CHAPITRE 6 Composants et Architecture 3 1/3

6.1 Introduction

6.1.1 Des objets aux composants

Objets:

“programmation in the small“

granularité fine

réutilisation, composition, mais à un niveau de détail important (e.g. structure de donnée, frame d'une fenêtre, etc.)

Pas de compositions autre que par programmation:

Héritage, clientèle (délégation)

Composants:

“programmation in the large“

granularité plus grosse,

réutilisation mais à un niveau différent (objet graphique spécifique, module fonctionnel, etc.)

Mécanismes de compositions (assemblage) spécifiques (e.g. “intéractif“, ADL, pas forcément par programmation)

Interface, Attributs, Implémentation

6.1.2 Définitions

Composant:

Module logiciel autonome pouvant fonctionner sur différentes plates-formes

Exporte des attributs, propriétés, ou méthodes

Peut être configuré

Capable de s'auto-décrire

Briques de base configurables pour la construction d'une application par composition

A Component =
a unit of Composition and Deployment

Component =
a module (80s!) but subject to:

- Configuration
  (variation on Non Functional Properties)
- Instantiation

To be deployed on various platforms:

some portability

6.1.3 Exemples

- Java Beans
- Enterprise Java Beans
- COM / DCOM, --> .NET, C#
- Composants CORBA (CCM)

6.1.4 Characteristics -- How ?

How it works --- Common characteristics

A standardized **way to describe a component:**

a specification of what a component:

Provides
(Interfaces, Properties to be configured)

Requires (services, etc.)

Accepts as parameterization

Usually dynamic discovery and use of components:

auto- description

(Explicit information: text or XML, reflection, etc.)

Usually components come to life through:

several classes, objects

Legacy code:

OO code wrapper to build components from C, Fortran, etc.

6.1.5 My Definition of Components

A component in a given infrastructure is:

a software module,

with a standardized description of what it needs and provides,

to be manipulated by tools for Composition and Deployment

6.1.6 Component Orientedness

Level 1: Instantiate - Deploy - Configure

- Simple Pattern
- Meta-information (file, XML, etc.)
  JavaBeans, EJB

Level 2: Assembly (flat)

- Server and client interfaces CCM

Level 3: Hierarchic

- Composite Fractal, ProActive, ...

Level 4: Distributed + Reconfiguration

- Binding, Inclusion, Location
  ProActive + On going work

Interactions / Communications:

Functional Calls:

service, event, stream

Non-Functional:
instantiate, deploy,
start/stop, inner/outer, re-bind

6.1.7 Architecture

Notion de Conteneur:

encapsulation d'un composant
prise en charge (automatique, masquage) de services systèmes:

- nommage
- découverte
- sécurité
- transaction
- persistance

Notion de Structure d'accueil:

Espace d'exécution des conteneurs et des composants (espace d'adressage, processus)
Fonctionnalité supplémentaires (téléchargement de code, etc.)

Notion de “Bus logiciel“, middleware:

Communications synchrone ou asynchrone

entre:

- client et composants
- composants

Messages (e.g. JMS, Java Messaging Services)

Signalisation et événements

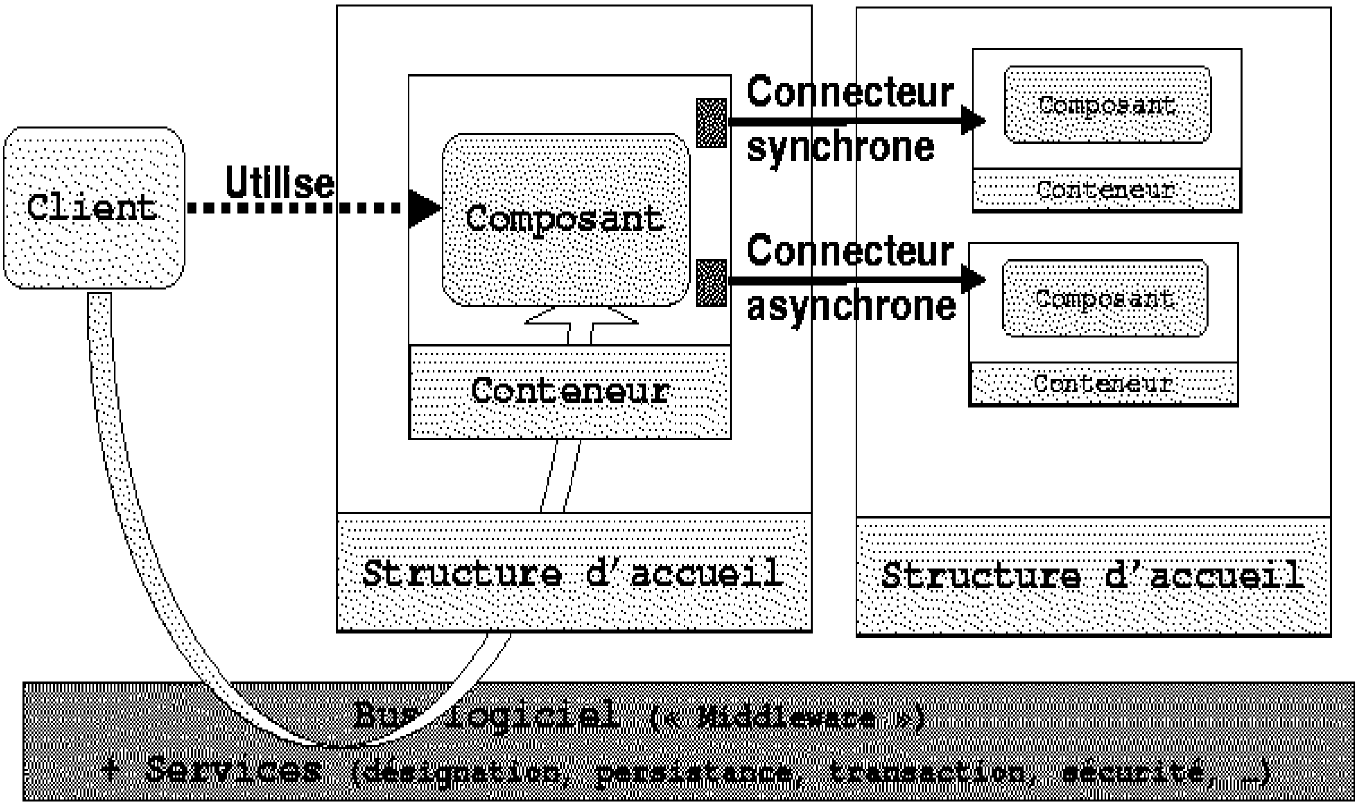

FIGURE 24 Architecture d'un système à composants

le composant est éventuellement un programme écrit dans un autre langage que la structure d'accueil:

dans ce cas on parle de "legacy Code":

"Code Patrimoine"

6.1.8 Composants et Introspection

L'introspection est une caractéristique importante des composants:

- format standardisé de description des composants
- capacité des composants à s'auto-décrire
- capacité de découvrir dynamiquement les caractéristiques d'un composant

6.1.9 Classification des solutions industriels

Partie client / Graphique / IHM:

Sur les postes clients

OLE, COM, ActiveX (Microsoft)

Java Beans (Sun)

Partie Serveur / métier:

Sur les postes serveurs, et
liaison avec les BD, transactions, sécurité

COM+, MTS (Microsoft), .Net

Enterprise Java Bean (Sun):

CORBA CCM

Exemples d'EJB serveurs:

BEA Systems, Weblogic

Objectweb JOnAS, JBOSS

Lutris, Enhydra

ObjectSpace, Voyager App. Server

Oracle, Oracle 8i iAS

Sun, iPlanet Application Server (Netscape AOL, Time Warner CNN ?)

Sybase, Enterprise Application Server,

IBM, WebSphere Application Server, ...

Attention: il y a différentes caractéristiques

- EJB version 1.1 ou 2.0 ou 3.x
- Servlets
    - avec serveur HTTP intégré, ou
    - a besoin d'un serveur externe
- JSP
- JMS: Java Message Service
- WAP/WML(support pour Wireless Application Protocol + Wireless Markup Language)
- CORBA ORB capabilities
- EJB ODBMS (inclut une BD)

- SOAP (Simple Object Access Protocol),
  Protocole de communication léger basé sur un format texte XML
  Au dessus: RPC en XML

6.2 Java Beans

Composant généralement graphique

Pouvant être composés dynamiquement:

- assemblage dynamique
- avec visualisation interactive

Construction interactive d'interface (cf. Next Step, InterfaceBuilder)

http://java.sun.com/docs/books/tutorial/javabeans/index.html

6.2.1 Principes et concepts

Un composant JavaBeans (“Bean“) est une classe Java (éventuellement plusieurs)

Chaque Bean a un certain nombre de

Caractéristiques (features):

propriétés (properties)

méthodes publiques

Événements

Par défaut, toutes les méthodes publiques sont des caractéristiques du bean

Ces caractéristiques sont “visibles“ car leur nommage suit une certaine “convention de nommage“ (un “design pattern“, affaibli)

Des outils interactifs de manipulation de beans peuvent:

- découvrir automatiquement (dynamiquement) les caractéristiques d'un bean à partir du byte code
- rendre ces caractéristiques manipulable interactivement (accessible par des menus, etc.)
- insérer un bean dans une fenêtre d'interface
- modifier ces caractéristiques:
  apparence, etc.
- construire des interactions avec les autres beans
- composer plusieurs beans pour créer
    - une applet,
    - une application
    - un nouveau bean

... sans écrire une ligne de code

Utilisation de la réflexion java (introspection)

Utilisation des conventions de nommage, ou d'API spécifiques pour fournir les informations

Événements:

- “Listener“ (abonné) et “source“ beans
- Un bean donné peut déclencher (fire) des événements (source)
- Il peut aussi en recevoir (handle) un certain nombre
- Les outils peuvent découvrir pour un bean donné les événements qu'il peut déclencher et gérer

Les beans sont persistants (attributs, mais également toutes les propriétés d'un bean).

La sérialisation Java est utilisée

6.2.2 Exemple

Le bean le plus “simple“

```
import java.awt.*;
import java.io.Serializable;
public class SimpleBean extends Canvas
                                  implements Serializable
{
  //Constructor sets inherited properties
              public SimpleBean(){
                 setSize(60,40);
                 setBackground(Color.red);
              }
}
```

Étend java.awt.Canvas

Implémente l'interface java.io.Serializable, une nécessité pour tous les beans.

Au moins un constructeur par défaut (sans argument).

Le constructeur positionne la couleur de fond et la taille.

Deux spécificités pour le manipuler dans la bean-box:

- Créer un “manifest file“ (fichier texte qui décrit le contenu d’un jar, e.g. file manifest.tmp):
  Name: SimpleBean.class
  Java-Bean: True
- Mettre dans un jar le .class et le manifest:
  jar cfm SimpleBean.jar manifest.tmp SimpleBean.class

6.2.3 Propriétés

Caractéristiques (features):

propriétés (properties)

méthodes publiques

Événements

Les propriétés sont des attributs (privés en général, éventuellement publics) pouvant être positionnés de l'extérieur.

Elles ont:

- un type
- un mode: read/write, read only, write only (dépend de l’existence des routine setX, getX)

Elles portent par exemple sur:

- la taille,
- couleur,
- position, ...
- le comportement

Des règles syntaxiques (~design pattern) permettent d'identifier les propriétés:

- une routine public T getX ()
- une routine public void setX (T t)
- un attribut private T X = <valeur par défaut>;

L'introspection Java, et ce design pattern, permettent aux éditeurs de découvrir automatiquement les propriétés.

Exemple:

On ajoute une propriété “color“ au SimpleBean:

1. Instance variable

```
private Color color = Color.green;
```

2. Méthode “getter“

```
public Color getColor(){
  return color;
}
```

3.Méthode “setter“

```
public void setColor(Color newColor){
  color = newColor;
  repaint();
}
```

4. Surcharger la méthode paint() héritée de Canvas afin de prendre en compte la propriété

```
public void paint(Graphics g) {
  g.setColor(color);
  g.fillRect(20, 5, 20, 30);
}
```

5.Compiler le Bean, le charger dans la beanBox

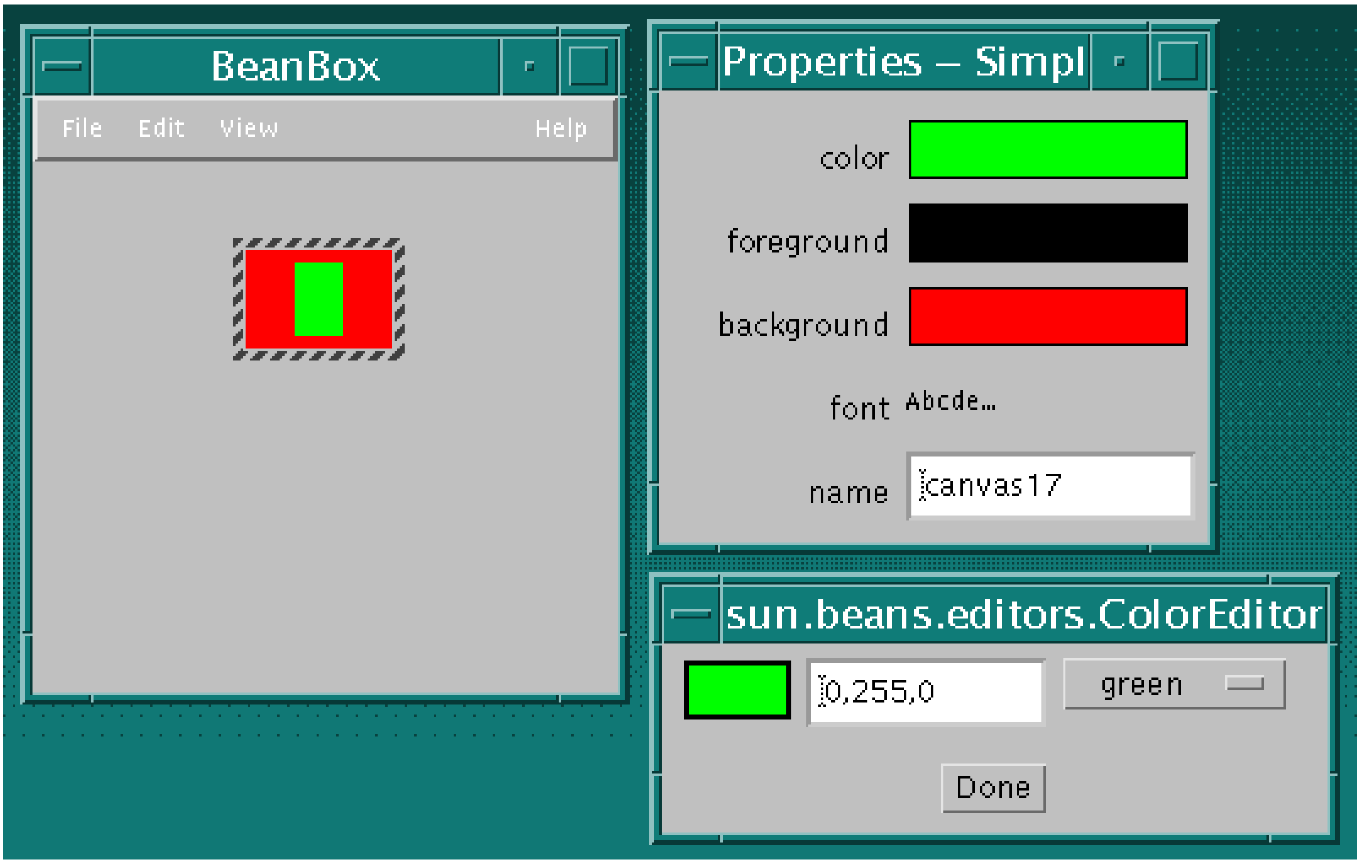

FIGURE 25 Modificatin de propriétés

L'outils découvre la propriété, et on peut la changer interactivement.

Il existe différents types de propriétés:

- simples
- indexées: ensemble de valeur (array)
- liées (bound): un événement est notifié à chaque modification de l'attribut
- contraintes (constrained): notification + vérification (“vetoed“ par le bean en cours ou un autre)

6.2.4 Evénements

Utilise les événements standard Java: JDK, Swing + Pattern

- event source
- event listener(s)
- un événement est un objet (id, source, etc.)

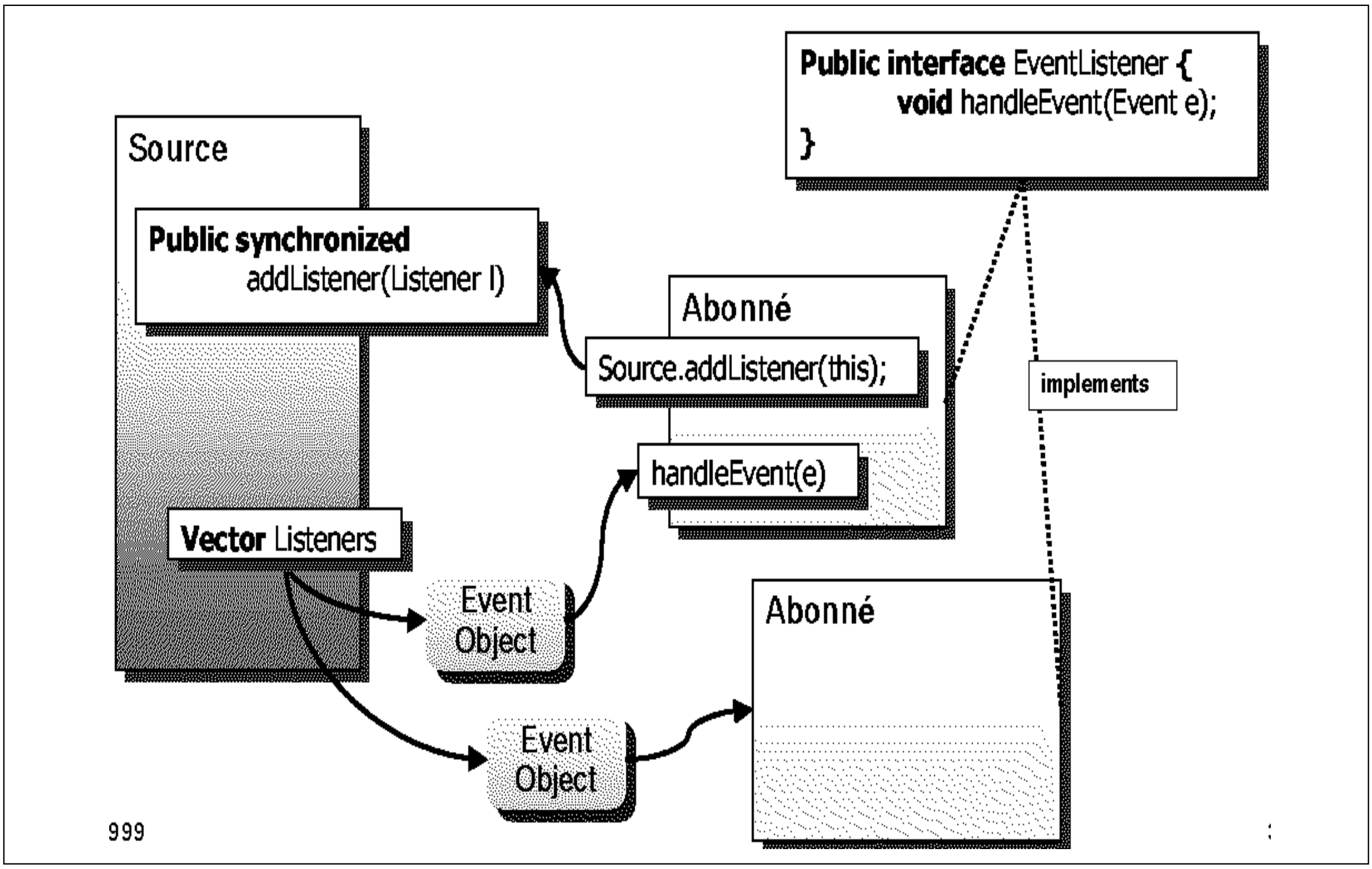

FIGURE 26 Principes des événements Java, JavaBeans

Par reflection, une source envoie un événement e1 aux listeners qui ont une routine handleEvent(E1)

Trois choses à faire:

1. Une classe qui gère des événements:
   - implémente une interface “listener“, ou
   - étendre une classe qui implémente une interface “listener“

```
public class MyClass implements ActionListener {
```

2. Enregistrement d'une instance de classe qui gère des événements:

```
someComponent.addActionListener(instancOfMyClass);
```

3. Écrire le code (la méthode) qui réagit à l'événement:

```
public void actionPerformed(ActionEvent e) {
  ...//code that reacts to the action...
}
```

Utilisation de l'introspection pour découvrir les événements qu'un bean peut déclencher:

Le pattern est le suivant:

```
public void add<EventListenerType>(<EventListenerType> a)
public void remove<EventListenerType>(<EventListenerType> a)
```

Une autre solution: BeanInfo (voir plus loin)

La BeanBox (ou un autre outils) va donc pouvoir découvrir dynamiquement les événements déclenchés par un bean, et les afficher en interactif, connecter dynamiquement source et listeners.

