Automation of Component Communication in Java

Alexander Sakharov

Problem Statement

Software components are self-contained entities characterized by properties, events, and methods. When assembled in applications, components communicate to each other without having direct static references between them. Overall, component communication amounts to property change propagation and inter-component calls including event method calls. Many of inter-component (communication) calls have to be dynamically dispatched. Writing this dispatching code could be quite intricate even when employing the command pattern. Communication-related code is often multi-threaded. When hard-coded, component communication implementations are error-prone and difficult to maintain for applications having big quantities of components.

In traditional one-tier applications, components are usually assembled into hierarchical composites. Automation of the assembly of visual components in composites is well supported by many tools. Visual Basic is one example of that. There are also general-purpose component assembly tools for Java: BeanBox; ContextBox. One significant problem with these tools is that they only help automate synchronous event delivery with listeners bound at design time. Apparently, component interaction is not always so in real life. These tools do not support dynamic dispatching of calls at all. Another problem with them is that they help bind properties of only those components that announce property changes. In reality, most components are not designed to generate property change events. These tools also require matching types for events and bound properties.

In enterprise applications, one client (e. g. a servlet) communicating with multiple components (e. g. EJBs or JSPs) is the most widely used communication model. