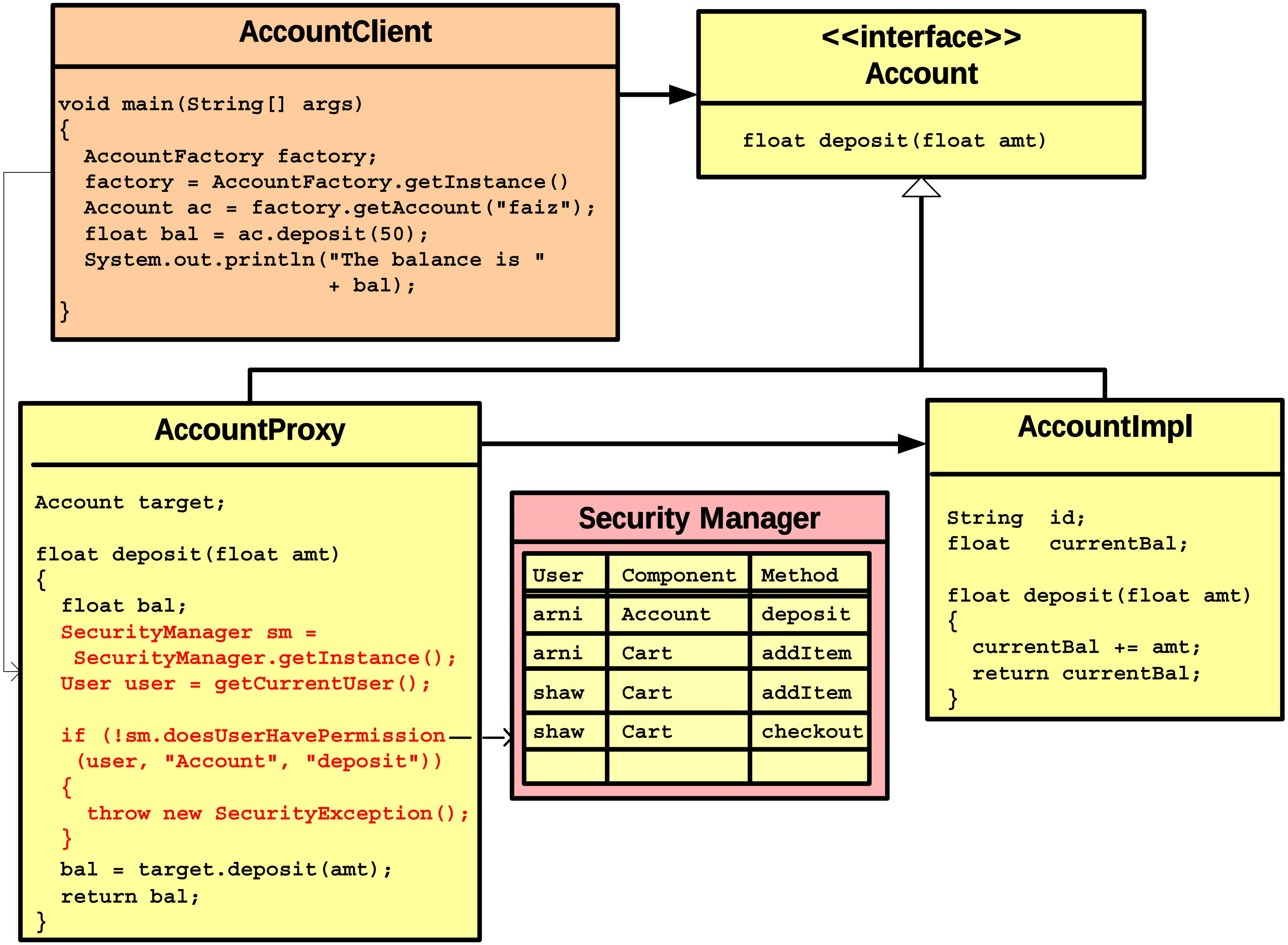
- Problem:
  - Not all users of a component may have the permissions to invoke the methods on the component

- Reasons:
  - Neither the client nor the component implementor should be responsible for managing security
  - Neither of them should be required to have the expertise in security policy enforcement

- Description:
  - In the proxy, perform the following steps in the business method (`deposit`)

1. Access the security manager where all the access rules are stored
2. Access the user’s identity (typically done using the current thread)
3. Verify with the security manager whether the user has the necessary privileges to to invoke the business method
4. If the caller has the necessary privileges, invoke the business method (`deposit`) on the implementation object
5. If the caller does not have the necessary privileges, throw an exception

Security Proxy



AccountClient

```
void main(String[] args)
{
  AccountFactory factory;
  factory = AccountFactory.getInstance()
  Account ac = factory.getAccount("faiz");
  float bal = ac.deposit(50);
  System.out.println("The balance is "
                   + bal);
}
```

<<interface>>
Account

```
float deposit(float amt)
```

AccountProxy

```
Account target;

float deposit(float amt)
{
  float bal;
  SecurityManager sm =
   SecurityManager.getInstance();
  User user = getCurrentUser();

  if (!sm.doesUserHavePermission
    (user, "Account", "deposit"))
  {
    throw new SecurityException();
  }
  bal = target.deposit(amt);
  return bal;
}
```

Security Manager

User	Component	Method
arni	Account	deposit
arni	Cart	addItem
shaw	Cart	addItem
shaw	Cart	checkout

AccountImpl

```
String  id;
float   currentBal;

float deposit(float amt)
{
  currentBal += amt;
  return currentBal;
}
```

Concurrency Proxy

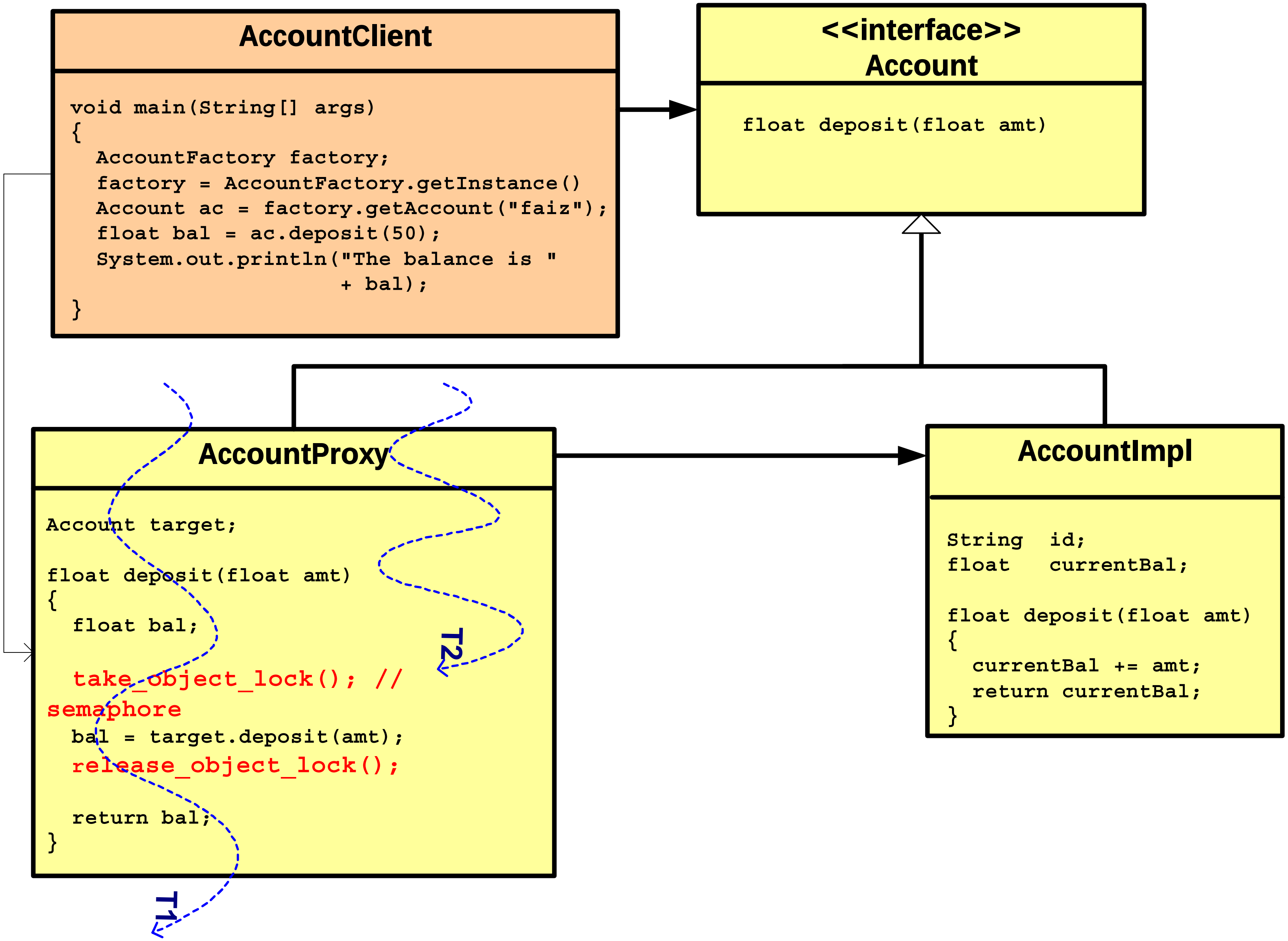
- Problem:
  - We may have multiple callers accessing the same component concurrently
  - The operations invoked may alter the state of the component
  - If the component is not shielded from concurrent access, we may potentially get inconsistent results

- Reasons:
  - Neither the client nor the component implementor should be responsible for managing concurrency
  - Neither of them should be required to have the expertise in concurrent and parallel programming
    - These skills are not easily available !!

- Description:
  - In the proxy, perform the following steps in the business method (`deposit`)

1. Take a semaphore on the object (i.e. lock the instance)
   - In Java, you may use the keyword `synchronize`
2. Invoke the business method (`deposit`) on the implementation object
3. Release the semaphore

Concurrency Proxy



AccountClient

```
void main(String[] args)
{
  AccountFactory factory;
  factory = AccountFactory.getInstance()
  Account ac = factory.getAccount("faiz");
  float bal = ac.deposit(50);
  System.out.println("The balance is "
                       + bal);
}
```

<<interface>>
Account

```
float deposit(float amt)
```

AccountProxy

```
Account target;

float deposit(float amt)
{
  float bal;

  take_object_lock(); //
semaphore
  bal = target.deposit(amt);
  release_object_lock();

  return bal;
}
```

AccountImpl

```
String  id;
float   currentBal;

float deposit(float amt)
{
  currentBal += amt;
  return currentBal;
}
```

Virtual Object Proxy

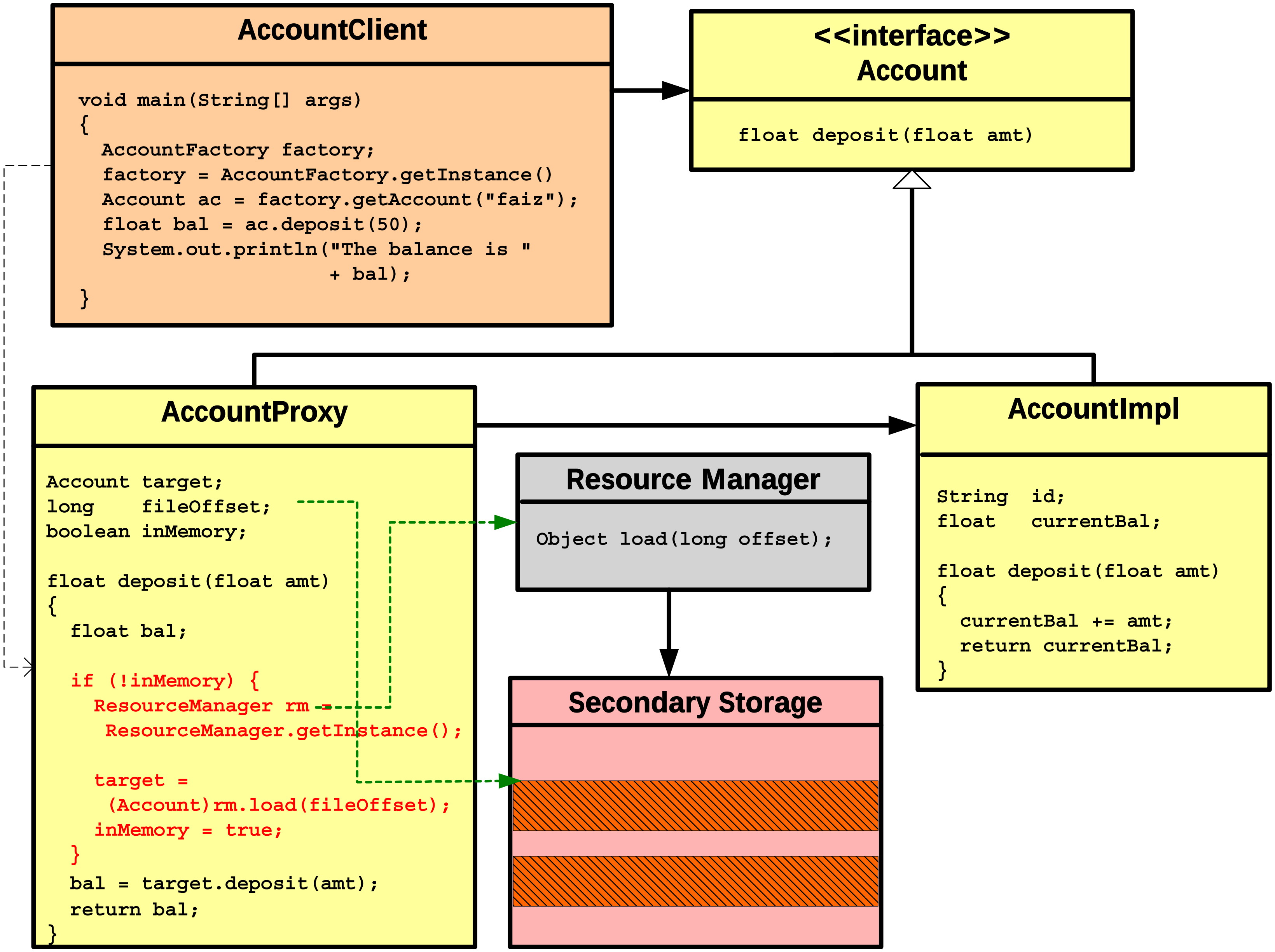
- Problem:
  - We may want to support a very large number of components
  - We may not have enough memory to hold all the components in memory

- Reasons:
  - Neither the client nor the component implementor should be responsible for managing resources
  - Neither of them should be required to have the expertise in resource and object pool management
    - These skills are usually found in operating system developers - a rare (and expensive) breed