6.2.5 Persistance

Un bean est obligatoirement persistant:

- propriétés (properties)
- information d'état (attributs qui ne sont pas des propriétés)

peuvent être sauvegardées et restaurées sur disque.

Utilisation de la sérialisation Java

Deux façons de rendre un bean persistant:

1. Implémenter Serializable:

   - interface java.io.Serializable
     automatique

Tout est sérialisé, sauf transient et static
Une telle classe doit avoir un constructeur sans argument, et avoir tous ses attributs sérialisables

2. Autre solution: Externalizable

   - interface java.io.Externalizable
     dans ce cas il faut explicitement implémenter:readExternal(ObjectInput) et
     writeExternal(ObjectOutput)

6.2.6 La BeanBox

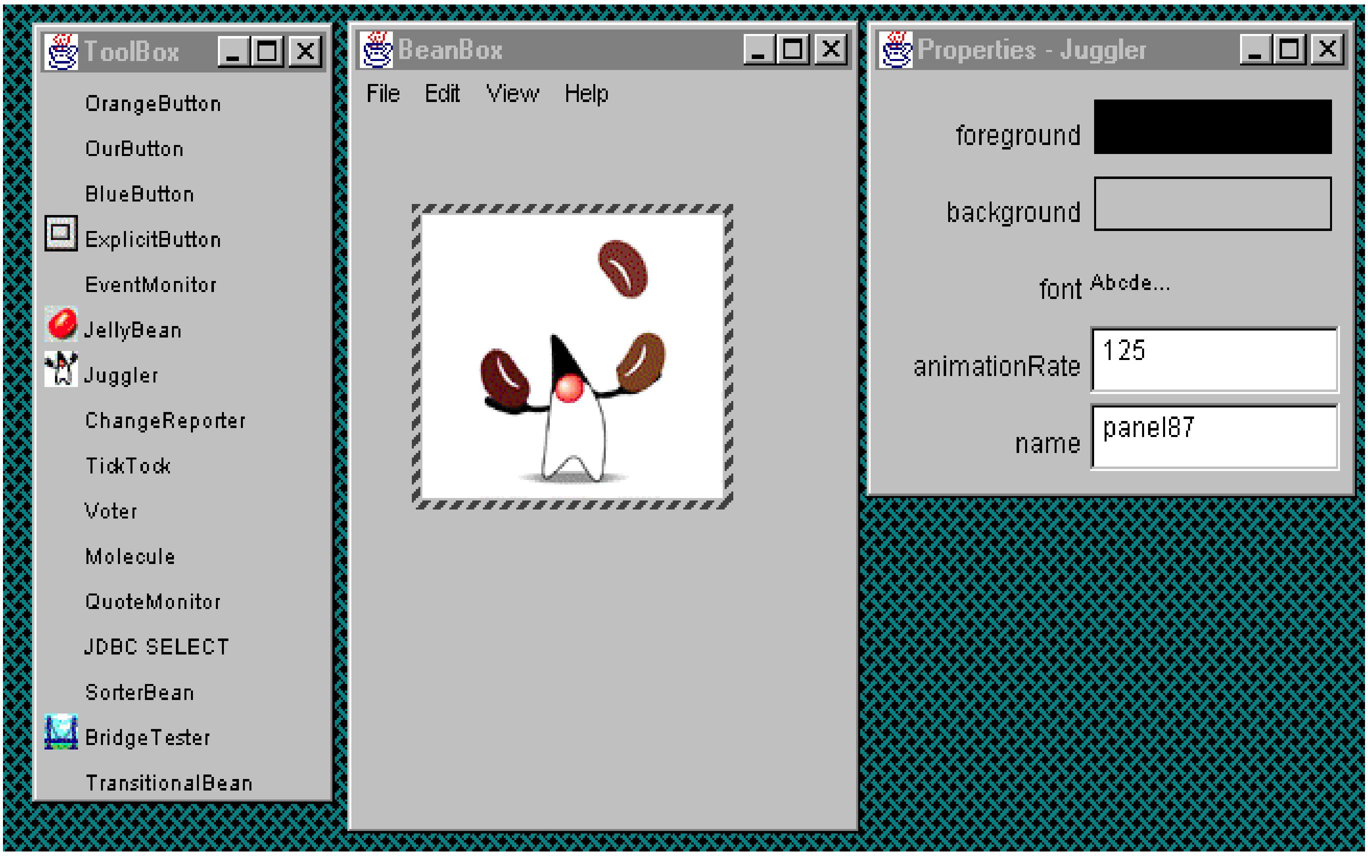

FIGURE 27 La BeanBox des JavaBeans

Permet:

- de tester les beans
- visualiser les propriétés et événements
- Connecter les beans entres eux
- générer une applet à partir du contenu de la BeanBox

6.2.7 Caractéristiques explicites: BeanInfo

BeanInfo permet de décrire explicitement les caractéristiques qu'un Bean exporte (propriétés, événements, méthodes)

Une classe Bean

ExplicitButton

alors on peut écrire une classe BeanInfo dont le nom est forcément (par convention):

ExplicitButtonBeanInfo

Dans ce cas on peut :

- se passer de la réflexion pour découvrir les caractéristiques,
- les restreindre,
- associer une icône au Bean,
- etc.

Seconde Méthode:

Description Explicite

6.2.8 Évaluation des JavaBeans

Basé sur des conventions (patterns un peu affaibli)

Utilisation intensive de l'introspection

Visualisation de l'application (fenêtre, etc.), mais pas de son architecture (class, objets qui la composent, etc.)

On peut éditer les propriétés, mais plus difficilement voir et gérer les liens entre les beans