- Description:
  - In the proxy, keep a flag that indicates whether the implementation object is in memory or on the secondary storage
  - Perform the following steps in the business method (`deposit`)

1. If the object is not in memory, load the object from the secondary storage
2. Invoke the business method (`deposit`) on the implementation object

Virtual Object Proxy



AccountClient

```
void main(String[] args)
{
  AccountFactory factory;
  factory = AccountFactory.getInstance()
  Account ac = factory.getAccount("faiz");
  float bal = ac.deposit(50);
  System.out.println("The balance is "
                     + bal);
}
```

<<interface>> Account

```
float deposit(float amt)
```

AccountProxy

```
Account target;
long    fileOffset;
boolean inMemory;

float deposit(float amt)
{
  float bal;

  if (!inMemory) {
    ResourceManager rm =
     ResourceManager.getInstance();

    target =
     (Account)rm.load(fileOffset);
    inMemory = true;
  }
  bal = target.deposit(amt);
  return bal;
}
```

Resource Manager

```
Object load(long offset);
```

Secondary Storage

AccountImpl

```
String  id;
float   currentBal;

float deposit(float amt)
{
  currentBal += amt;
  return currentBal;
}
```

Communication Proxy

- Problem:
  - The caller and the implementation component may be in distinct processes (and perhaps machines too)
- Reasons:
  - Neither the client nor the component implementor should be responsible for making and managing network connections
  - Neither of them should be required to have the expertise in network and socket level programming
- Description:
  - The proxy contains the attributes for:
    - The name of the host containing the server-side component
    - The port on the server where the server process is listening for connections
    - An object id, that uniquely denotes the server-side component instance

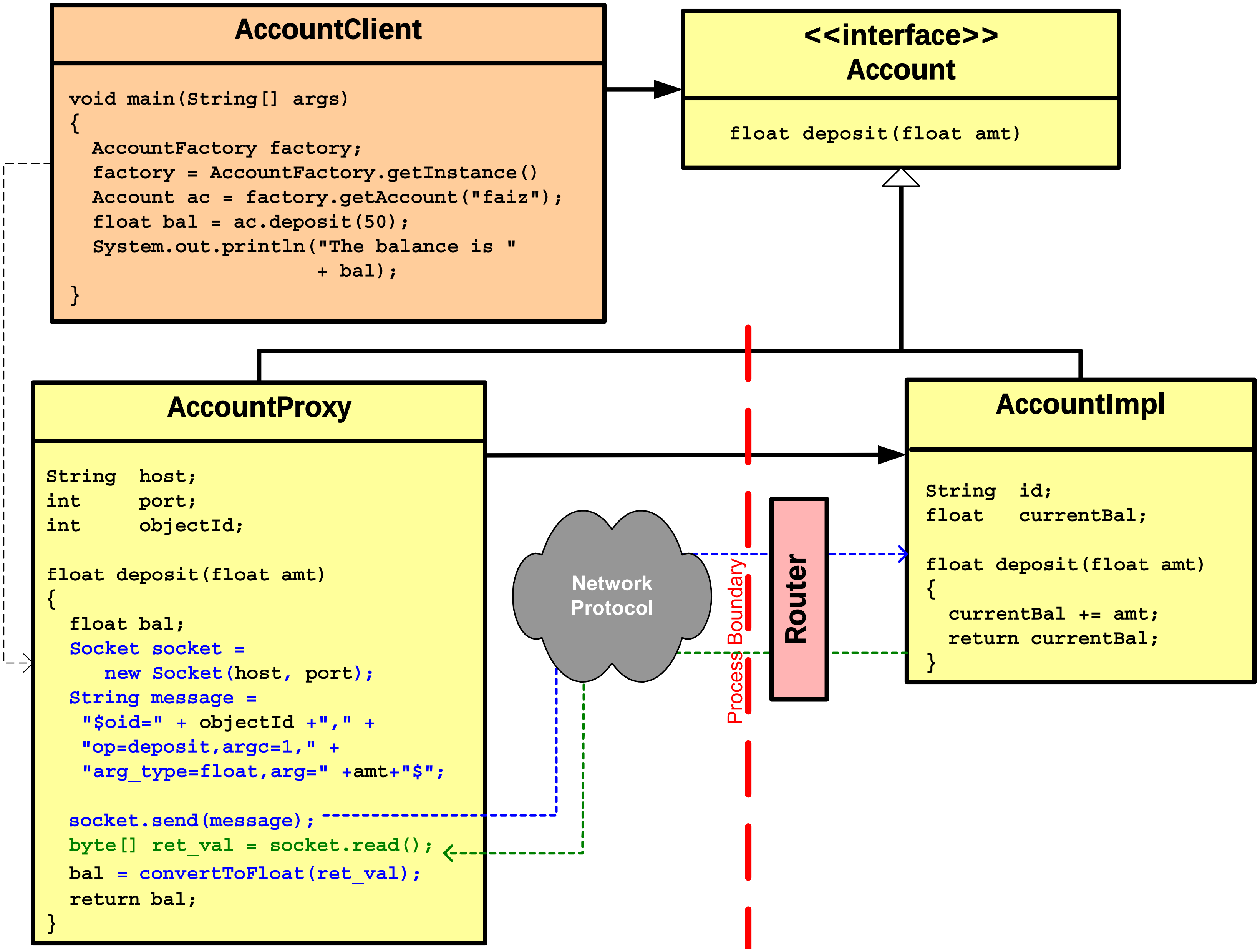
Communication Proxy (Contd.)

- Router
  - On the server, we also have another object called the ***Router***
  - The router is responsible for:

  1. Receiving messsage *requests* from remote clients
  2. Parsing the message *requests*
  3. Invoking the method, specified in the message, on the target component
  4. Collecting the return value and converting it to a network *response*
  5. Sending the *response* back to the remote caller

- In the proxy, perform the following steps in the business method (`deposit`)
  1. Using the host name and the port, estabilish a network connection with the server (the router)
  2. Construct a network message indicating the target object, the operation to be invoked and the parameters
  3. Send the message *request* using the socket
  4. Collect (*read*) the network *response*
  5. Convert the *response* to the appropriate return type and return to caller

Communication Proxy



```
void main(String[] args)
{
  AccountFactory factory;
  factory = AccountFactory.getInstance()
  Account ac = factory.getAccount("faiz");
  float bal = ac.deposit(50);
  System.out.println("The balance is "
                     + bal);
}
```

```
float deposit(float amt)
```

```
String  host;
int     port;
int     objectId;

float deposit(float amt)
{
  float bal;
  Socket socket =
     new Socket(host, port);
  String message =
   "$oid=" + objectId +"," +
   "op=deposit,argc=1," +
   "arg_type=float,arg=" +amt+"$";

  socket.send(message);
  byte[] ret_val = socket.read();
  bal = convertToFloat(ret_val);
  return bal;
}
```

```
String  id;
float   currentBal;

float deposit(float amt)
{
  currentBal += amt;
  return currentBal;
}
```

Enterprise JavaBeans Workflow

In the section, using a simple example, we shall take a conceptual look at the following:

- Development of EJB components
- Application assembly
- Deployment code generation
- Deployment of the EJB application

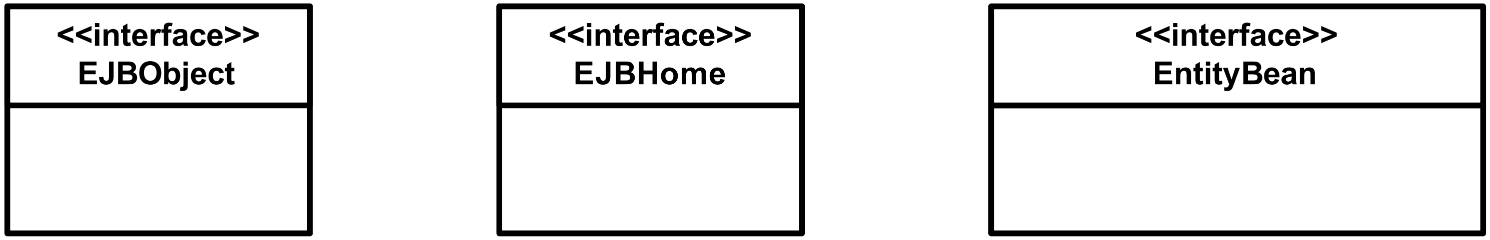
Description of the Example

- ◆ We shall create a component that represents a customer’s bank account
  - It shall have one operation, called `deposit`, to perform deposits
    - It takes in one parameter, of type `float`, representing the amount to be deposited
    - It returns the current balance which is also of type `float`
  - It shall have a pair of attributes
    - Customer name
    - Current balance

- ◆ What kind of EJB should this be?
  - Session?
  - Entity?
  - Message-Driven Bean?

- ◆ What should its distribution property be?
  - Remote?
  - Local?

EJB Development Process (Step 1)



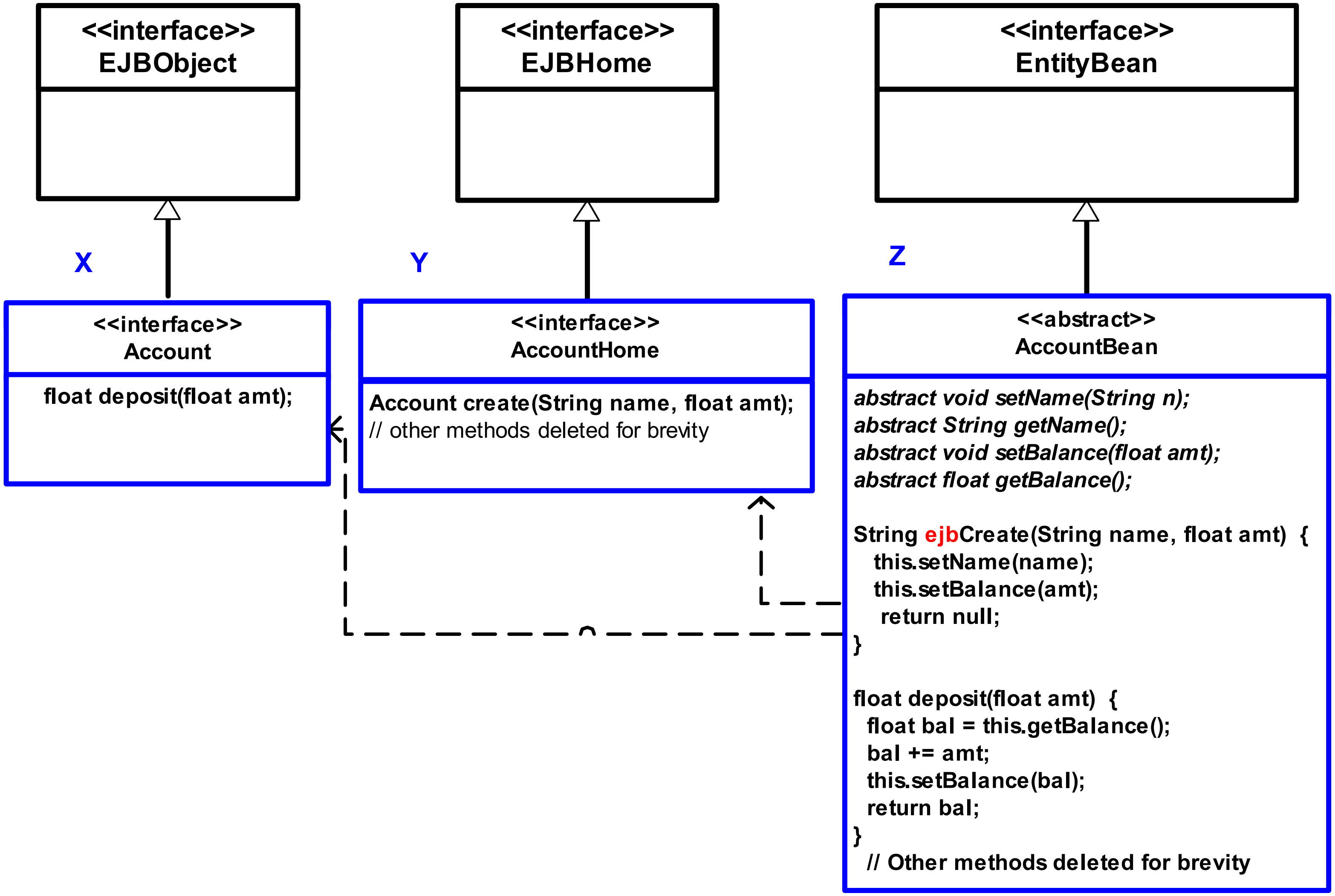
- We shall model the customer account component as a **remote entity** bean
  - Ideally it should be a *local* entity bean

- `EJBObject (javax.ejb.EJBObject)`
  - This interface is the general specification of a proxy
  - All remote EJB component interfaces must extend this interface

- `EJBHome (javax.ejb.EJBHome)`
  - This interface may be thought of as a general specification of a EJB *factory*
  - All remote EJB factories (referred to as *home* objects) must extend it

- `EntityBean (javax.ejb.EntityBean)`
  - This interface represents the contract between the EJB container and the entity bean
  - Contains a list of callback methods
  - All entity bean implementation classes must *implement* this interface

EJB Development Process (Step 2)



<<interface>>
EJBObject

<<interface>>
EJBHome

<<interface>>
EntityBean

X

Y

Z

<<interface>>
Account

float deposit(float amt);

<<interface>>
AccountHome

Account create(String name, float amt);
// other methods deleted for brevity

<<abstract>>
AccountBean

```
abstract void setName(String n);
abstract String getName();
abstract void setBalance(float amt);
abstract float getBalance();

String ejbCreate(String name, float amt)  {
   this.setName(name);
   this.setBalance(amt);
    return null;
}

float deposit(float amt)  {
  float bal = this.getBalance();
  bal += amt;
  this.setBalance(bal);
  return bal;
}
  // Other methods deleted for brevity
```

EJB Development Process (Step 2)

- The component interface, `Account`
  - Extend the interface `javax.ejb.EJBObject`
  - Specify the method, `deposit`

- The home interface, `AccountHome`
  - Extend the interface `javax.ejb.EJBHome`
  - Specify the factory method, `create`
  - This interface also contains other factory methods called ***finder*** methods, but we shall address those in a later section

- The ***abstract*** bean implementation class, `AccountBean`
  - Implement the interface `javax.ejb.EntityBean`
  - For each attribute, declare a pair of abstract *get/set* methods
    - `getName, setName, getBalance, setBalance`
  - Corresponding to the method, `create`, in the home interface implement the initializer method `ejbCreate`
    - This method should take the same parameters as the method `create`
    - This method should initialize all the attributes that map to the database
  - Implement the business method, `deposit`
  - Other methods, that also need to be implemented, will be covered later

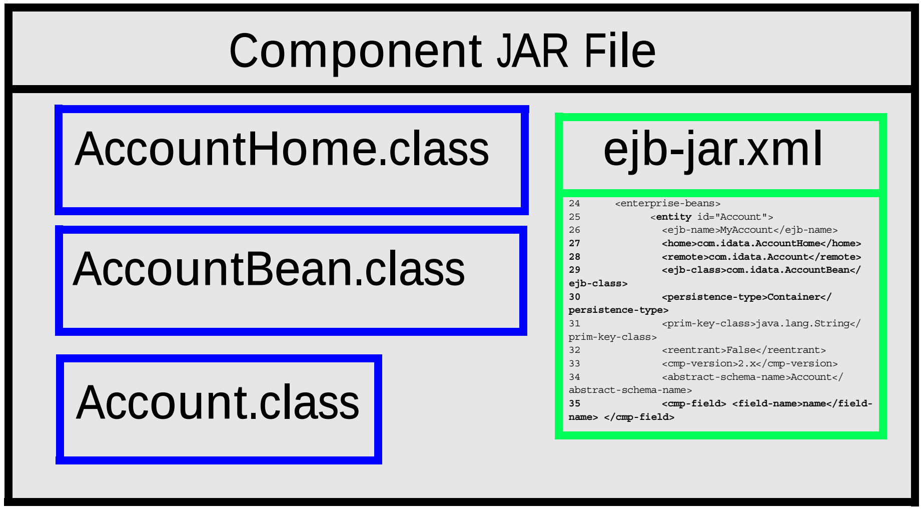
EJB Development Process (Step 3)

- The component developer specifies the deployment descriptor information in a document named ejb-jar.xml
  - The standard deployment descriptor file
- For each EJB component, the information provided here includes:
  - The type of the bean (Entity, Session, Message-Driven)
  - The home interface (Local and/or Remote)
  - The component interface (Local and/or Remote)
  - The bean class implementation
- For entity beans, we specify:
  - The primary key type
  - The entity bean type (CMP or BMP)
  - For container-managed persistent (CMP) entity beans, we also specify:
    - The abstract persistent schema
    - The fields (attributes) that map to database table columns
- For session beans, we specify the session bean type (stateless or stateful)
- For MDB, we specify the *destination* type (Topic or Queue)

EJB Development Process (Step 3)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC
     "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
     "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar id="ejb-jar_ID">
 <display-name>MyFirstEJBApp</display-name>
   <enterprise-beans>
      <entity id="Account">
        <ejb-name>MyAccount</ejb-name>
        <home>com.idata.AccountHome</home>
        <remote>com.idata.Account</remote>
        <ejb-class>com.idata.AccountBean</ejb-class>
        <persistence-type>Container</persistence-type>
        <prim-key-class>java.lang.String</prim-key-class>
        <reentrant>False</reentrant>
        <cmp-version>2.x</cmp-version>
        <abstract-schema-name>Account</abstract-schema-name>
        <cmp-field> <field-name>name</field-name> </cmp-field>
        <cmp-field> <field-name>balance</field-name> </cmp-field>
        <primkey-field>name</primkey-field>
      </entity>
   </enterprise-beans>
</ejb-jar>
```

EJB Development Process (Step 3)



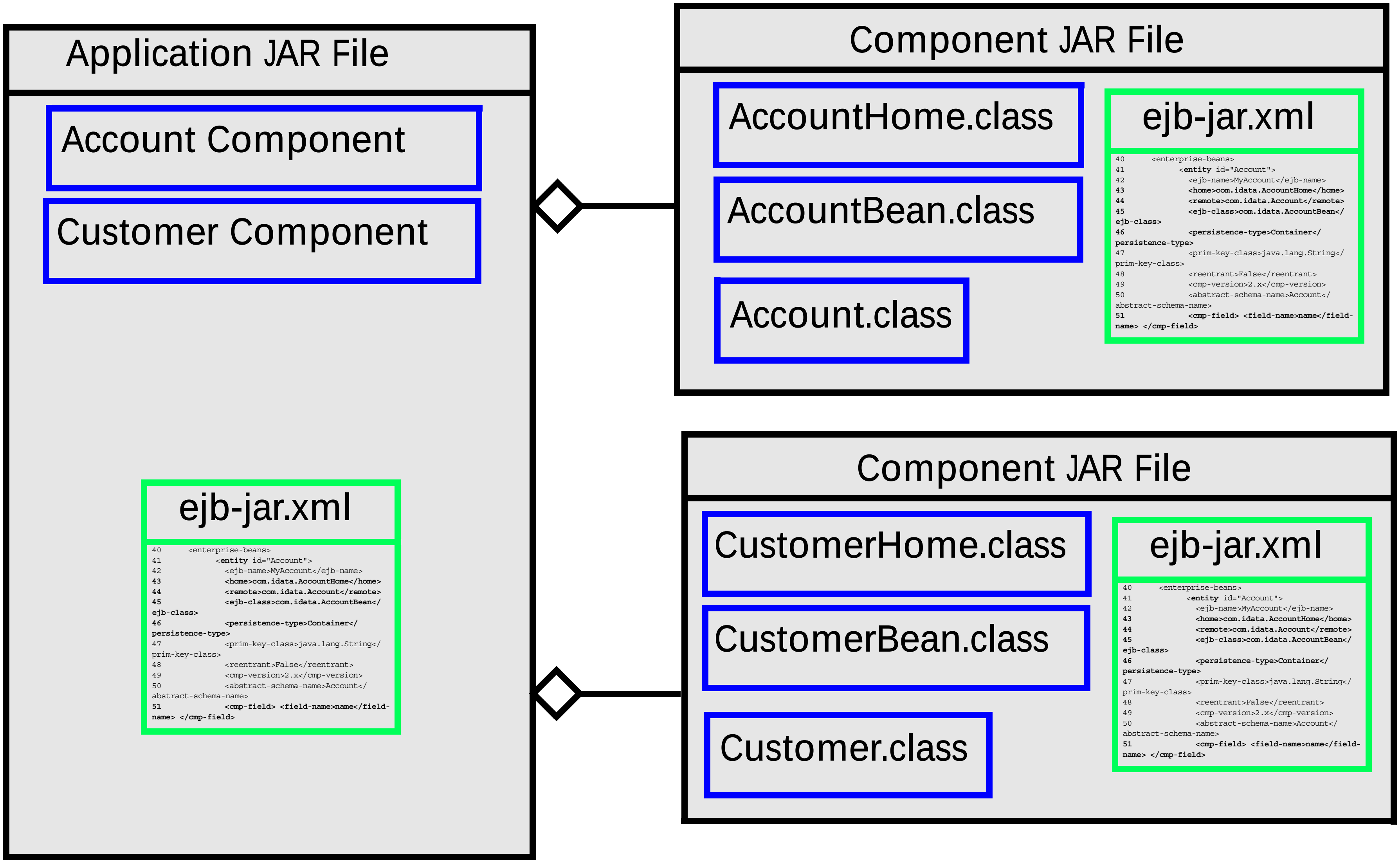
```
<enterprise-beans>
        <entity id="Account">
          <ejb-name>MyAccount</ejb-name>
          <home>com.idata.AccountHome</home>
          <remote>com.idata.Account</remote>
          <ejb-class>com.idata.AccountBean</ejb-class>
          <persistence-type>Container</persistence-type>
          <prim-key-class>java.lang.String</prim-key-class>
          <reentrant>False</reentrant>
          <cmp-version>2.x</cmp-version>
          <abstract-schema-name>Account</abstract-schema-name>
          <cmp-field> <field-name>name</field-name> </cmp-field>
```

- The component developer compiles the Java files
  - AccountHome.java
  - Account.java
  - AccountBean.java
  - Any miscellenous helper files

- The component developer packages these compiled class files along with the deployment descriptor (ejb-jar.xml) in to a *Component* JAR file

- This *Component* JAR file is given to the *Application Assembler*

EJB Application Assembly Process



Application JAR File

Account Component

Customer Component

ejb-jar.xml

```
<enterprise-beans>
        <entity id="Account">
          <ejb-name>MyAccount</ejb-name>
          <home>com.idata.AccountHome</home>
          <remote>com.idata.Account</remote>
          <ejb-class>com.idata.AccountBean</ejb-class>
          <persistence-type>Container</persistence-type>
          <prim-key-class>java.lang.String</prim-key-class>
          <reentrant>False</reentrant>
          <cmp-version>2.x</cmp-version>
          <abstract-schema-name>Account</abstract-schema-name>
          <cmp-field> <field-name>name</field-name> </cmp-field>
```

Component JAR File

AccountHome.class

AccountBean.class

Account.class

ejb-jar.xml

```
<enterprise-beans>
        <entity id="Account">
          <ejb-name>MyAccount</ejb-name>
          <home>com.idata.AccountHome</home>
          <remote>com.idata.Account</remote>
          <ejb-class>com.idata.AccountBean</ejb-class>
          <persistence-type>Container</persistence-type>
          <prim-key-class>java.lang.String</prim-key-class>
          <reentrant>False</reentrant>
          <cmp-version>2.x</cmp-version>
          <abstract-schema-name>Account</abstract-schema-name>
          <cmp-field> <field-name>name</field-name> </cmp-field>
```

Component JAR File

CustomerHome.class

CustomerBean.class

Customer.class

ejb-jar.xml

```
<enterprise-beans>
        <entity id="Account">
          <ejb-name>MyAccount</ejb-name>
          <home>com.idata.AccountHome</home>
          <remote>com.idata.Account</remote>
          <ejb-class>com.idata.AccountBean</ejb-class>
          <persistence-type>Container</persistence-type>
          <prim-key-class>java.lang.String</prim-key-class>
          <reentrant>False</reentrant>
          <cmp-version>2.x</cmp-version>
          <abstract-schema-name>Account</abstract-schema-name>
          <cmp-field> <field-name>name</field-name> </cmp-field>
```

EJB Application Assembly Process

- The application assembler collects the *Component* JAR files from potentially multiple component developers

- For an EJB application, the assembler creates
  - A single JAR file, called the *Application* (*EJB Application*) JAR file, that defines the application
  - The contents of all the relevant *Component* JAR file are copied into the *Application* JAR file
  - A deployment descriptor file, `ejb-jar.xml`, is created for the *Application* JAR file
    - The contents of the `ejb-jar.xml` files from the *Component* JAR files are copied into the `ejb-jar.xml` in the *Application* JAR file
  - The application assembler also needs to define the following:
    - Transaction attributes
    - Security roles and privileges
    - Linking of EJBs and resolving references

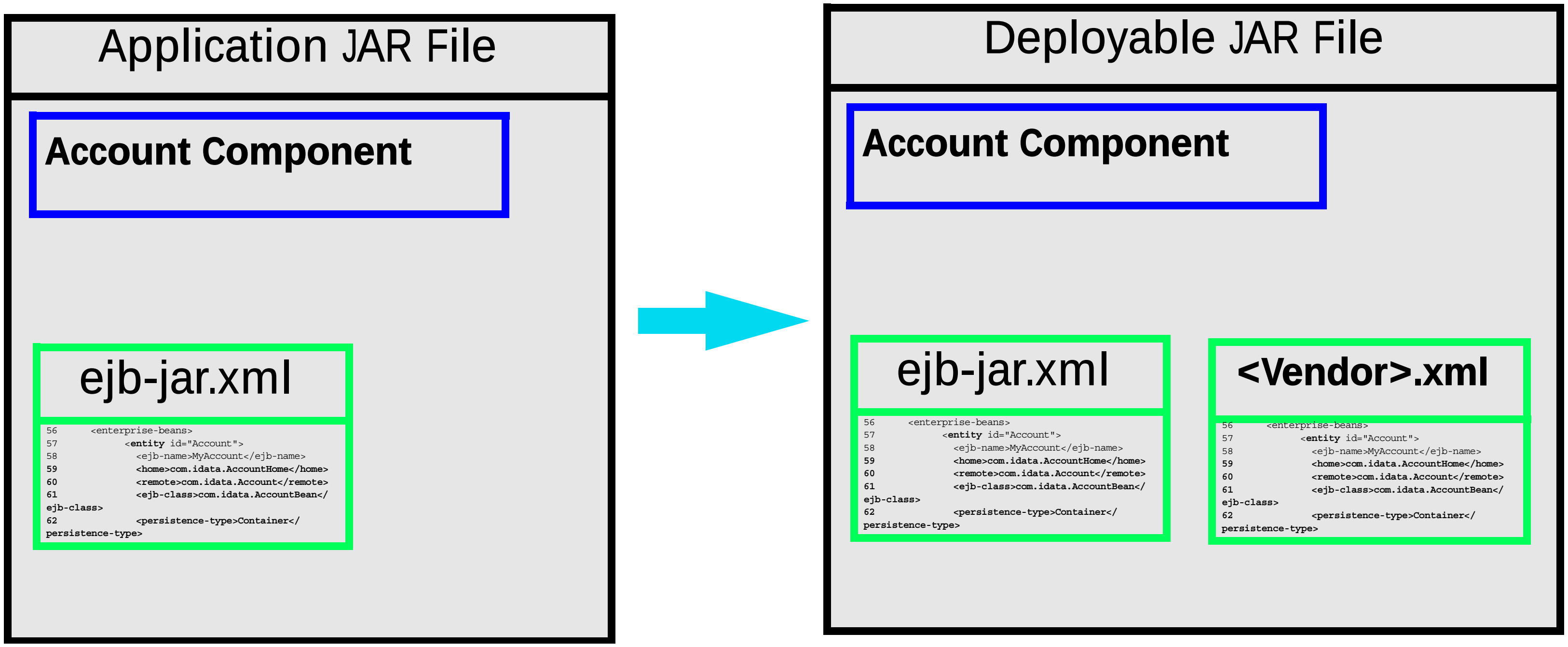
- Since we have only a single component in our example, we do not have any application assembly to perform
  - In our special case, the *Component* JAR file <u>is</u> the *Application* JAR file

EJB Application Deployment Process

- The EJB application deployer collects the *EJB Application* JAR file from the EJB application assembler
- The deployer, using EJB container-specific tools, creates another EJB container-specific deployment descriptor and stores it in the EJB *Application* JAR file which is now referred to as the ***Deployable*** JAR file
- The vendor-specific deployment descriptor is used to define the following mappings:
  - Entity beans abstract schema to tables or relations in the database
  - CMP fields in the entity beans to columns in the database tables
  - References to resources (DataSources, Message Queues and Topics) in the Application JAR to real resource objects
    - The administrator should have already created and configured these resource objects
  - Security role names to actual security groups or users in the operating environment
    - The administrator should have already created and configured the users and groups

EJB Application Deployment Process

- In the vendor-specifc deployment descriptor, the deployer defines the names to be used to register the EJB *Home* objects in a **JNDI**-based naming server
  - These names are referred to **JNDI** names
  - For our example application containing the sole component, Account, let us say the deployer chooses **CitiBank** as the name of the Account bean's *Home* object
  - The administrator should have already created and configured the JNDI-based naming service

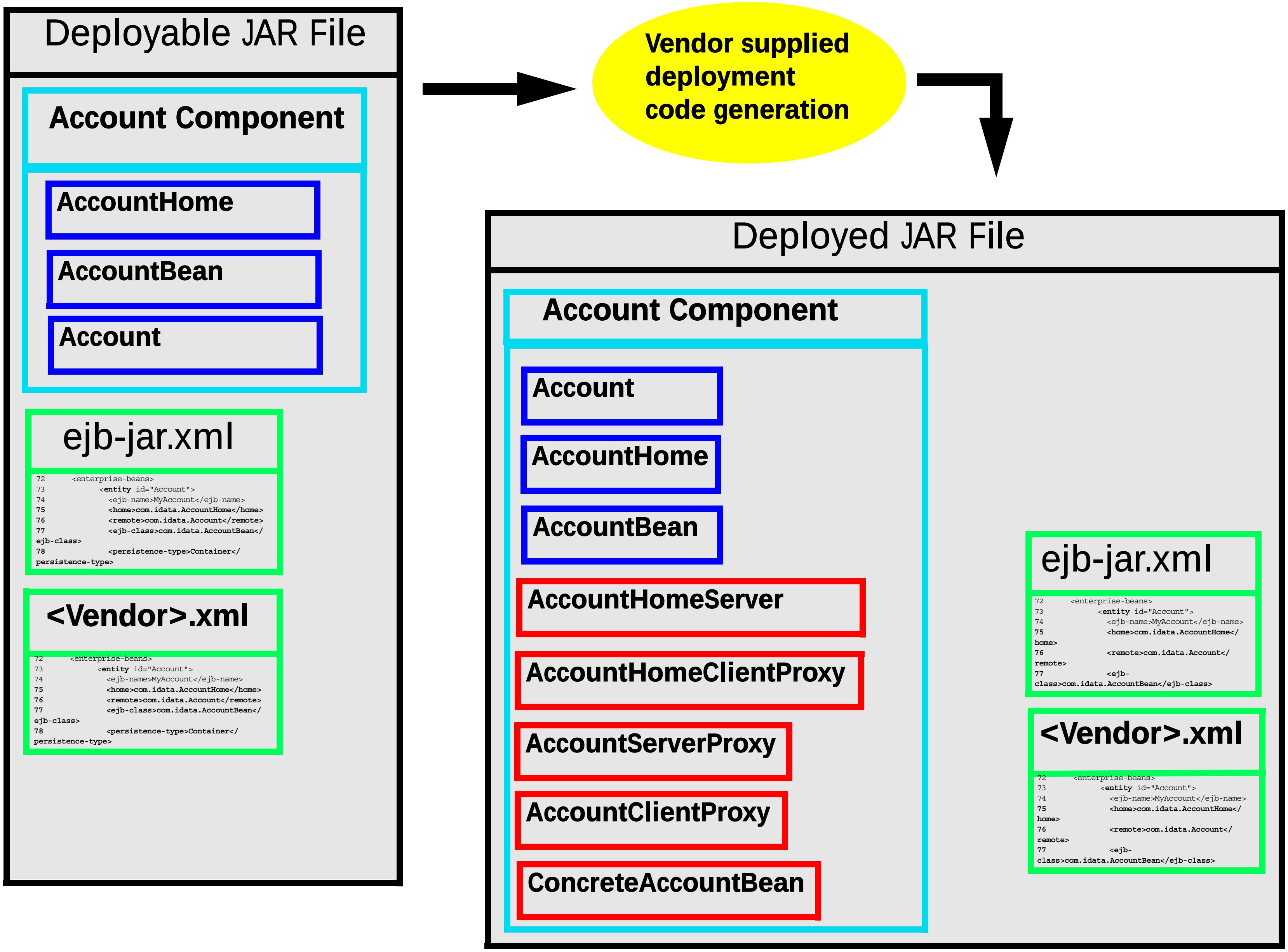