Ne gère pas, et n'est pas orienté vers, les applications réparties

6.2.9 Exemple de bean: Juggler

a) Juggler.java

```
package sunw.demo.juggler;

/**
 * A simple JavaBean demonstration class that displays an animation
 * of Duke juggling a couple of coffee beans.   The Juggler class
 * is a good simple example of how to write readObject/writeObject
 * serialization methods that restore transient state.   In this case
 * the transient state is an array of images and a Thread.
 */

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.net.URL;
import java.beans.*;
import java.beans.beancontext.*;
import java.beans.DesignMode.*;
import sunw.demo.methodtracer.*;

public class Juggler extends Applet implements Runnable,
BeanContextProxy,
                              BeanContextServicesListener,
                              PropertyChangeListener, DesignMode {
   private transient Image[] images;
   private transient Thread animationThread;
   private int rate = 125;
   private transient int loop;
   private boolean stopped = true;
   private boolean debug = false;
   private boolean dmode = false;
   private transient MethodTracer mtService;
   private transient MethodTracer mt;

   private BeanContextChildSupport bccs = new BeanContextChildSupport() {

  protected void initializeBeanContextResources() {

     try {
```

```
            // Get method tracing service if it›s available.
            BeanContextServices bcs =
(BeanContextServices)bccs.getBeanContext();
            if (bcs.hasService(MethodTracer.class)) {
                mtService = (MethodTracer)bcs.getService(
                                    getBeanContextProxy(), Juggler.this,
                                    MethodTracer.class, null, Juggler.this);
          } else {
                bcs.addBeanContextServicesListener(Juggler.this);
            }

            // Allow nesting BeanContext control design/runtime mode.
            bcs.addPropertyChangeListener(«designMode», Juggler.this);

       } catch (ClassCastException ex) {
            // Nesting BeanContext is not a BeanContextServices
            // so do nothing.
       } catch (Exception e) {
            System.err.println(«Error initializing BeanContext resources.»);
            System.err.println(e);
       }
   }

   protected void releaseBeanContextResources() {
       if (mtService != null) { mtService = mt = null; }
           try {
            BeanContextServices bcs =
(BeanContextServices)getBeanContext();
            bccs.removePropertyChangeListener(«designMode», Juggler.this);
            bcs.removeBeanContextServicesListener(Juggler.this);
       } catch (Exception ex) {
       }
   }
    };

    public BeanContextChild getBeanContextProxy() {
   return bccs;
    }

    /**
     * Applet method: start the Juggler applet.
     */
```

```
 public synchronized void start() {
startJuggling();
 }

 /**
  * Applet method: stop the Juggler applet.
  */

 public synchronized void stop() {
stopJuggling();
 }

 /**
  * Initialize the Juggler applet.
  */

 private void initialize() {

    // Load the image resources:
images = new Image[5];
    for (int i = 0; i < 5; i++) {
        String imageName = «Juggler» + i + «.gif»;
        images[i] = loadImage(imageName);
        if (images[i] == null) {
          System.err.println(«Couldn›t load image « + imageName);
          return;
        }
    }
 }


 /**
  * This is an internal utility method to load GIF icons.
  * It takes the name of a resource file associated with the
  * current object›s class-loader and loads a GIF image
  * from that file.
  * <p>
  * @param resourceName    A pathname relative to the DocumentBase
  *       of this applet, e.g. «wombat.gif».
  * @return    a GIF image object.    May be null if the load failed.
  */
 private java.awt.Image loadImage(String name) {
```

```
    if (mt != null) mt.traceMethod();
try {
   java.net.URL url = getClass().getResource(name);
   return createImage((java.awt.image.ImageProducer) url.getContent());
} catch (Exception ex) {
   return null;
}
 }


 /**
  * Draw the current frame.
  */
 public void paint(Graphics g) {
    if (mt != null) mt.traceMethod();
int index = (loop%4) + 1;
    // If the animation is stopped, show the startup image.
    if (stopped) {
   index = 0;
}
if (images == null || index >= images.length) {
   return;
}
Image img = images[index];
if (img != null) {
   g.drawImage(img, 0, 0, this);
}
 }


 /**
  * If false, suspend the animation thread.
  */
 public synchronized void setEnabled(boolean x) {
    if (mt != null) mt.traceMethod();
    super.setEnabled(x);
    notify();
 }


 /**
  * Resume the animation thread if we›re enabled.
  * @see #stopJuggling
```

```
 * @see #setEnabled
 */
public synchronized void startJuggling() {
    if (mt != null) mt.traceMethod();
    if (images == null) {
   initialize();
}
if (animationThread == null) {
        animationThread = new Thread(this);
        animationThread.start();
}
    stopped = false;
    notify();
}

/**
 * Suspend the animation thread if neccessary.
 * @see #startJuggling
 * @see #setEnabled
 */
public synchronized void stopJuggling() {
    if (mt != null) mt.traceMethod();
    stopped = true;
loop = 0;
// Draw the stopped frame.
    Graphics g = getGraphics();
if (g == null || images == null) {
   return;
}
Image img = images[0];
if (img != null) {
   g.drawImage(img, 0, 0, this);
}
}


/**
 * An event handling method that calls startJuggling.   This method
 * can be used to connect a Button or a MenuItem to the Juggler.
 *
 */
public void startJuggling(ActionEvent x) {
    startJuggling();
```

```
    }

    /**
     * This method can be used to connect a Button or a MenuItem
     * to the Juggler.stopJuggling method.
     */
    public void stopJuggling(ActionEvent x) {
        stopJuggling();
    }


    /**
     * Returns false if the Juggler is stopped, true otherwise.
     */
    public boolean isJuggling() {
   return stopped;
    }

    public int getAnimationRate() {
        return rate;
    }

    public void setAnimationRate(int x) {
        rate = x;
    }


    public Dimension getMinimumSize() {
        return new Dimension(144, 125);
    }

    /**
     * @deprecated provided for backward compatibility with old layout
managers.
     */
    public Dimension minimumSize() {
   return getMinimumSize();
    }

    public Dimension getPreferredSize() {
        return minimumSize();
    }
```

```
/**
 * @deprecated provided for backward compatibility with old layout
managers.
 */
public Dimension preferredSize() {
return getPreferredSize();
}

/**
 * Returns true if debugging is enabled, false if it›s not.
 */
public boolean isDebug() {
    return debug;
}

/**
 * Turns debugging on, only if a MethodTracer service is available
 * and we are in design mode.
 */
public void setDebug( boolean debug) {
    if (debug) {
        if (isDesignTime() && (mtService != null)) {
            mt = mtService;
          this.debug = true;
        } else if (mtService == null) {
            System.err.println(«MethodTracer service not available.»);
            this.debug = false;
        } else if (!isDesignTime()) {
          System.err.println(«Debugging not available during runtime.»);
            this.debug = false;
   }
    } else {
        mt = null;
   this.debug = false;
    }
}

/*
 * PropertyChangeListener method.  Currently only listen for designMode.
 */
public void propertyChange( PropertyChangeEvent evt) {
if (evt.getPropertyName().equals(«designMode»)) {
   boolean dmode = (boolean)((Boolean)evt.getNewValue()).booleanValue();
```

```
        setDesignTime(dmode);
    }
     }

     /*
      * If switching to runtime, turn off method tracing if it was enabled.
      * If switching to design time and debugging is true, then enable
      * method tracing if the service is available.
      */
     public void setDesignTime(boolean dmode) {
    this.dmode = dmode;
    if (dmode) {
        if (isDebug() && (mtService != null)) {
               mt = mtService;
        }
    } else if (!dmode && (mt != null)) {
        mt = null;
    }
     }

    /*
     * Returns true if we›re in design mode, false if in runtime mode.
     */
     public boolean isDesignTime() {
    return dmode;
     }

     /*
      * BeanContextServicesListener methods.
      */
     public void serviceRevoked( BeanContextServiceRevokedEvent bcsre) {
         System.err.println(«Method Tracing service revoked.»);
         setDebug( false);
         mtService = null;
     }

     public void serviceAvailable( BeanContextServiceAvailableEvent bcsae) {
     if (bcsae.getServiceClass() == MethodTracer.class) {
         // MethodTracer service has just become available.
        try {
               mtService =
(MethodTracer)bcsae.getSourceAsBeanContextServices().getService(
getBeanContextProxy(), this, MethodTracer.class, null, this);
```

```
        } catch ( Exception ex) {
            System.err.println(ex);
        }
    }
    }

    public void run() {
        if (mt != null) mt.traceMethod();
        try {
            while(true) {
             // First wait until the animation is not stopped.
             synchronized (this) {
                while (stopped || !isEnabled()) {
                  wait();
              }
             }
             loop++;
           // Now draw the current frame.
           Graphics g = getGraphics();
             Image img = images[(loop % 4) + 1];
             if (g != null && img != null) {
                 g.drawImage(img, 0, 0, this);
             }
             Thread.sleep(rate);
           }
        } catch (InterruptedException e) {
        }
    }

}
```

b) JugglerBeanInfo.java

```
/**
 * The only thing we define in the Juggler BeanInfo is a GIF icon.
 */

package sunw.demo.juggler;
import java.beans.*;

public class JugglerBeanInfo extends SimpleBeanInfo {

   public java.awt.Image getIcon(int iconKind) {
      if (iconKind == BeanInfo.ICON_COLOR_16x16) {
         java.awt.Image img = loadImage("JugglerIcon.gif");
         return img;
      }
      return null;
   }

   public PropertyDescriptor[] getPropertyDescriptors() {
      try {
         PropertyDescriptor debug =
            new PropertyDescriptor("debug", beanClass);
  PropertyDescriptor animationRate =
            new PropertyDescriptor("animationRate", beanClass);
  PropertyDescriptor name =
            new PropertyDescriptor("name", beanClass);
  debug.setBound(true);
  animationRate.setBound(true);
         name.setBound(true);
  PropertyDescriptor rv[] = {debug, animationRate, name};
         return rv;
      } catch( IntrospectionException e) {
         throw new Error(e.toString());
      }
   }
public int getDefaultPropertyIndex() {
      // the index for the animationRate property.
      return 1;
   }

   private final static Class beanClass = Juggler.class;
}
```

c) Images et icones

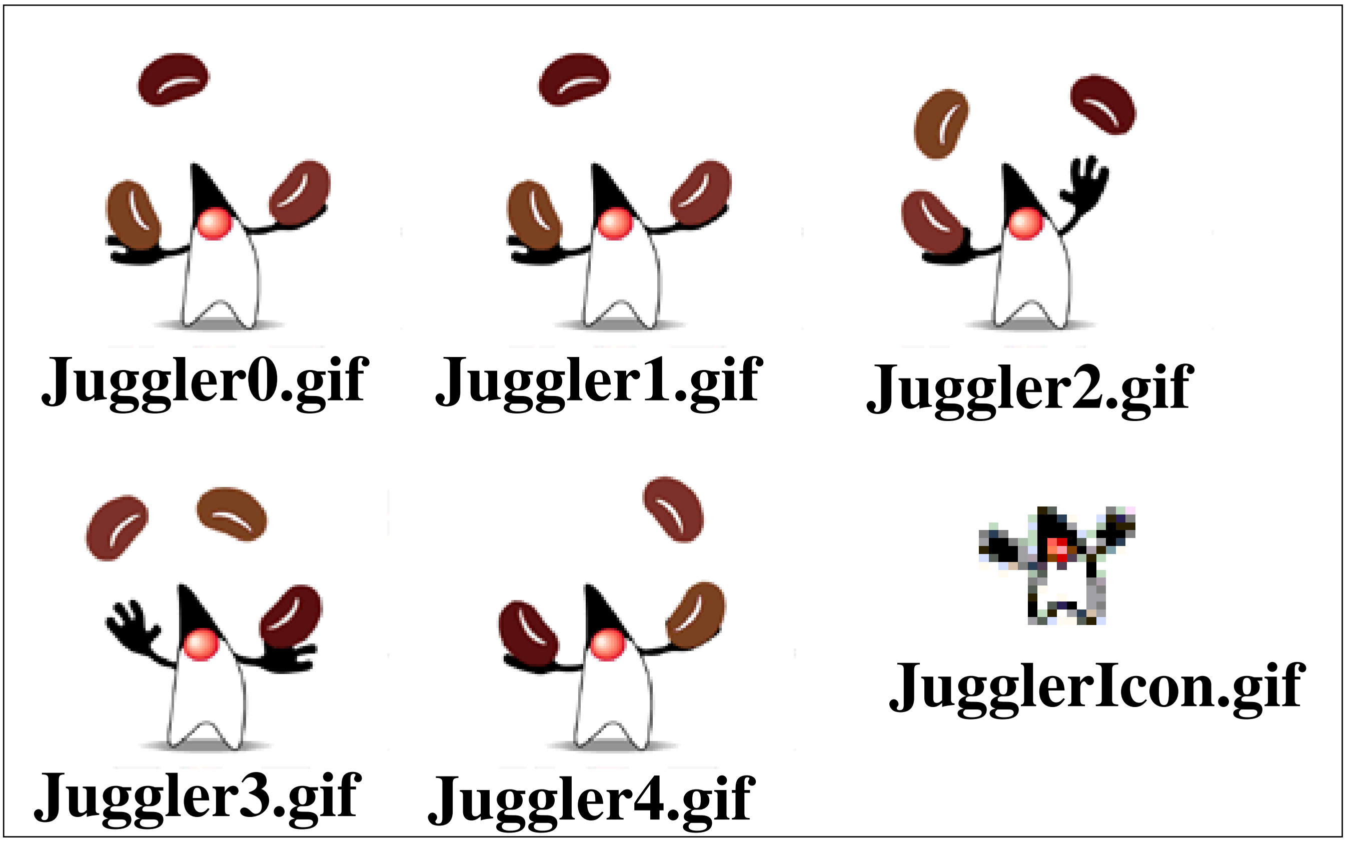

FIGURE 28 Images du Jongleur

6.2.10 Conlusion on JavaBeans

Quite simple :

a Java class (or several)

a naming convention to identify properties:

--> • method: public T getX

--> • method: public T setX

--> • an attribute: private T X= <default value>

a communication pattern:

--> • Events, Source, Listeners

and ...

a class turned into a graphical Component !