```
      <enterprise-beans>
            <entity id="Account">
              <ejb-name>MyAccount</ejb-name>
              <home>com.idata.AccountHome</home>
              <remote>com.idata.Account</remote>
              <ejb-class>com.idata.AccountBean</
ejb-class>
              <persistence-type>Container</
persistence-type>
```

```
      <enterprise-beans>
            <entity id="Account">
              <ejb-name>MyAccount</ejb-name>
              <home>com.idata.AccountHome</home>
              <remote>com.idata.Account</remote>
              <ejb-class>com.idata.AccountBean</
ejb-class>
              <persistence-type>Container</
persistence-type>
```

```
      <enterprise-beans>
            <entity id="Account">
              <ejb-name>MyAccount</ejb-name>
              <home>com.idata.AccountHome</home>
              <remote>com.idata.Account</remote>
              <ejb-class>com.idata.AccountBean</
ejb-class>
              <persistence-type>Container</
persistence-type>
```

EJB Application Deployment Process



```
      <enterprise-beans>
            <entity id="Account">
              <ejb-name>MyAccount</ejb-name>
              <home>com.idata.AccountHome</home>
              <remote>com.idata.Account</remote>
              <ejb-class>com.idata.AccountBean</ejb-class>
              <persistence-type>Container</persistence-type>
```

```
      <enterprise-beans>
            <entity id="Account">
              <ejb-name>MyAccount</ejb-name>
              <home>com.idata.AccountHome</home>
              <remote>com.idata.Account</remote>
              <ejb-class>com.idata.AccountBean</ejb-class>
              <persistence-type>Container</persistence-type>
```

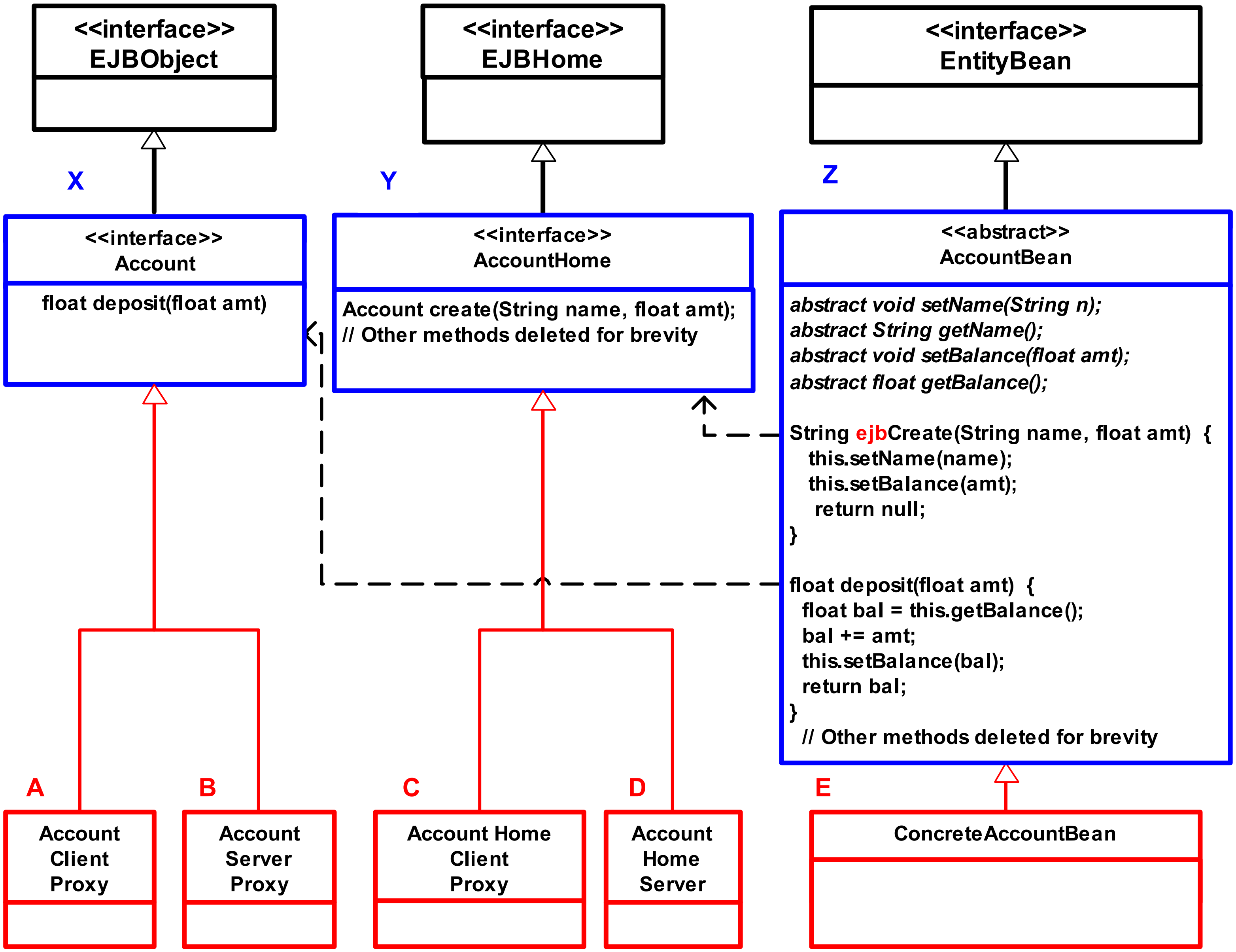
```
      <enterprise-beans>
            <entity id="Account">
              <ejb-name>MyAccount</ejb-name>
              <home>com.idata.AccountHome</home>
              <remote>com.idata.Account</remote>
              <ejb-class>com.idata.AccountBean</ejb-class>
```

```
      <enterprise-beans>
            <entity id="Account">
              <ejb-name>MyAccount</ejb-name>
              <home>com.idata.AccountHome</home>
              <remote>com.idata.Account</remote>
              <ejb-class>com.idata.AccountBean</ejb-class>
```

EJB Application Deployment Process

- The deployer, using a EJB container-specific deployment tool and the ***Deployable*** JAR file as input, generates another JAR file called the ***Deployed*** JAR file
- The ***Deployed*** JAR file contains the deployment-specific container classes
- This JAR file is **<u>not</u>** portable
- This step is also referred to as *EJB Compilation*
- This JAR file is loaded on to the EJB application server
- Now the application is ready to be executed

EJB Application Deployment Process



```
float deposit(float amt)
```

```
Account create(String name, float amt);
// Other methods deleted for brevity
```

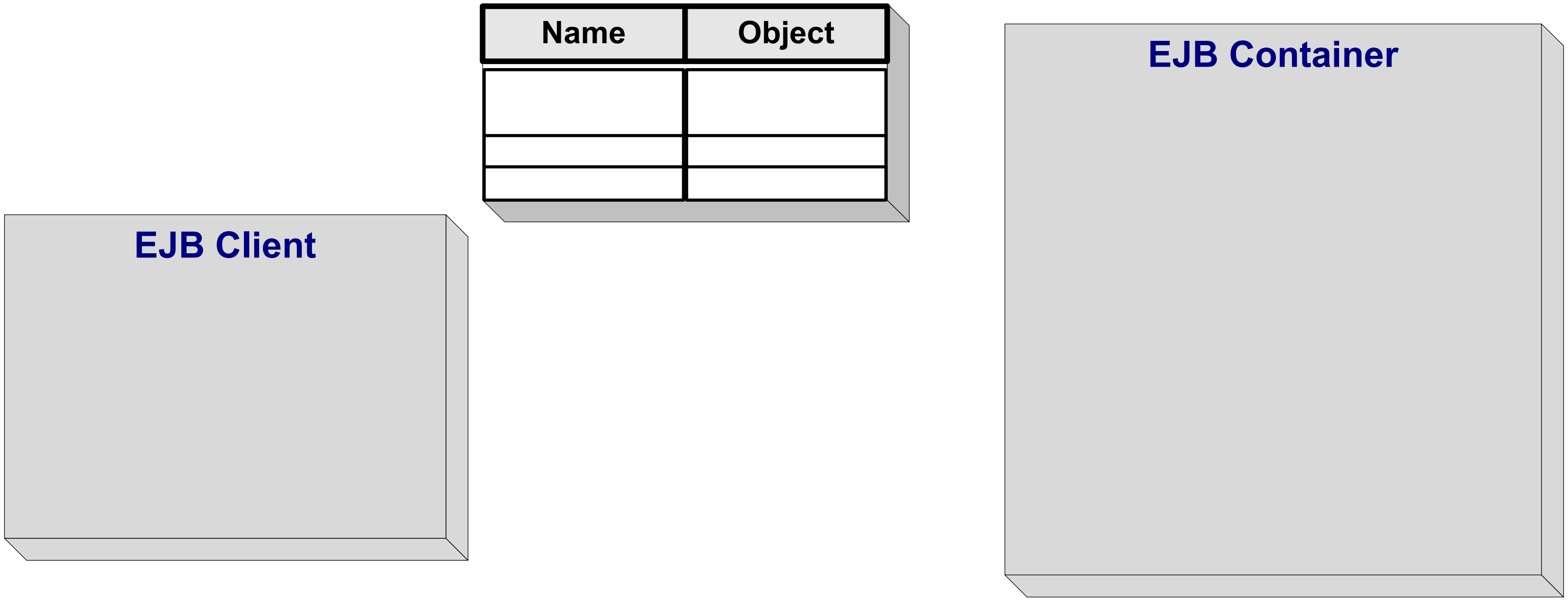
```
abstract void setName(String n);
abstract String getName();
abstract void setBalance(float amt);
abstract float getBalance();

String ejbCreate(String name, float amt)  {
   this.setName(name);
   this.setBalance(amt);
    return null;
}

float deposit(float amt)  {
  float bal = this.getBalance();
  bal += amt;
  this.setBalance(bal);
  return bal;
}
  // Other methods deleted for brevity
```

How do Enterprise JavaBeans Work?

In this section, we shall take a look at the following

- EJB Application deployment
- EJB container internals
- Interaction between the EJB component and the EJB Container
- Interactions between the client and the EJB container

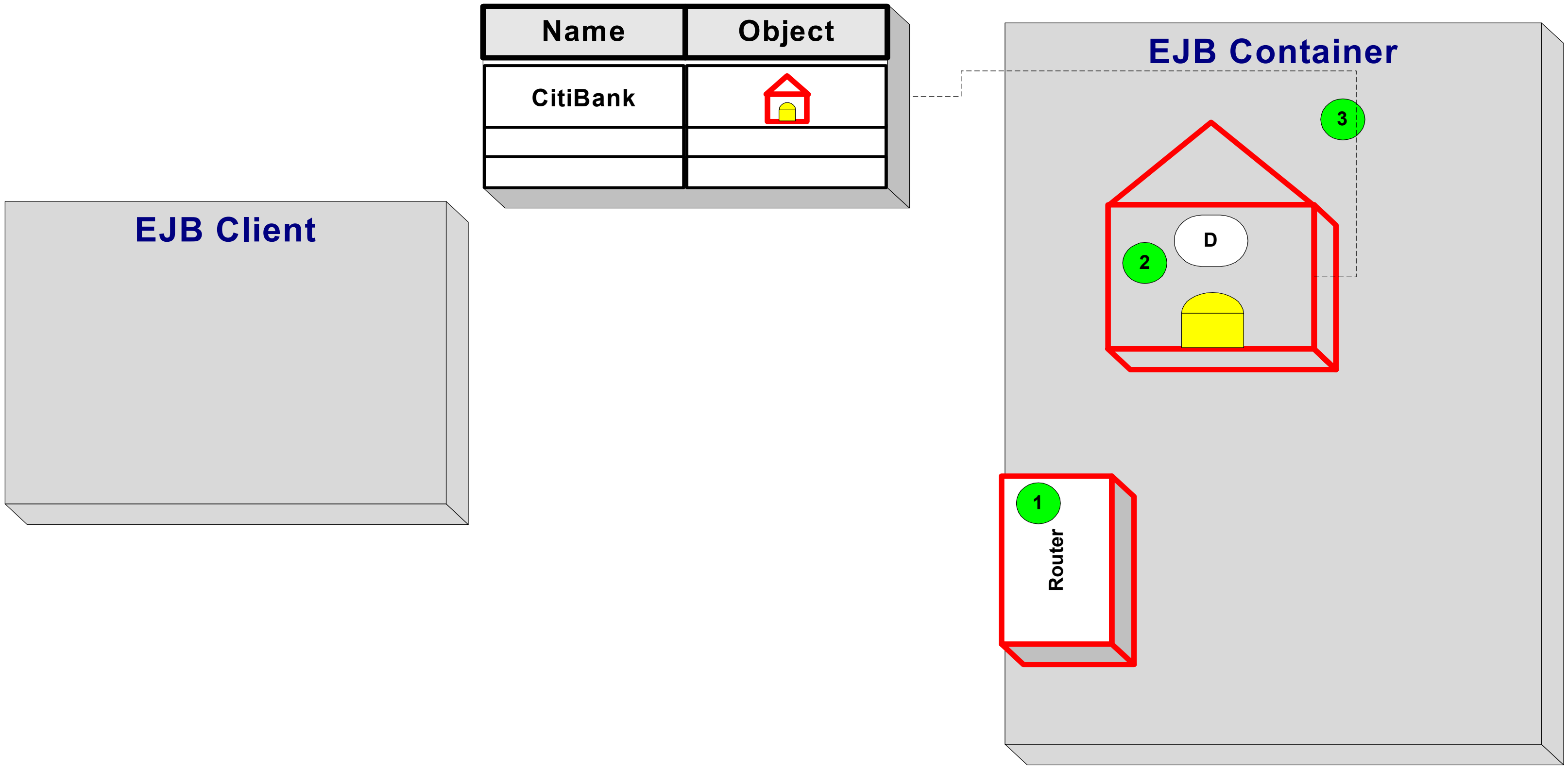
EJB Execution



Name	Object

- ◆ Before the application is deployed, we have three processes
  - • The EJB container
    - Running within an application server (not shown)
  - • The naming service
    - Initially empty
  - • The client process

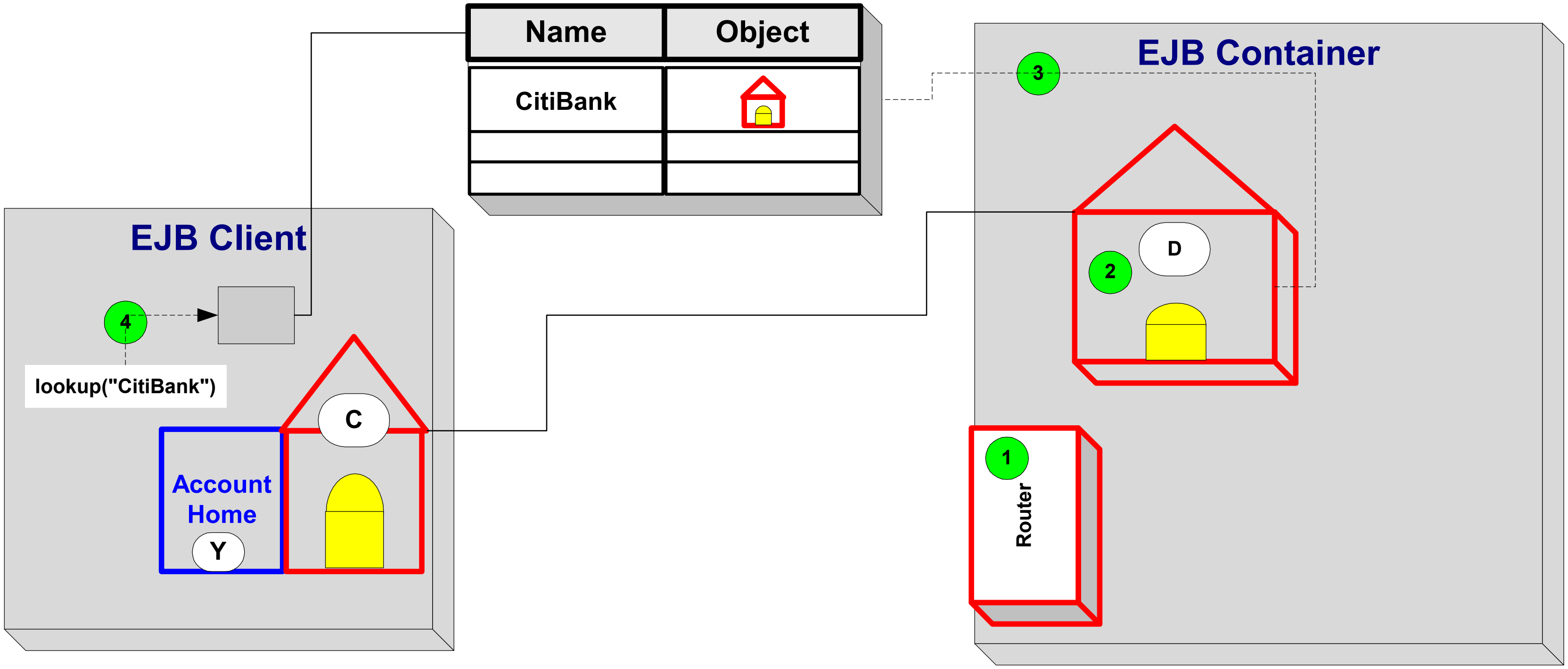
EJB Execution



Name	Object
CitiBank	🏠

1. A router object is instantiated in the EJB container (or server)
2. The AccountHomeServer object is instantiated in the EJB container
3. The AccountHomeServer object is regsitered (JNDI bound) in the naming service using the name specified by the deployer
   - CitiBank

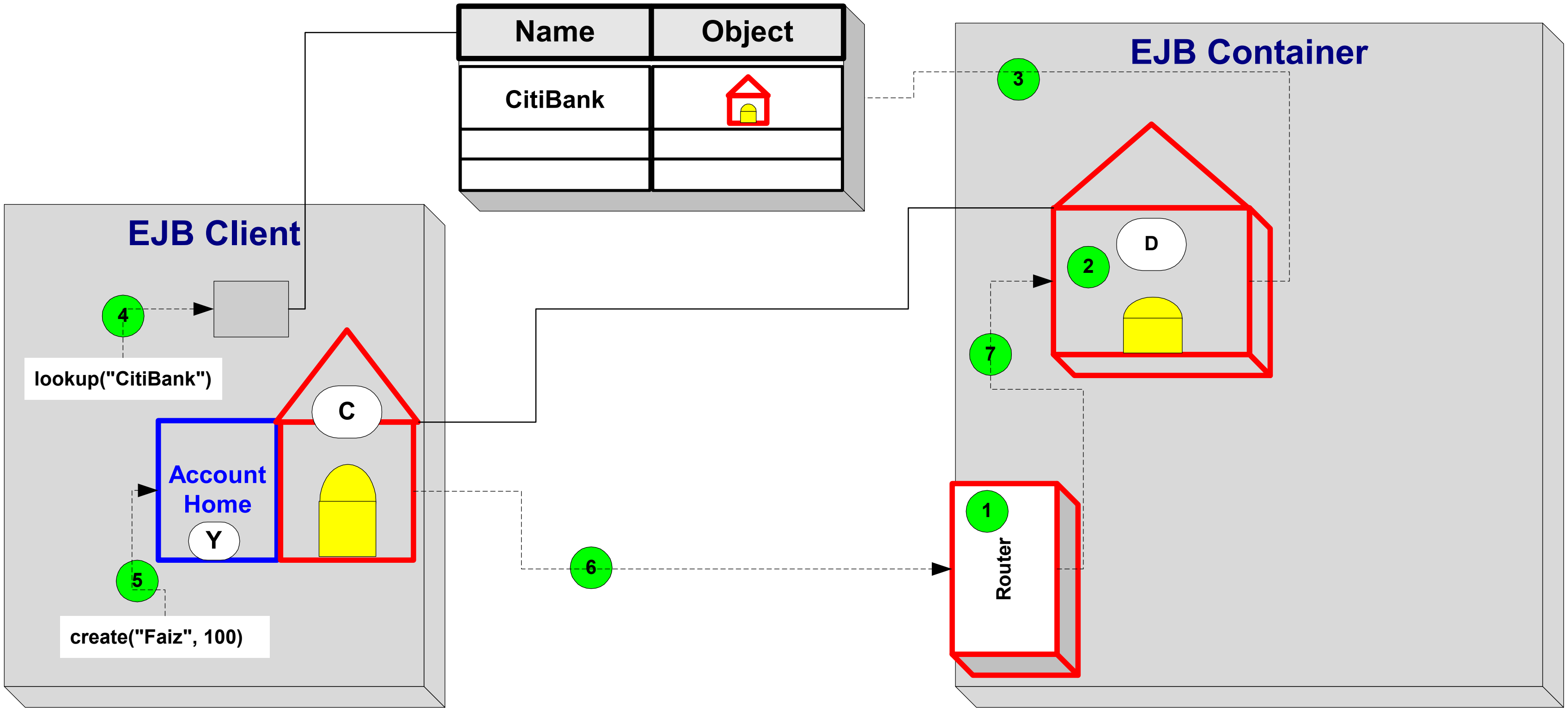
EJB Execution



4. Using the JNDI API, the client connects to the naming service and performs a lookup using the JNDI name of the home object (CitiBank)

- It receives an object of type `AccountHomeClientProxy` and the client narrows (casts) it to the type `AccountHome`
- This object is a proxy to the server-side home object
- What kind of proxy is `AccountHomeClientProxy` ?

EJB Execution



Name	Object
CitiBank	🏠

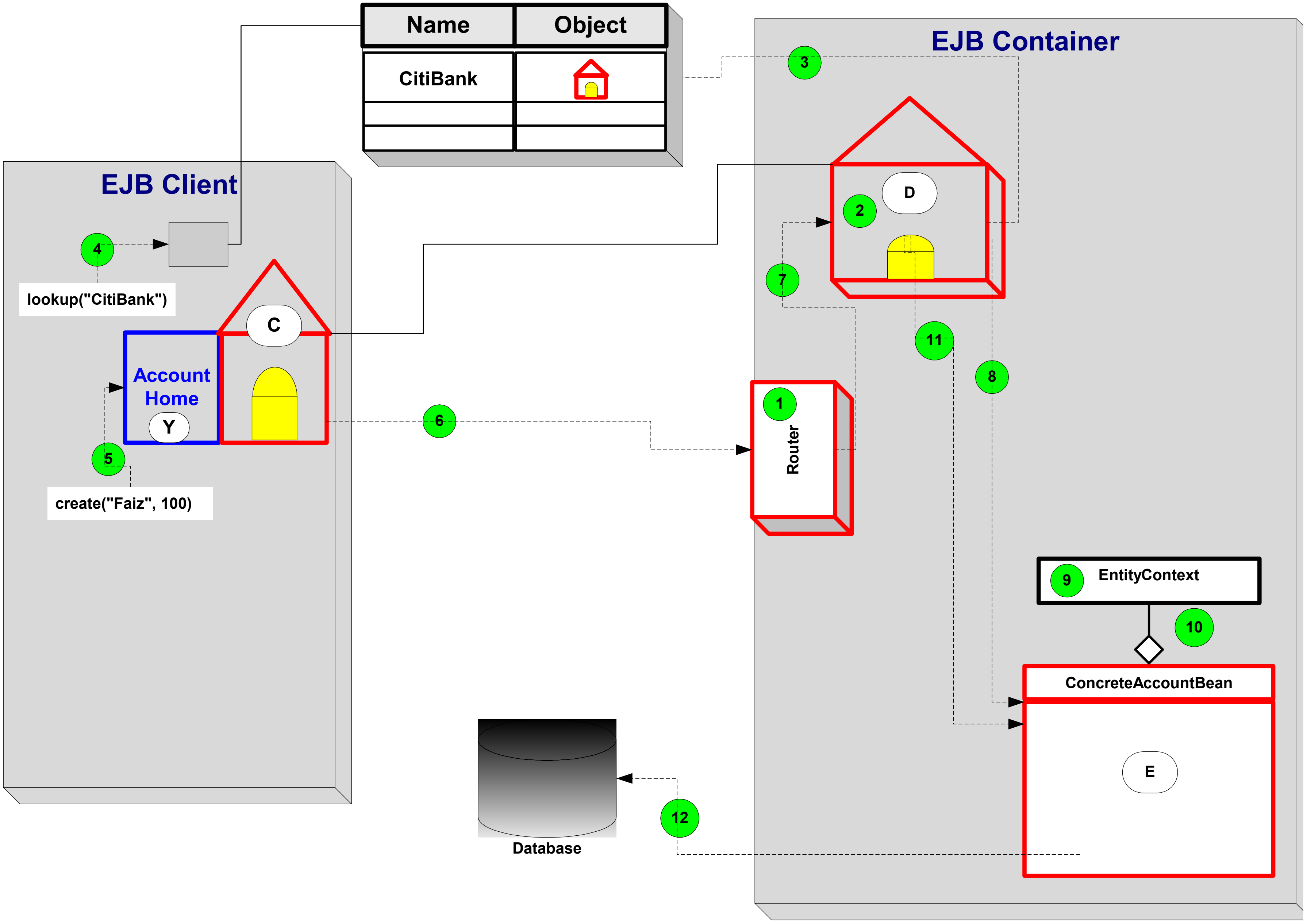
5. The client invokes the factory method on the client proxy

   - create("Faiz", 100)

6. The client proxy sends a network message, containing the method invocation request, to the router on the server/container

7. The router, receives the request, parses it and dispatches the call, create("Faiz", 100), to the server-side home object, `AccountHomeServer`

EJB Execution



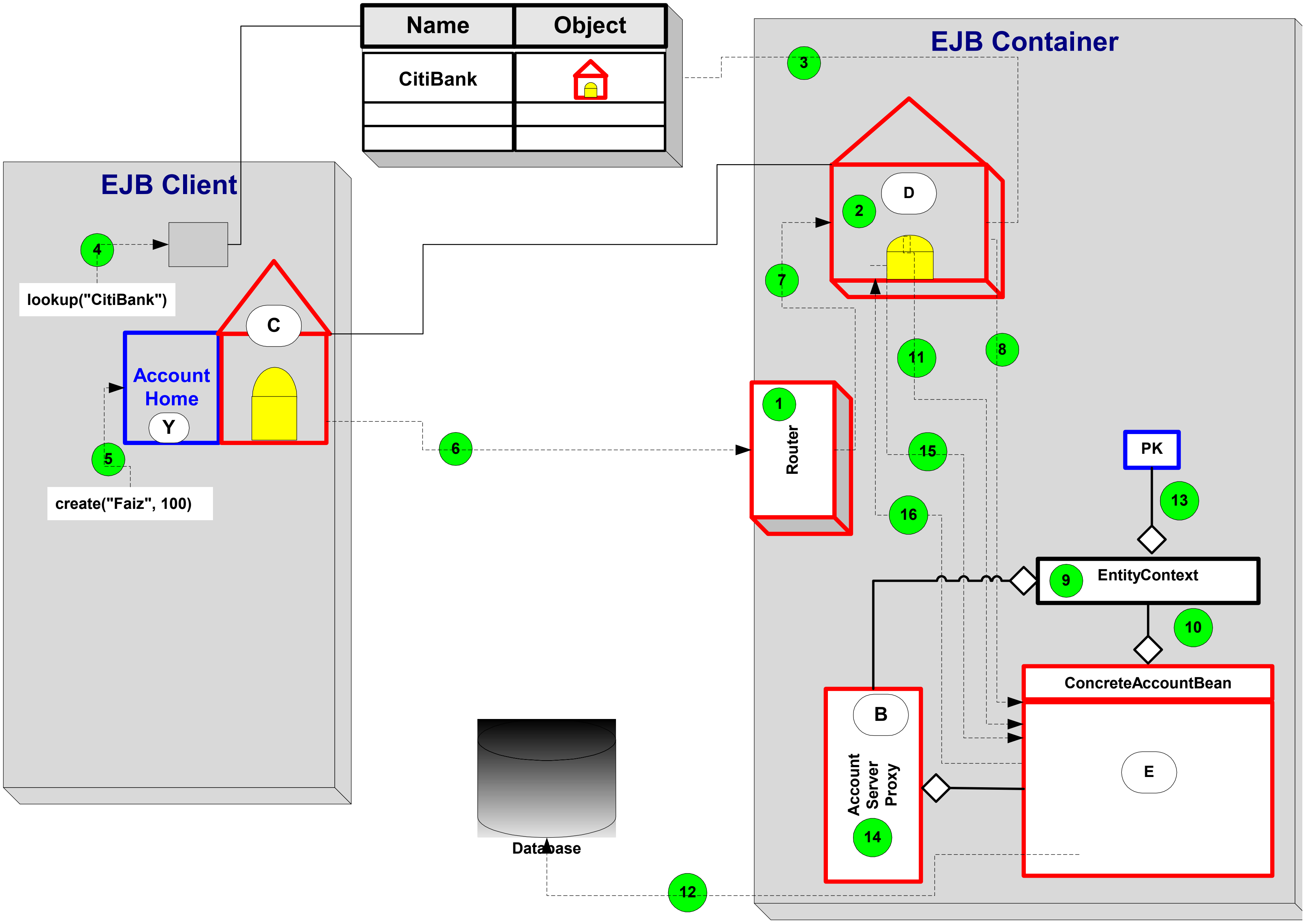
Name	Object
CitiBank	

EJB Execution

- The mechanically generated method, `create(String n, float amount)`, on the server-side object, `AccountHomeServer` performs the following operations:

8. Instantiates the concrete implementation of the EJB class, `ConcreteAccountBean`
9. Instantiates a container object of type `javax.ejb.EntityContext`
   - This context object may be used by the bean to access container services such as transactions and security
10. Assigns the context object to the concrete bean using the method `setEntityContext(javax.ejb.EntityContext)` on the bean
    - This method must be implemented on the abstract class `AccountBean`
11. Invokes the method `ejbCreate("Faiz", 100)` on the concrete bean
12. Reads the contents of the concrete bean and performs a database insert

EJB Execution



Name	Object
CitiBank	⌂

EJB Client

4

lookup("CitiBank")

C

Account Home

Y

5

create("Faiz", 100)

6

EJB Container

3

D

2

7

11

8

1

Router

15

16

PK

13

9

EntityContext

10

ConcreteAccountBean

B

Account Server Proxy

14

E

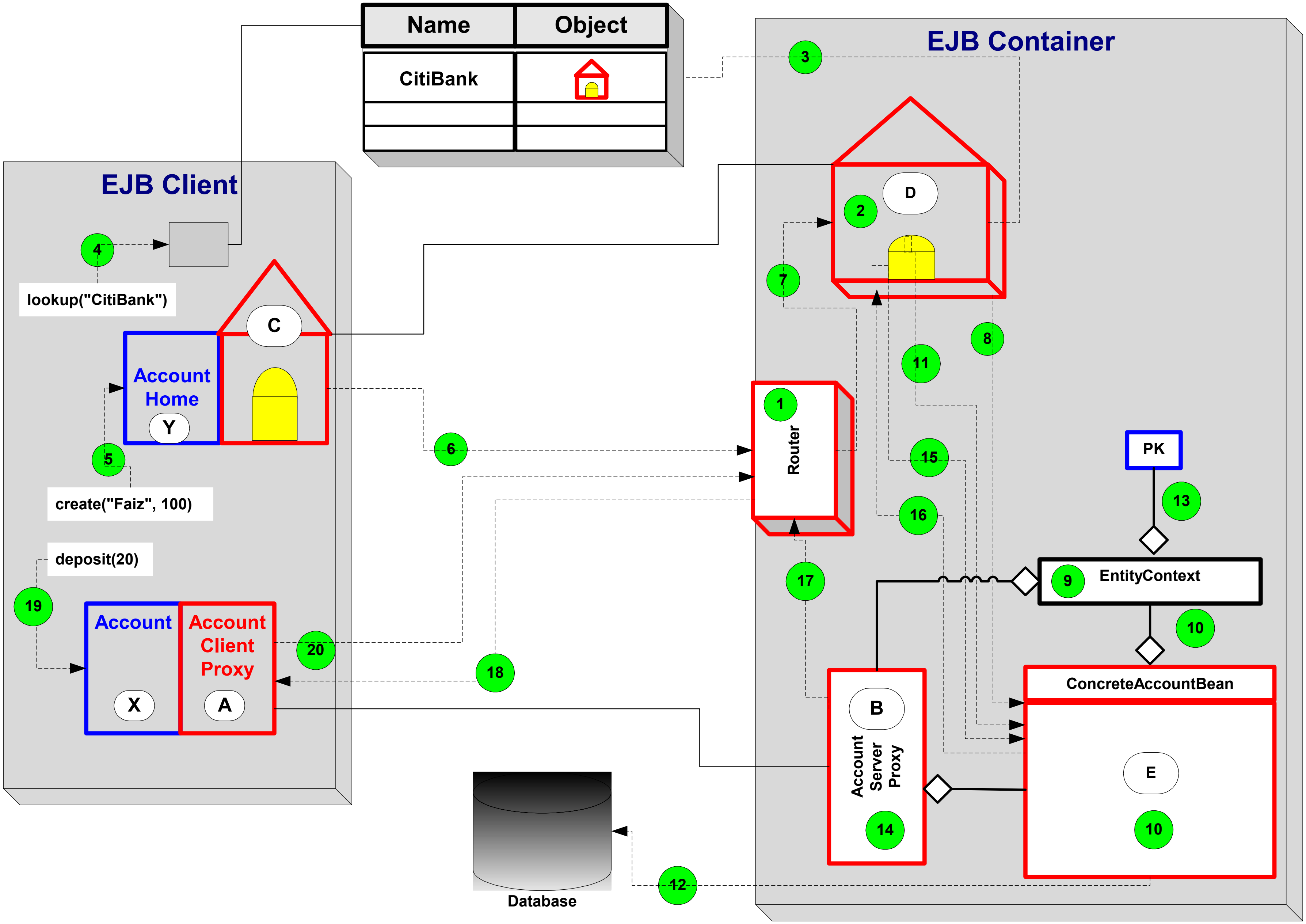
Database

12

EJB Execution

13. Instantiates the primary key object and assigns it to the entity context object
14. Instantiates the class `AccountServerProxy` and connects that object to the concrete bean and the entity context
15. Invokes the method `ejbPostCreate("Faiz", 100)` on the concrete bean
16. Returns the object of type `AccountServerProxy`

EJB Execution



Name	Object
CitiBank	

EJB Client

4

lookup("CitiBank")

Account Home

C

Y

5

create("Faiz", 100)

deposit(20)

19

Account

Account Client Proxy

X

A

20

6

18

EJB Container