The Java introspection allows to

discover dynamically the properties,

and to configure them,
assemble JB interactively

(Also BeanInfo to explicit the properties.)

So for JavaBeans:

Software module = Java Class

(potentially several)

Standardized description = getX, setX, X, listeners

(add, remove), BeanInfo

Tools:

Composition = BeanBox, JBuilder, Java Env.

Eclipse ... ?

Deployment = JVM

(no Run Time, except for Java libraries being used)

Nothing very new:

cf. NeXTStep Interface Builder
(MacOS10, ObjectiveC, InterfaceBuilder)

but life made a bit easier with bytecode and introspection

6.3 Principes des archtectures n 1/3

6.3.1 Introduction

L'architecture 3-tiers est composée de trois éléments, ou plus précisément dans ce cadre là de trois tiers:

- Interface
- ...
- Traitement
- ...
- Persistance

C'est à dire:

- AWT, Browser, ...
- Requêtes, services, ...
- BD (transaction, stockage)

On parle aussi de couche fonctionnelle, où à chacun des tiers est attachée un rôle spécifique.

6.3.2 Architectures typiques 3 1/3

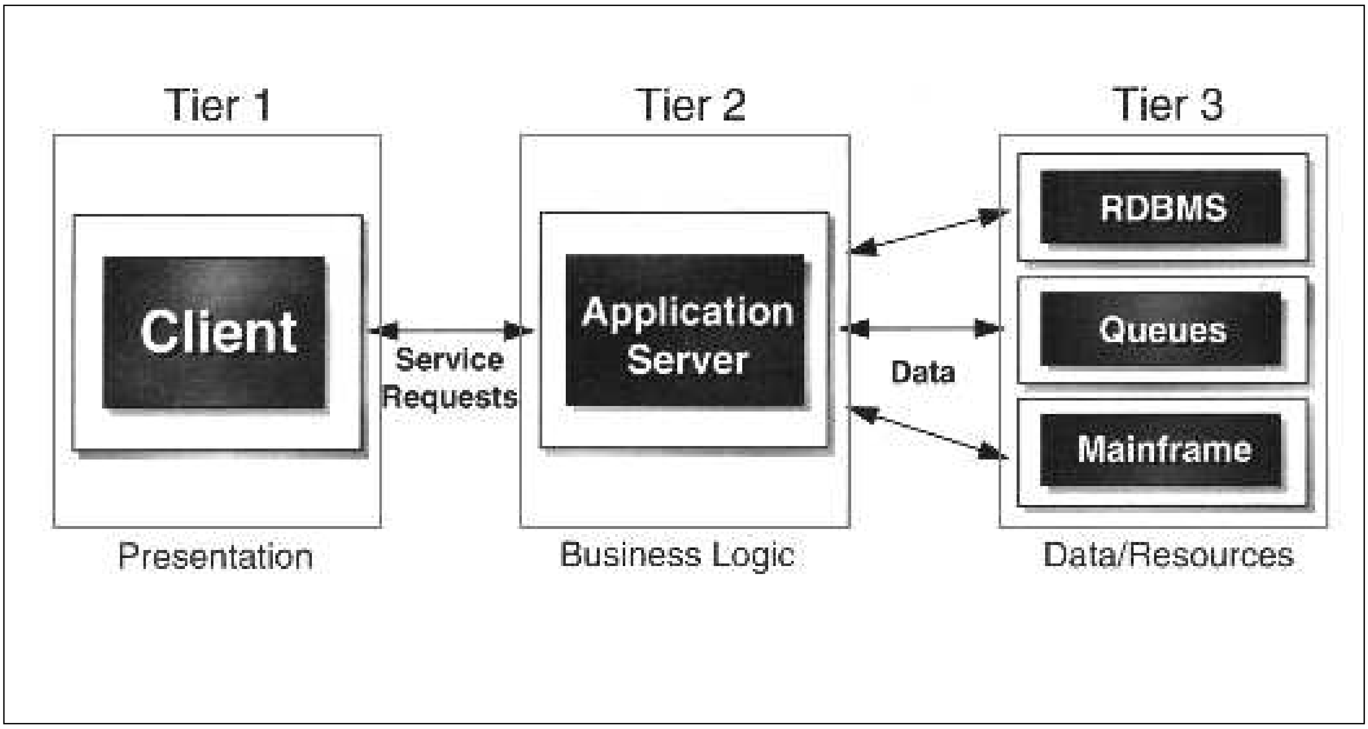

FIGURE 29 Une architecture 3-tiers

Une architecture souvent rencontrée comporte un serveur Web: architecture 4 1/3

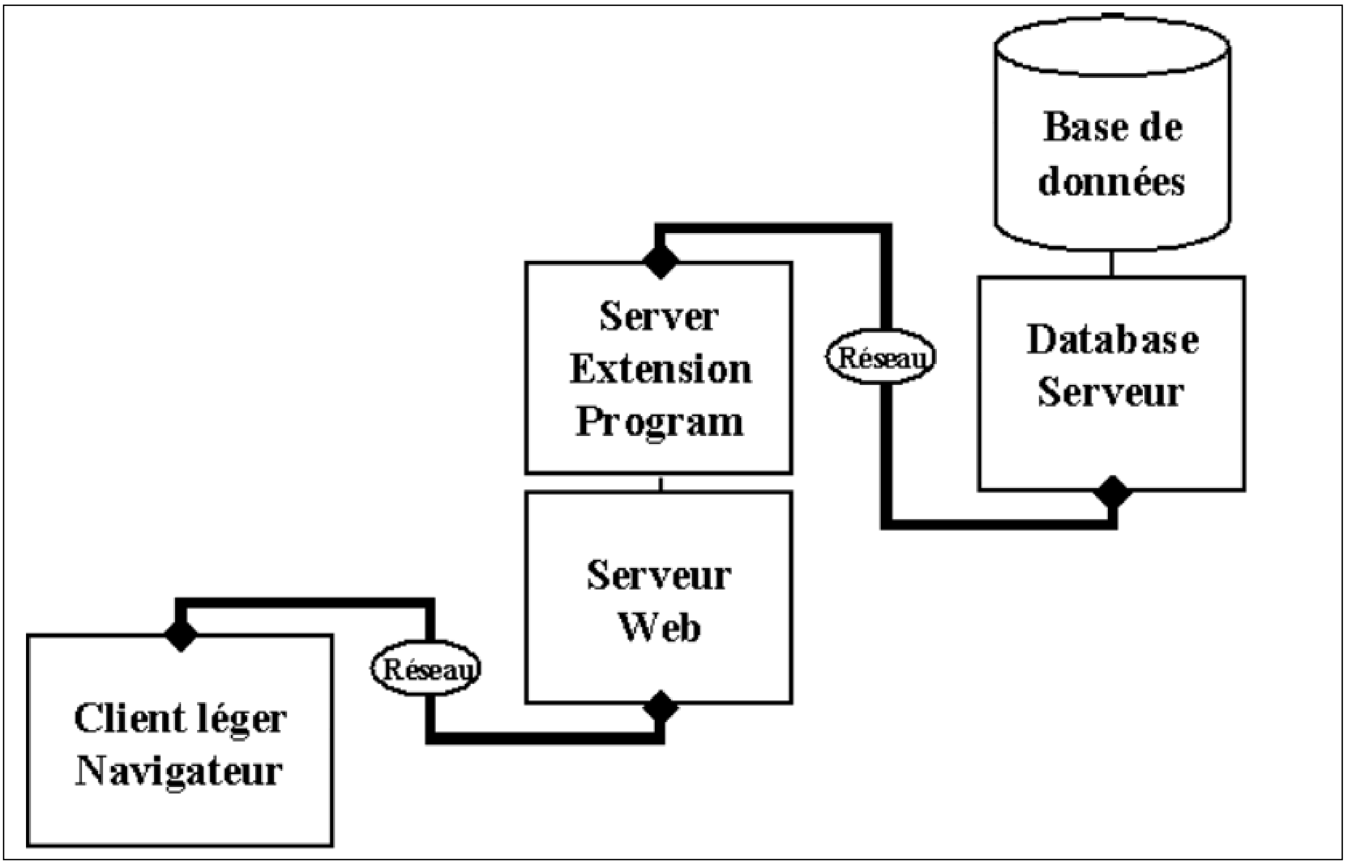

FIGURE 30 Une architecture 4-tiers avec serveur http

Mais ce n'est pas une obligation.

6.3.3 Objectifs

Rendre une application

- facile à développer
- facile à déployer (sans recompilation)
- facile à administrer, configurer
- adaptable et portable

6.3.4 Avantages

Les avantages de l'architecture 3-tiers sont principalement au nombre de quatre :

1. Les requêtes clients vers le serveur sont d'une plus grande flexibilité que dans celles de l'architecture 2-tiers basées sur le langage SQL;

2. Faisant le pendant avec la première remarque, l'utilisateur, pour l'écriture du client, n'est pas supposé connaître le langage SQL, qui ne sera pas implémenté dans la partie client qui ne s'occupe (rappelons le) que de fonctions d'affichage.