3

D

2

7

8

11

1

Router

15

16

17

PK

13

9

EntityContext

10

ConcreteAccountBean

B

Account Server Proxy

14

E

10

12

Database

EJB Execution

17. The router, which had invoked the method `create("Faiz", 100)` on the object of type `AccountHomeServer`, receives an object of type `AccountServerProxy`

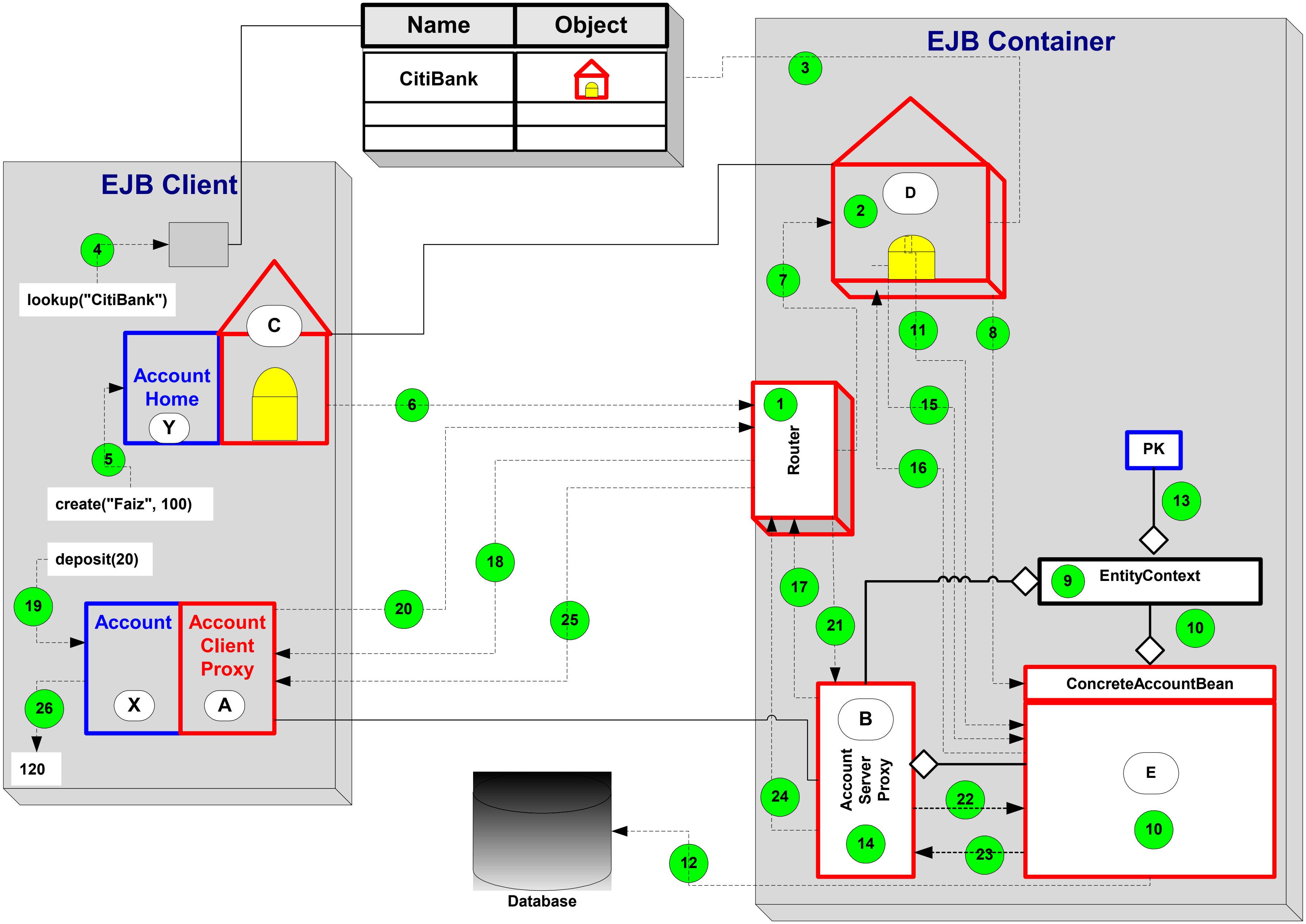
18. Corresponding to an instance of the class `AccountServerProxy`, the router constructs an object of type `AccountClientProxy`, serializes it and sends it back as a response to the client

    - This is the return value for the client invoked method, `create("Faiz", 100)`, in step 5

19. Since this object, `AccountClientProxy`, implements the interface `Account`, the client can now invoke the business method, `deposit(20)`, on this ***remote reference***

20. The `AccountClientProxy` constructs and sends a request message on the network to the router

EJB Execution



Name	Object
CitiBank	🏠

EJB Client

lookup("CitiBank")

Account Home

create("Faiz", 100)

deposit(20)

Account

Account Client Proxy

120

EJB Container

Router

PK

EntityContext

ConcreteAccountBean

Account Server Proxy

Database

EJB Execution

21. The router receives the message, parses it and dispatches (i.e. invokes the method `deposit(20)`) it to the corresponding `AccountServerProxy` object
22. The `AccountServerProxy` dispatches (i.e. invokes the method `deposit(20)`) it to the concrete bean
23. The component author implemented method, `deposit(float amount)` is executed and a value (`float` with a value of 120) is returned
24. The `AccountServerProxy` passes the return value back to the router as a return value of the method invoked in step 21.
25. The router constructs a network response message encapsulating the return value (`float` with a value of 120) and sends it back to the `AccountClientProxy`
26. The `AccountClientProxy` receives the network response, extracts the return value, converts it to `float` and returns it to the caller who had invoked this method in step 19
    - The client should receive a `float` of value 120

Some Questions

- What kind of proxy is `AccountHomeClientProxy`?
- What kind of proxy is `AccountClientProxy`?
- What kind of proxy is `AccountServerProxy`?

Overview of Java Message Service

Enterprise Messaging System

- Enterprise messaging systems allow two or more applications to exchange information in the form of messages

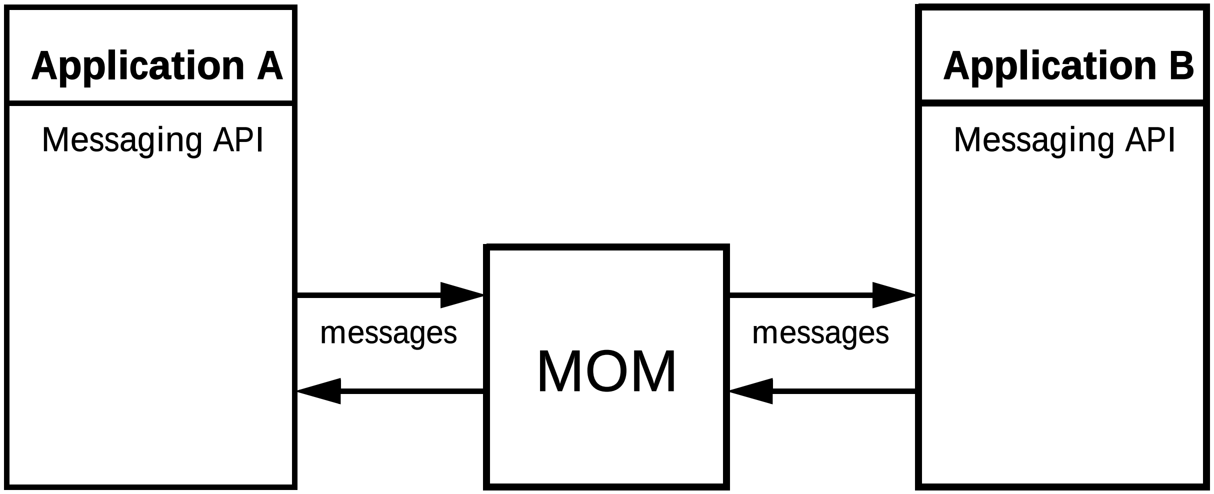
- A ***message*** is a self-contained package of business data and network routing header information.

- Applications exchange messages through virtual channels called *destinations*.
  - When sending, a message is addressed to a destination, not a specific application
  - Any application that is interested in that destination may receive that message
  - This decouples the sender and receiver

- The entity that handles these virtual channels is called ***Message-Oriented Middleware* (*MOM*)**.

Message-Oriented Middleware

- In MOM, messages are delivered *asynchronously* from one system to others.
  - Sender is not required to wait for the message to be received or handled by the recipient(s).
- Asynchronous messages are treated as autonomous units:
  - Carries all the data and state needed by the business logic that processes it
- Applications use some messaging API to construct a message and then hand it off to MOM for delivery to one or more recipients.



Messaging API

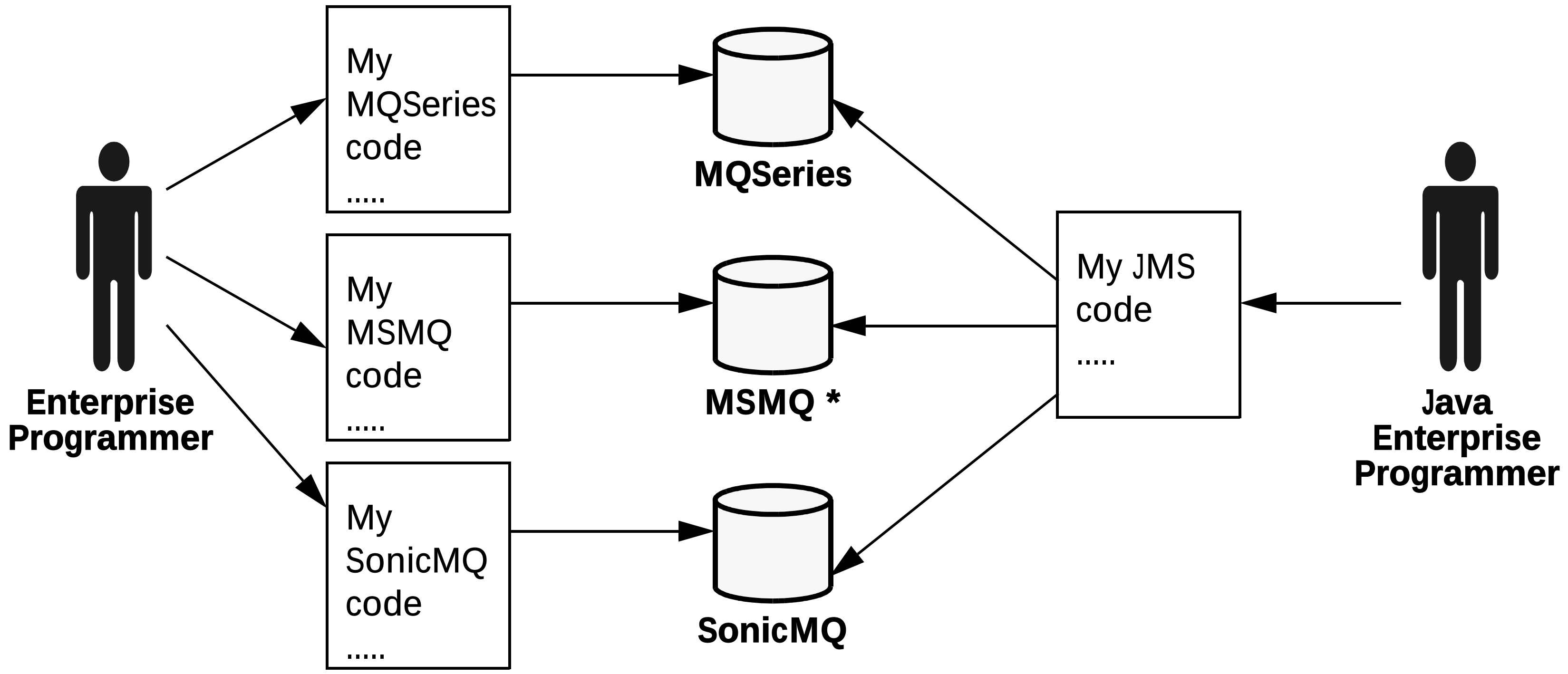
- All MOM vendors provide application developers an API for sending and receiving messages
  - Basic semantics of these APIs are the same
  - Implementations may differ in network protocols, routing, adminiatration facilities, etc.
- This similarity in APIs makes the Java Messaging Service (JMS) possible
- Different MOM vendors may support different functionalities
  - Broadcasting a message to many destinations vs. sending to a single destination
  - Insurance of delivery ranges from *best efforts* to *guaranteed*
- Popular MOM products include:
  - IBM MQSeries
  - Progress SonicMQ
  - Microsoft MSMQ
  - Bea WebLogic Server
  - Sun Java Message Queue
  - OpenJMS (open source)

Java Message Service (JMS)

- JMS is a vendor-independent messaging API
  - Defines a common set of enterprise messaging concepts and facilities
  - Not a union of existing vendor-specific APIs, nor an intersection of them
  - Minimizes the set of concepts a Java programmer must learn to use MOM products
  - Maximizes the portability of messaging applications
- JMS is not a messaging system itself; it's a set of interfaces and classes for accessing messaging systems.
- If a MOM vendor provides an implementation of these interfaces and classes, Java programmers can then use JMS API to access this vendor's messaging system.
  - This MOM product then becomes a ***JMS Provider***.

Benefits of Using JMS

- Reuse the same API to access many different messaging systems



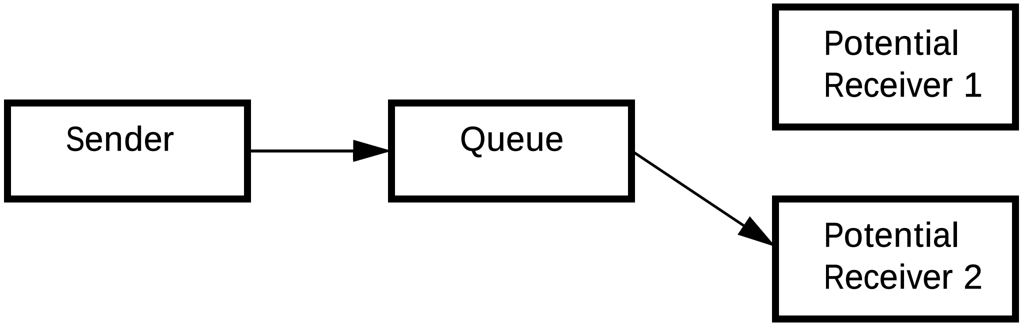
* It is not expected that Microsoft will support JMS.

JMS Messaging Styles

- JMS messaging is peer-to-peer
  - All users of JMS are referred to generically as ***JMS clients***.
  - A JMS client that produces a message is called a ***producer***.
  - A JMS client that receives a message is called a ***consumer***.
  - A JMS client can be both a producer and a consumer.
- JMS offers two messaging styles (also called ***messaging domains***):
  - ***Point-to-point (PTP or P2P)*** queuing
    - One-to-one delivery of messages; messages are inserted and removed from a queue.
  - ***Publish-and-subscribe (Pub/Sub)***
    - One-to-many broadcast of messages

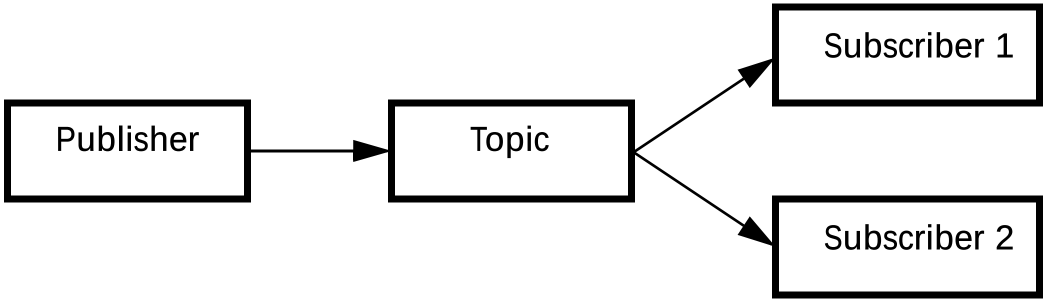
Point-To-Point

- One-to-one between producer and consumer
- *Producers* are referred to as *Senders*
- *Consumers* are referred to as *Receivers*
- The *Destination* is referred to as *Queue*
- Typically implemented by a queue
  - Multiple *senders* can place messages in it
  - Multiple *receivers* can consume messages from it
  - Only one *receiver* gets to consume a message
    - Once a message has been consumed by a *receiver*, it is no longer available to other *receivers*



Publish-and-Subscribe

- One producer can send a message to many consumers
- Producer *publishes* (hence also called *publishers*) messages to a *Topic*
- Consumers *subscribe* (hence also called *subscribers*) to a Topic
- Any messages addressed to a topic are delivered to all the topic's subscribers



Message-Driven Beans

Message-driven Beans

- A **message-driven bean** (MDB) is
  - An asynchronous message consumer
  - A **`MessageListener`** object on a JMS destination
  - Invoked by the container as a result of the arrival of a JMS message
- A message-driven bean has neither a home nor a remote interface
- A message-driven bean has no conversational state
- A message-driven bean is anonymous

Association of MDB With JMS Destination

- Bean Provider uses `message-driven-destination` deployment descriptor element to advise Deployer which JMS destination this MDB is to be associated with.

- An MDB is associated with a JMS Destination (`Queue` or `Topic`) when the bean is deployed.

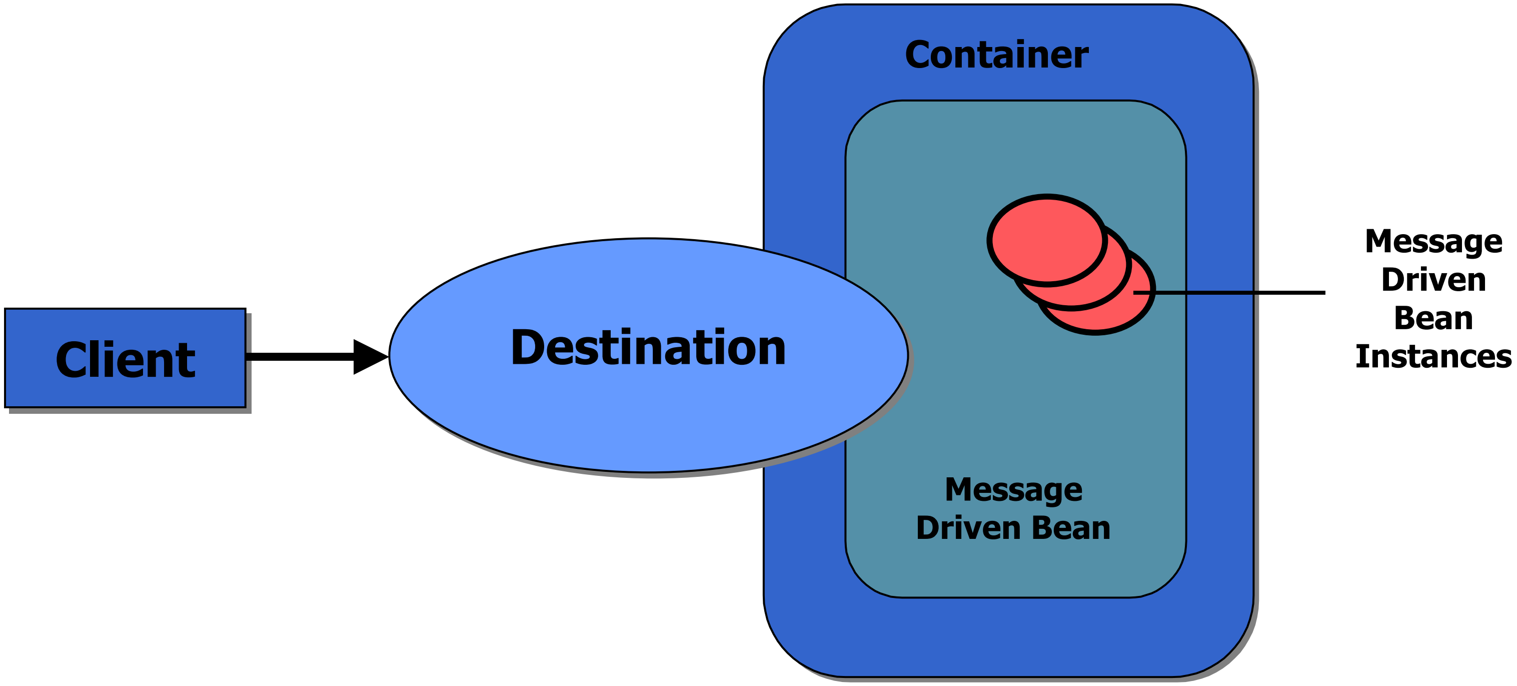
Client View of an MDB

- To a client, an MDB is simply a JMS message consumer (`MessageListener` for a JMS Destination).
- Client sends message to the JMS Destination.
- Clients locate a JMS Destination (for which an MDB is a listener) by using JNDI:

```
Context initialContext = new InitialContext();
Queue stockOrder = (javax.jms.Queue)initialContext.
    lookup(“java:com/env/jms/stockOrder”);
```

Client View of an MDB

- Client view of MDBs deployed in a container:



Message-driven Bean Class

- All message-driven beans must implement the `MessageDrivenBean` and the `MessageListener` interfaces:

```
package javax.ejb;
public interface MessageDrivenBean {
    public void ejbRemove();
    public void setMessageDrivenContext
        (MessageDrivenContext c);
}

package javax.jms;
public interface MessageListener {
    public onMessage(Message msg);
}
```

- In addition, they must also implement an `ejbCreate()` method

onMessage() Method

- The `onMessage()` method is called by the bean's container when a message has arrived for the bean to service.
- The `onMessage()` method contains the logic that handles the processing of the message
  - Typically code to transform (adapt) types from the messaging domain to the business domain
- The `onMessage()` method has a single argument, the incoming message

XML Overview

What Is XML?

- ◆ Extensible Markup Language (XML) is a markup language similar to HTML, but XML allows you to make your own tags

- ◆ XML has many benefits over previous data formats.
  - It is flexible.
  - It frees programmers from writing complex parsers.
  - It ties in nicely with Java technology.
  - It describes the kind of data, not how to display it.

Basic Pieces of an XML Document

- Elements – Contain data or other elements.
- Attributes – Provide specific information about an element.
- Entities – Special characters that you use to avoid clashes with XML syntax.
  - For example, ’<’ is XML syntax, but you could use the entity: “&lt;” (It stands for less than.)

A Simple XML Example

- Storing information about books in XML:

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- This is a comment in XML. (Just like HTML)-->
<book>
<title>Catcher in the Rye</title>
<author>JD Salinger</author>
<price unit="Dollars">7.95</price>
</book>
```

<book> ... *</book>* is an element.

Unit is an attribute on the price element.

You can write an empty element two ways: <foo></foo> or <foo/>.

XML Syntax – Well-Formed XML

- Every tag must be closed.
- Tags must be nested properly.
  - -The following is incorrect:

```
<book>
<title>
</book>
</title>
```

  - -The following is correct:

```
<book>
<title>
</title>
</book>
```

XML Syntax – Valid XML

- Valid XML:
    - Is well formed
    - Is more strict than well formed XML
    - Contains a Document Type Definition (DTD) that explains the grammar that should be followed in the document
- Given a DTD, you can verify if an XML document is grammatically correct.

DTD

- A DTD for the book example:

```
<?xml version=’1.0’?>
<!DOCTYPE book [
<!ELEMENT book (title, author, price)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ATTLIST price unit (dollars|pounds) "dollars">
]>
```

- This DTD says that:
  - A book must contain three elements: title, author, and price.
  - The ATTLIST line describes the unit attribute on price. Valid values for unit are “dollars” and “pounds”; the default is “dollars.”

- Given an XML document and a DTD, a validating parser can check the syntax to verify the right tags are in the right place and that they contain the right type of data.

XML Parsers

- XML parsers do two major tasks:
  - Parse the document to check for correctness.
  - Provide a way for a programmer to access the data in the document.
- You must choose between two kinds of XML parsers: validating and non-validating.
  - A validating parser can parse valid XML documents.
  - A non-validating parser can parse well formed XML documents.
- Next, you must choose between a tree-based parser and an event-driven one

Overview of Web Services Concepts

This section provides overview of some fundamental concepts in Web services:

- What are Web services
- Web service architecture
- SOAP
- WSDL
- UDDI

What are Web Services

- A *Web service* is a set of related application functions that can be programmatically invoked over the Internet
- Applications can mix and match Web services to perform complex tasks with minimal programming
  - Example: An application can use airline reservation service, credit check service, etc. to package into a complete travel service

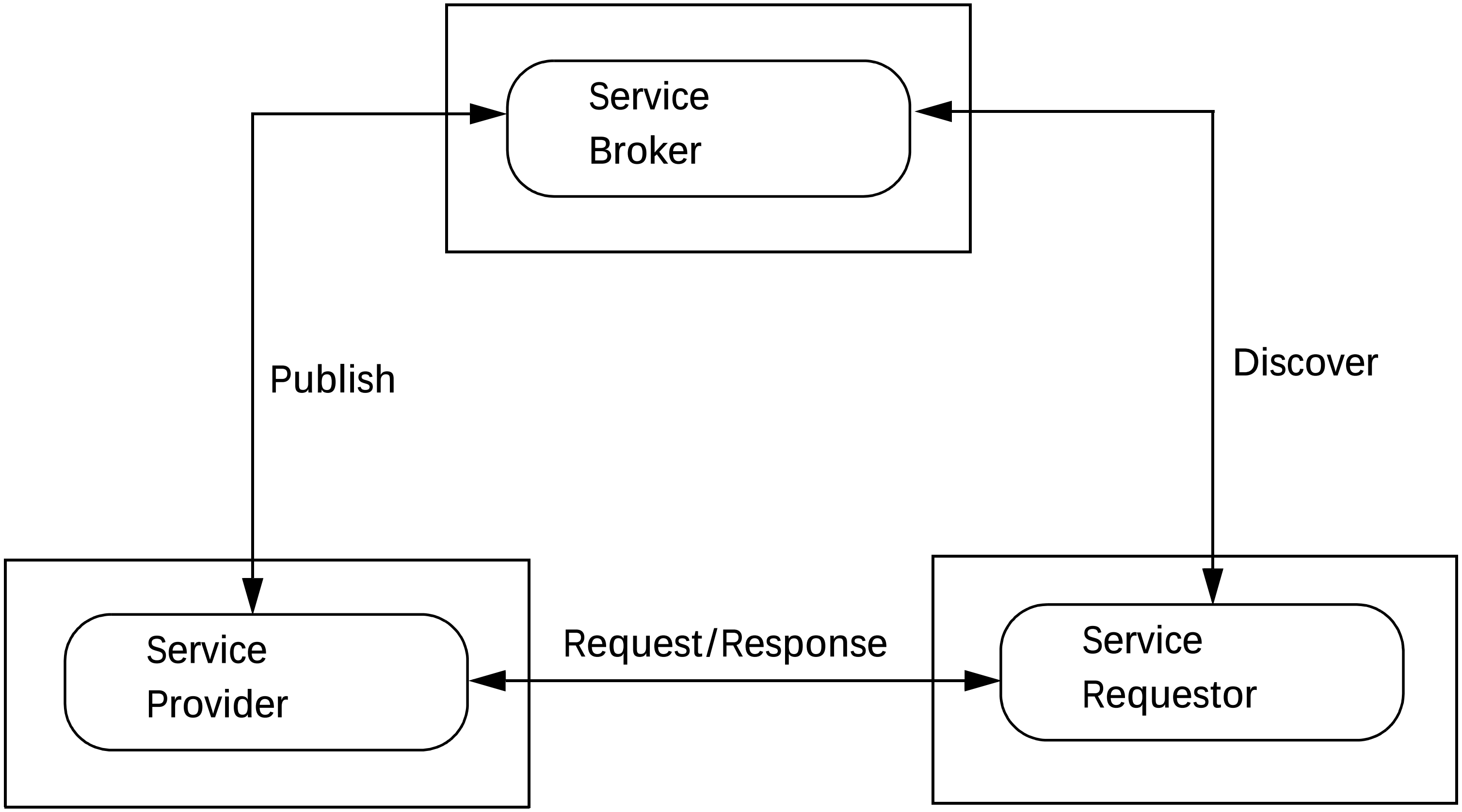
Categories of Web services

- Business information
  - Use Web serivices to share information with customers or other businesses
  - Example: stock quotes, news, etc.

- Business integration
  - Use Web services to provide "for fee" services to its customers
  - Example: credit checks, reservation systems, etc.

- Business process externalization
  - Use Web services to dynamically integrate business processes
  - Example: associate different companies to combine manufacturing, assembly, wholesale distribution, and retail sales of a particular product

Web Service Roles and Interactions



Key technologies for web services

- SOAP (Simple Object Access Protocol) for communication
- WSDL (Web Services Description Language) for description of services
- UDDI (Universal Description Discovery and Integration) for discovery
- XML as a general data format

SOAP

- SOAP is an XML-based standard for messaging over HTTP and other Internet protocol
- SOAP enables interoperability between systems written in different technologies
- SOAP messages consist of three parts:
  - An envelope that defines a framework for describing what is in a message and how to process it.
  - A set of encoding rules for expressing instances of application-defined data types.
  - A convention for representing remote procedure calls and responses.

WSDL

- ◆ WSDL is an XML document for describing :
  - What a Web service can do - *interface specification*
  - How to invoke it - *binding specification*
  - Where it resides - *service specification*

- ◆ "*What a Web service can do*" is described abstractly through *portTypes*, *operations* and *messages*:
  - *PortTypes* correspond to Java interface or class
  - *Operations* correspond to methods
  - *Messages* correspond to a single piece of information moving between the invoker and the service. A method call involves two messages: request and response.

- ◆ "*How to invoke it*" is described by bindings
  - WSDL is protocol neutral
  - Bindings bind the abstract portType descriptions to a particular protocol, such as SOAP.

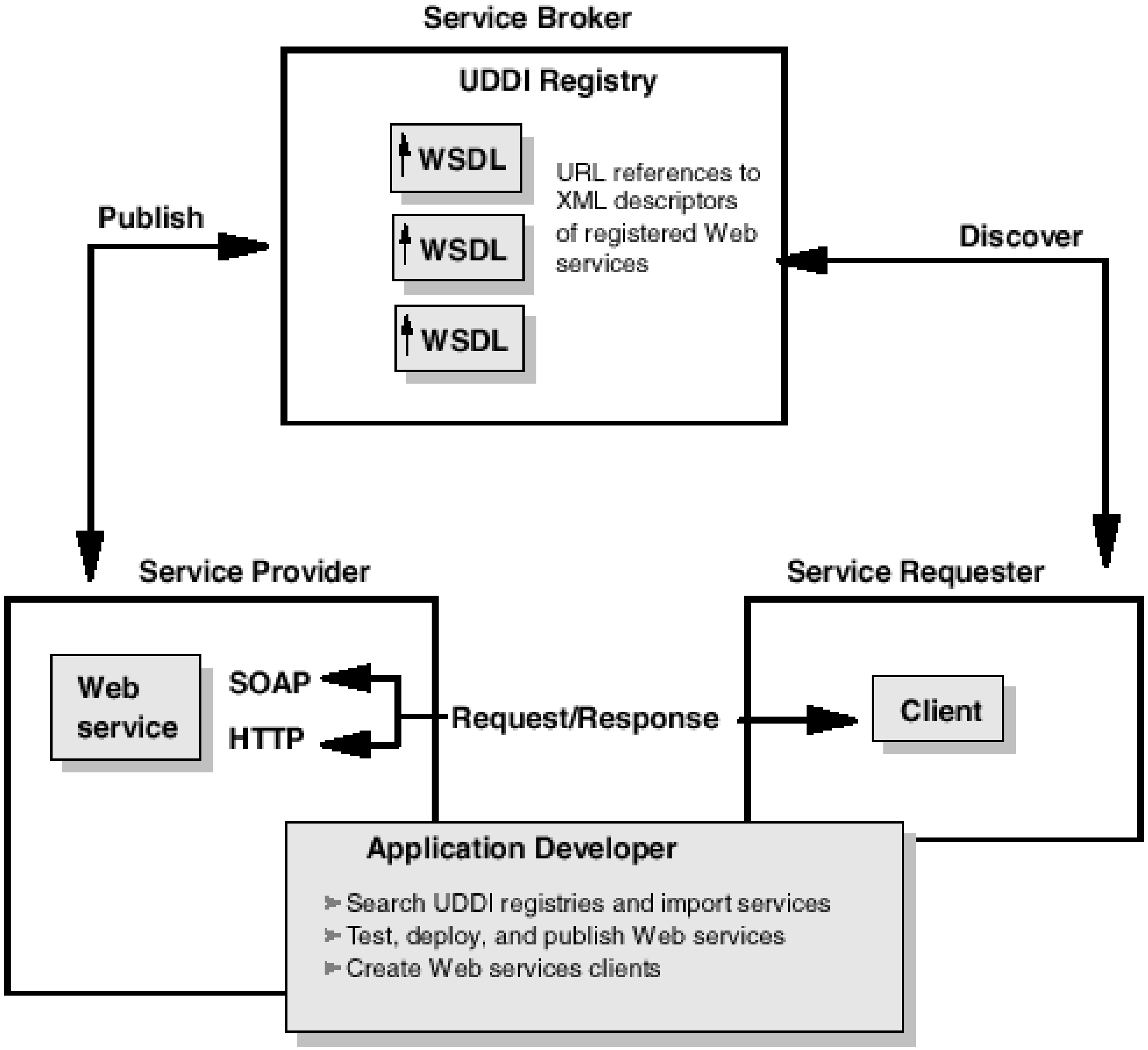
- ◆ "*Where it resides*" is described by services (a collection of ports)
  - Essentially a URL that points to the service provider

UDDI

- UDDI registry stores
  - Business information - information about the business that has published Web services
  - Service information - information about Web services
  - Binding information - information for determining the entry point and construction specifications for invoking Web services
  - Metadata - information describing the specifications for services

- UDDI has two functions:
  - It is a SOAP-based protocol that defines how UDDI clients communicate with registries
  - It is a particular set of globally replicated registries

Web Service Roles and Technologies


Service Broker
UDDI Registry
WSDL
WSDL
WSDL
URL references to XML descriptors of registered Web services
Publish
Discover
Service Provider
Service Requester
Web service
SOAP
HTTP
Request/Response
Client
Application Developer
Search UDDI registries and import services
Test, deploy, and publish Web services
Create Web services clients