De fait des modifications peuvent être faites au niveau du SGBD sans que cela impacte la couche client. Par ailleurs et bien que nous ayons mentionné le langage SQL au niveau des bases de données, on peut très bien envisager une organisation des données sans présupposition quant au langage lui même et à leur organisation (relationnelle, hiérarchique...).

Cette flexibilité permet à une entreprise d'envisager dans le cadre d'une architecture 3-tiers une gandre souplesse pour l'introduction de toutes nouvelles technologies.

3. D'un point de vue développement, la séparation qui existe entre le client, le serveur et le SGBD permet une spécialisation des développeurs sur chaque tiers de l'architecture.

4. Plus de flexibilité dans l'allocation des ressources;

La portabilité du tiers serveur permet d'envisager une allocation et ou modification dynamique au grés des besoins évolutifs au sein d'une entreprise.

6.3.5 Inconvénients

Les inconvénients sont essentiellement au nombre de deux :

1. Une expertise de développement à acquérir qui semble plus longue que dans le cadre d'une architecture 2-tiers.

2. Corollaire du point précédent, les coûts de développements d'une architecture 3-tiers sont plus élevés que pour du 2-tiers, au début semble t'il,

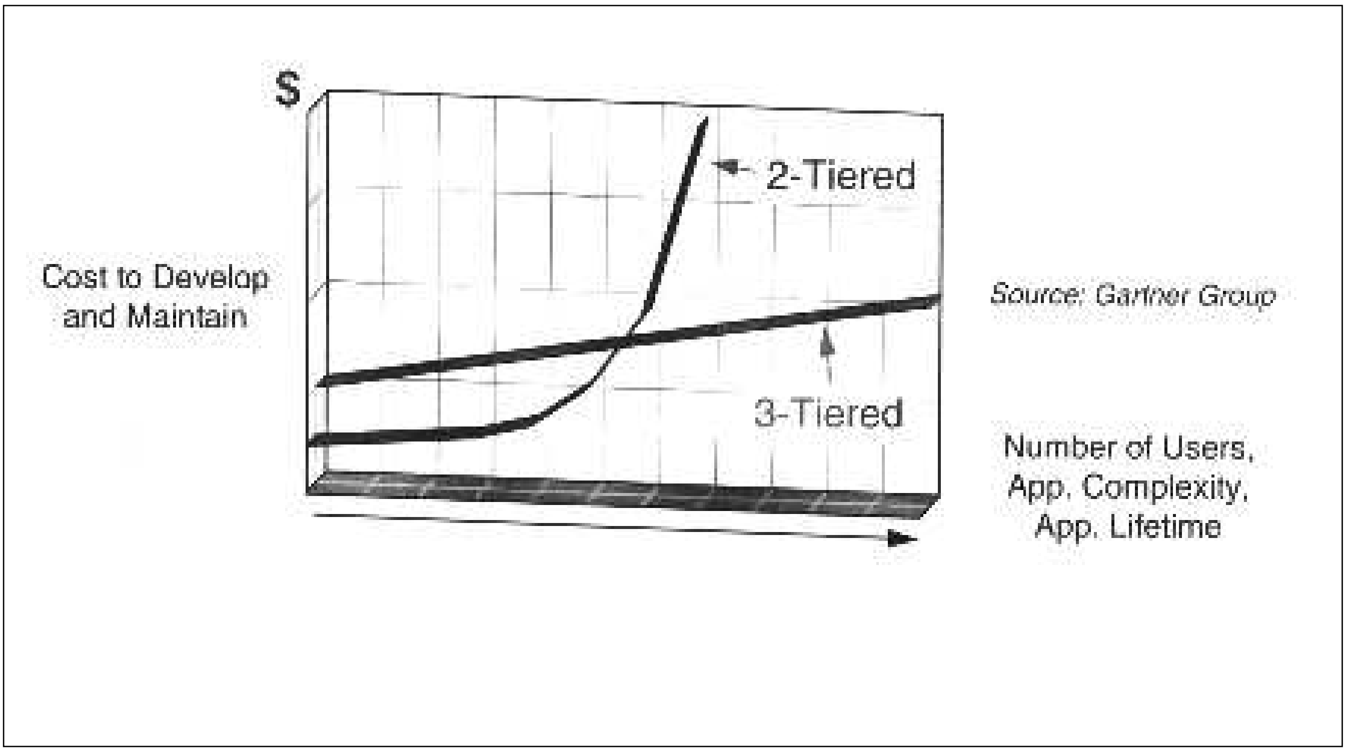

FIGURE 31 Cout 2 tiers vs. 3 tiers

D'après une étude du cabinet Gartner group (1998).

6.3.6 Couches détaillés

- 1er tiers: Présentation - Interface
  - **- Interaction avec l'utilisateur**
  - **- Présentation (PDA, Web, Client lourd)**
  - **- Vérifications**
  - **- Contrôle de flux**
  - **- Sécurité**
- 2ème tiers:
  Traitement, Logique applicative, etc.:
  - **- Contrôle de flux**
  - **- Objets Applicatifs**
    (identique qq soit client)

    exemple: gestion d'un achat sur Internet
  - **- Objets de calcul**
  - **- Objets Métiers:**

    exemple: mise à jour du compte client
- 3ème tiers: Persistance, Gestion des données:
  - **- Persistance**
  - **- Intégrité**
  - **- Accès physique (hétérogénéité)**

6.3.7 Outils et techniques standards

- 1er tiers: Interface
  - **- Téléphone, PDA,**
  - **- Browser, AppletViewer**
  - **- Browser avec Plugins**
  - **- Client AWT, SWING, ou C, C++,**
  - **...**
- 2 ième tiers:
  Gestion de la présentation, dynamicité, etc.:
  - **- Serveur HTTP,**
  - **- Servlets, JSP,**
  - **- CGI, ASP**
- 3 ième tiers:
  Traitement, Logique applicative, etc.:
  - **- JVM + Serveur EJB + RMI**
  - **- Message Oriented Middleware (MOM): JMS**
- 4ème tiers: Persistance, Gestion des données:
  - **- JDBC**
  - **- BD Objets**
  - **- Moteur transactionnels**

Entre les couches:

Communications:

- **Sockets (TCP/IP),**
- **RMI,**
- **HTTP + XML (SOAP ou XMLRPC)**
- **etc.**

Orientations du cours - TD:

Etude sommaire des Composants et EJB (en rapport avec la programmation répartie)

Dans le mini-projet:

Réalisation d'une appli N tiers sans EJB pour bien comprendre les services rendus et cachés dans les EJB (Factory, Stockage).

Ou architecture P2P (selon les années).

6.4 Enterprise Java Beans: EJB

6.4.1 Introduction

Cadre:

Applications N tiers (4 1/3):

Avec gestion de la distribution

http://java.sun.com/products/ejb/index.html
http://web2.java.sun.com/products/ejb/training.html

6.4.2 Principes et concepts

EJB spécifications:

- architecture (4 1/3) Java, où la partie serveur est construite par des Enterprise Beans
- architecture: ensemble d'interfaces
- les EJB sont des composants serveurs, applicatifs, “métiers“
- EJB: partie centrale de la plate-forme

Éléments de l'architecture EJB:

- Clients
- Serveurs
- Conteneur

- Composants logiciels:
  Enterprise Beans, EB, ou Beans

Relations entre les éléments:

- un client “référence“ un EJB
- un EJB serveur, JVM, “contient” des EJB conteneur
  (en pratique des objets spéciaux)
- un EJB conteneur contient des Beans
- les conteneurs isolent les clients des Beans

Conteneurs et serveurs implémentent les mécanismes de bas niveau:

transactions,

persistance,

gestion mémoire, ...

Entre Client et EJB Server/Conteneur : RMI

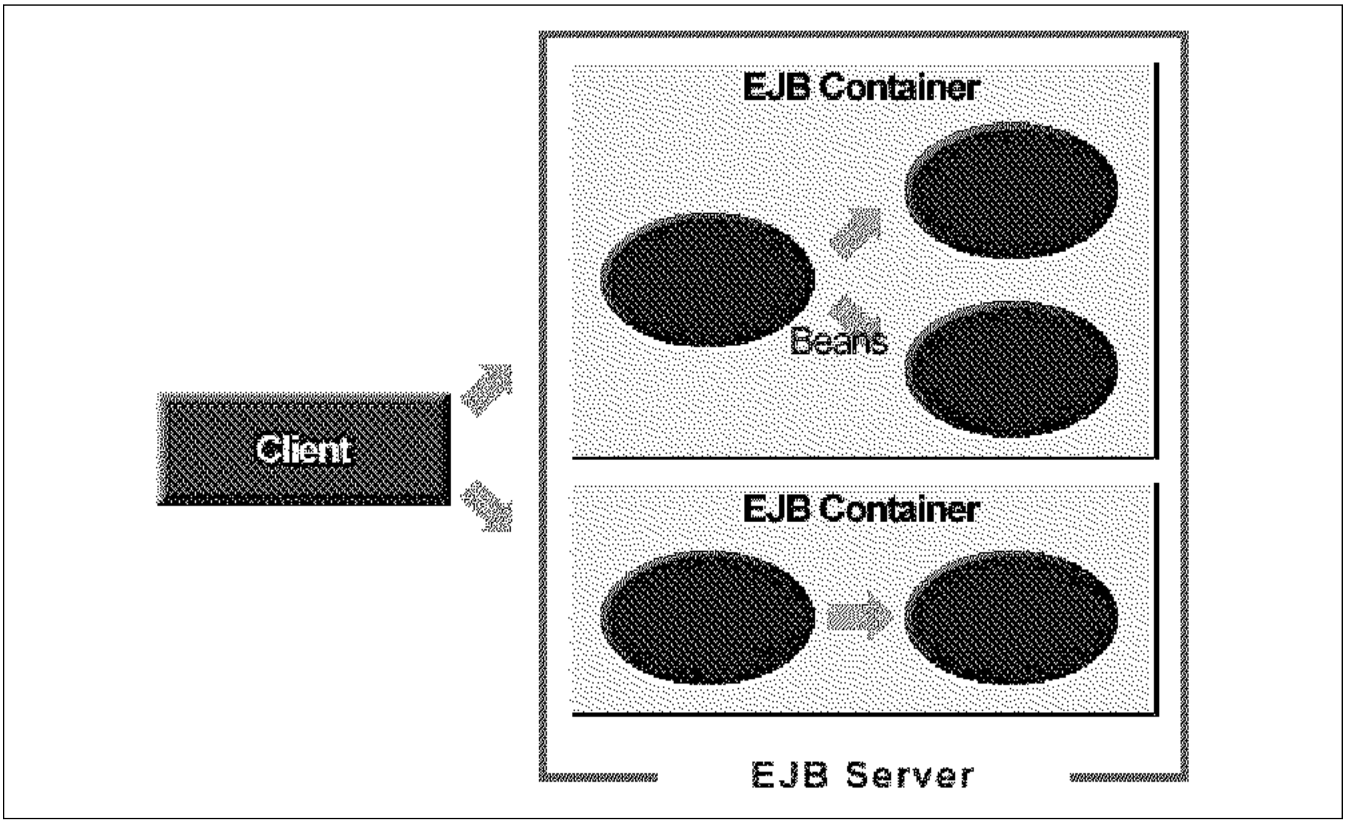

FIGURE 32 Principes du modèle d'exécution

Les EJB peuvent encapsuler des services implémentés dans un autre langage que Java.

6.4.3 Plate-forme EJB

Technologies utilisées:

- JVM, JDK

avec en particulier:

- JDBC: Java Database Connectivity
- JTA: Java Transaction API
- JMS: Java Messaging Services
- JNDI: Java Naming and Directory Services
- RMI: Remote Method Invocation
- JRMP: Java Remote Method Protocol
  (RMI protocol)
- IIOP: Internet Inter-Orb Protocol
  RMI over IIOP
  RMI / IIOP
-
- En cours de construction:
  RMI over SOAP-HTTP

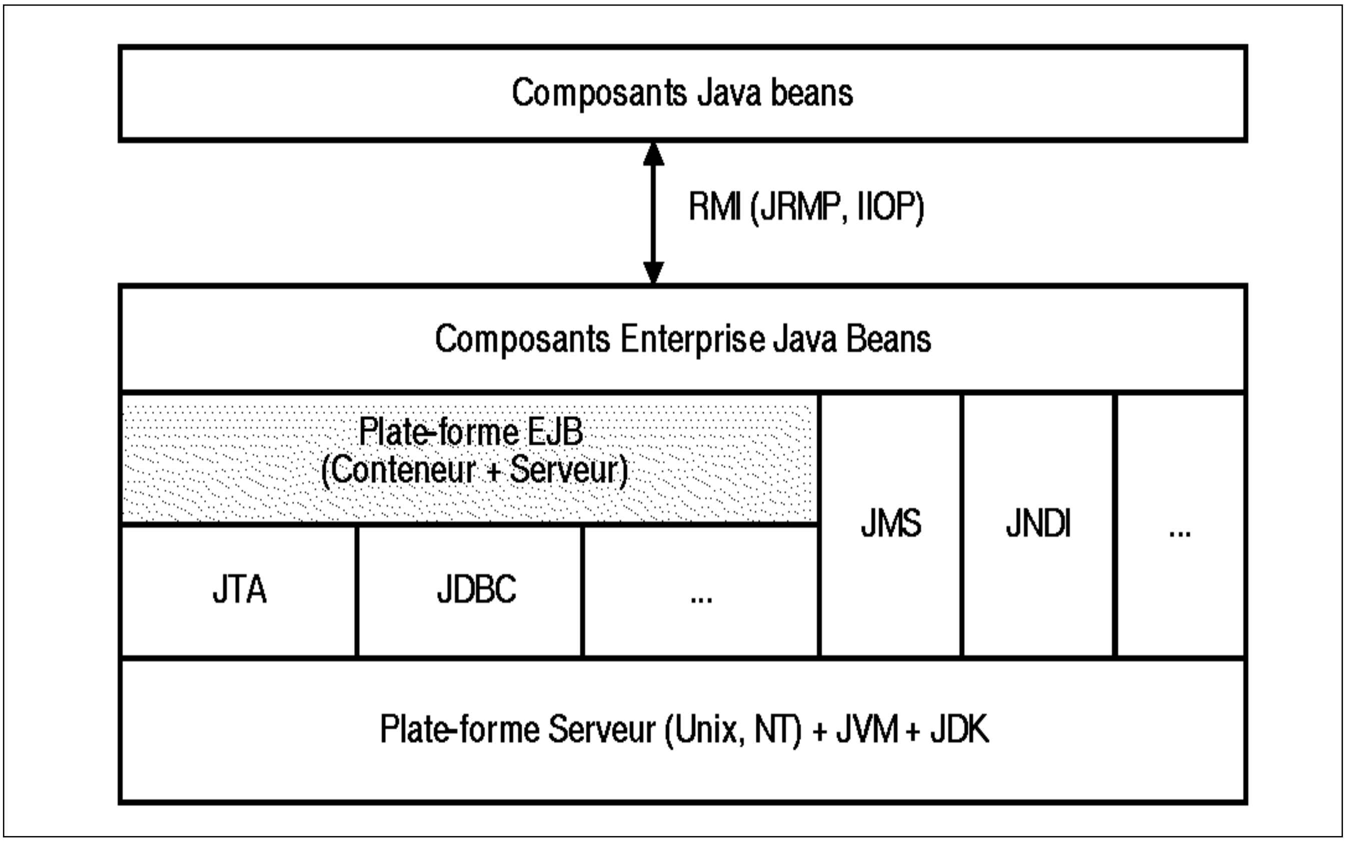

FIGURE 33 Plate-forme EJB

6.4.4 Rôles et contrats

Volonté de gérer les aspects :

Développement,

Déploiement,

Exécution

Modèle un peu économique de l'industrie des composants

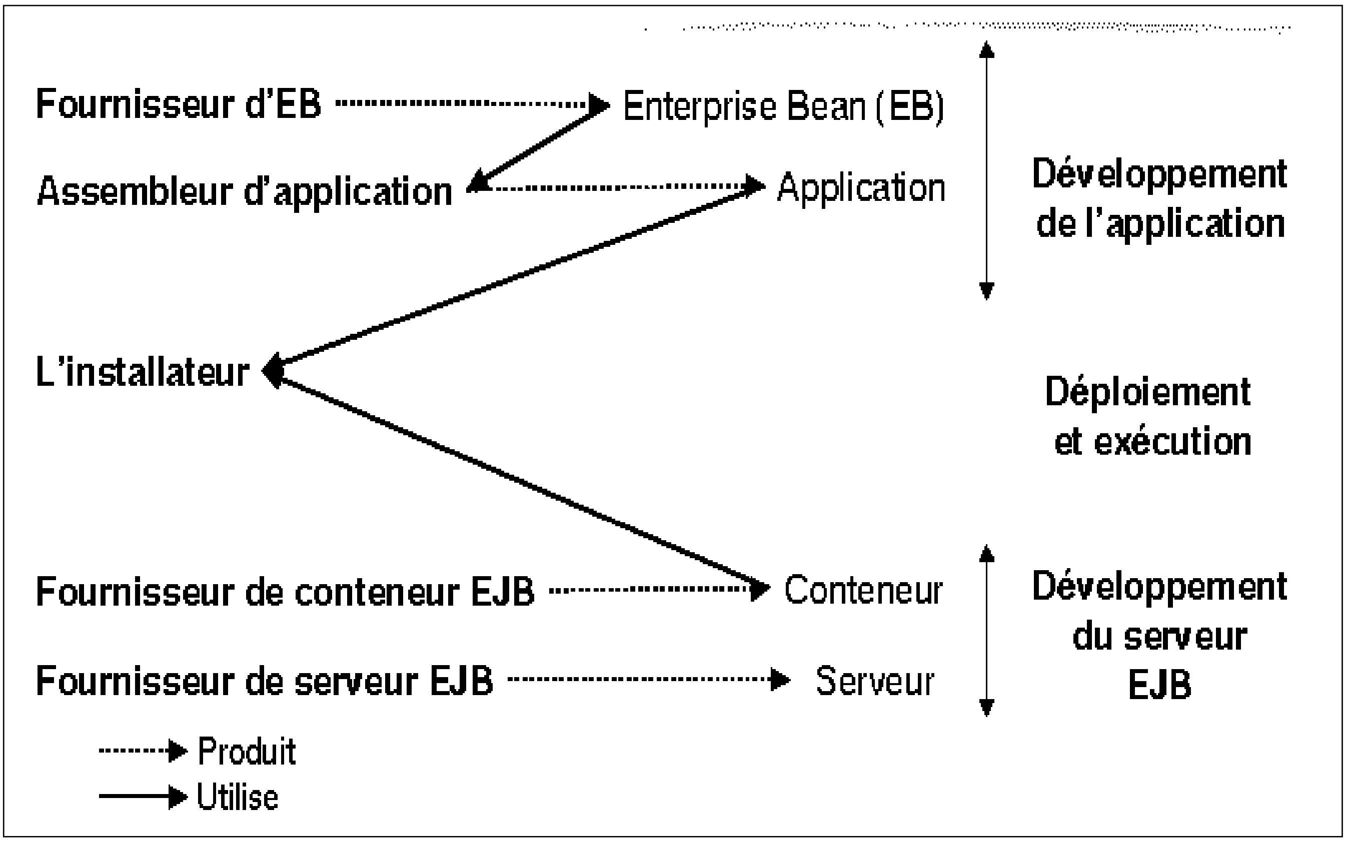

FIGURE 34 Rôles dans la production d'une application

En pratique, pour l'instant au moins,

Fournisseur de Conteneur = Fournisseur de Serveur

6.4.5 Architecture à l'exécution

Interface en deux parties distinctes:

- la partie Home qui permet de gérer le cycle de vie d'un bean:
  - créer (Factory),
  - détruire,
  - rechercher un EJB
- la partie objet EJB lui même
  - appel des méthodes de l'objet

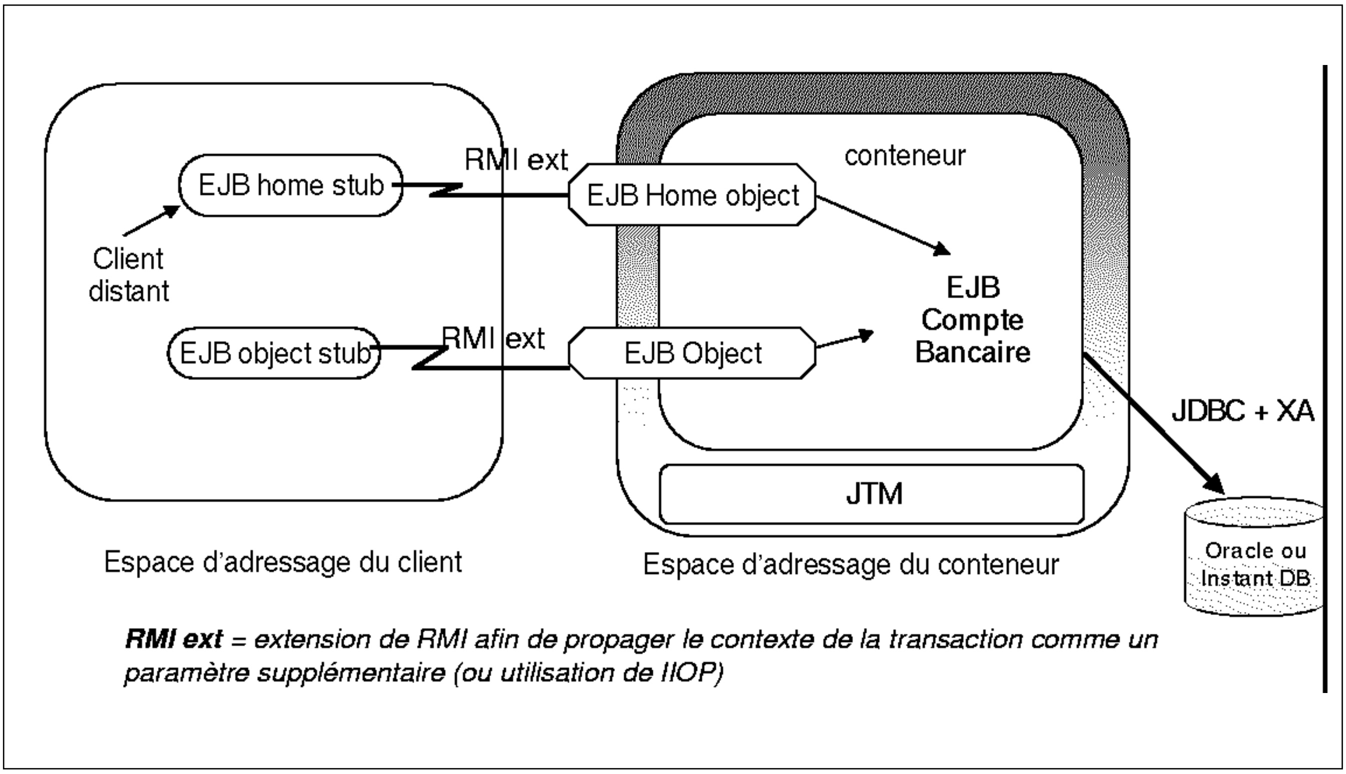

FIGURE 35 Architecture à l'exécution

Possible de le reproduire en RMI (e.g. en projet).

6.4.6 Autres aspects

a) Session et Entity Beans

Session Beans:

non persistant (short-lived), associé à un seul client

Deux catégories:

- Stateless:
  pas de données internes
  inutile de le rendre passif (on le recrée, ou en fait on le prend dans un pool)
- Statefull:
  plusieurs appel de méthodes en provenance du même client (donnent des résultats différents)

Détruits par un arrêt ou panne du serveur d'EJB

Transparents aux transactions (pas d'états)

Entity Beans

Représentent les données d'une BD

Persistants (long-lived), gérée par:

- Bean Managed persitence, ou
- Container Managed persitence

Accès multiples (x clients), et Transactions

Survivent aux pannes du serveur d'EJB

6.4.7 Autres aspects

a) Message Driven Beans

A message driven bean is a stateless, server-side, transaction-aware component that is driven by a Java message (javax.jms.message). It is invoked by the EJB Container when a message is received from a JMS Queue or Topic. It acts as a simple message listener.

A Java client, an enterprise bean, a Java ServerPagesTM (JSP) component, or a non-J2EE application may send the message. The client sending the message to the destination need not be aware of the MDBs deployed in the EJB Container. However, the message must conform to JMS specifications.

Before MDBs were introduced, JMS described a classical approach to implement asynchronous method invocation. The approach used an external Java program that acted as the listener, and on receiving a message, invoked a session bean method.

However, in this approach the message was received outside the application server and was thus not part of a transaction in the EJB Server. MDB solves this problem.

From Pramati,

on TheDerverSide.COM, http://www.theserverside.com/articles/article.tss?l=Pramati-MDB

b) Transaction distribuées

c) Sécurité

d) Concurrence

e) Répartition

6.4.8 Un exemple simple

a) EJB Remote Interface

La vue client de l'EJB. Interface Fonctionnelle

Le développeur de l'EJB doit définir cette interface

L'implémentation effectivement utilisée de cette interface sera générée par les outils fournis avec le conteneur.
Mais une implémentation des routines devra être donnée par le développeur de l'EJB dans la classe qui va implémenter le bean.

```
/** * Demo -- this is the "remote" interface of our enterprise
* JavaBean, it defines only one simple method called
* demoSelect(). As this is meant to be the simplest of
* examples demoSelect() never goes to a database, it
* just returns a string
* Note: The implementation of this interface is
* provided by the container tools but the demoSelect()
* method and any other methods in this interface
* will need to have equivalent implementations in the
* demobean.java which is supplied by the bean writer */
package ejb.demo;
import java.rmi.RemoteException;
import java.rmi.Remote;
import javax.ejb.*;
public interface Demo extends EJBObject, Remote {
	// NB this simple example does not even do a
	// lookup in the database
public String demoSelect() throws RemoteException;
}
```

Syntaxe RMI

Des restrictions existent sur les paramètres, les valeurs de retour, exceptions, etc.

Étendre l'interface Remote n'est pas forcé car si toutes les routines d'une interface génèrent l'exception RemoteException et respectent les contraintes RMI, alors une classe remote peut implémenter cette interface

b) L'interface Home

Pour un EJB session, cette interface donne le mécanisme par lequel le conteneur crée de nouveaux “session beans“ pour le compte du client.

Le développeur de l'EJB doit définir cette interface

Syntaxe RMI

Les classes effectives implémentant cette interface seront également générée par les outils fournis avec le conteneur

```
/**
* DemoHome.java - This is the Home interface it must
* extend javax.ejb.EJBHome and define one or more
* create() methods for the bean.
* Note: The implementation of this interface is
* generated by the container tools.
*/
package ejb.demo;
```

```
import javax.ejb.*;                          // Factory
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.*;
/**
* This interface is extremely simple it declares only
* one create method.
*/
public interface DemoHome extends EJBHome {
  public Demo create() throws CreateException,
                                 RemoteException;
}
```

Note: on retourne un objet de type Demo

c) La classe qui implémente l'EJB

On va coder les fonctionnalités de l'EJB, les “aspects métiers“ (“business logic“)

Le développeur de l'EJB doit définir cette classe

1. On doit implémenter une interface particulière: SessionBean (autre interface possible: EntityBean)

- Note: les routines sont vides car on ne va pas chercher qq chose dans la base de données

2. On va également implémenter les routines de l'interface de l'EJB (**Demo**)

```
package ejb.demo;

import javax.ejb.*;
import java.io.Serializable;
import java.util.*;
import java.rmi.*;

public class DemoBean implements SessionBean {
static final boolean verbose = true;

private transient SessionContext ctx;
private transient Properties props;

// Implement the methods in the SessionBean interface
public void ejbActivate() { // appelée par le conteneur
                              // appelée lorsque l'objet redevient actif
  if (verbose)
  System.out.println("ejbActivate called");
}
public void ejbRemove() {
              // appelée par le conteneur lorsqu'un client appelle
             // la méthode remove().
  if (verbose)
  System.out.println("ejbRemove called");
}
public void ejbPassivate() { // appelée par le conteneur
                             // appelée lorsque l'objet devient passif
  if (verbose)
  System.out.println("ejbPassivate called");
}
```

```
/**
* Sets the session context * @param SessionContext
*/
public void setSessionContext(SessionContext ctx) {
  if (verbose)
        System.out.println("setSessionContext called");
  this.ctx = ctx;
  props = ctx.getEnvironment();
}
```

```
/**
* This method corresponds to the create method in
* the home interface DemoHome.java.
* The parameter sets of the two methods are
* identical. When the client calls
* DemoHome.create(), the container allocates an
* instance of the EJBean and calls ejbCreate().
*/
public void ejbCreate () {
  if (verbose)
        System.out.println("ejbCreate called");
}
```

Enfin, voici la partie fonctionnelle, logique buisness!

```
/**
* **** HERE IS THE BUSINESS LOGIC *****
* Do the demoSelect() but don't even go to
* the database in this eg but instead just
* return a String.
* The really BIG thing to notice here is that
* this is the only code we have invented at all
* the rest of the code has been declarations
* or simply implementing methods which are
* part of the EJB interfaces and in this example
* are not even used.


*/
public String demoSelect()
  throws RemoteException
{
  return("hello world");
}
}
```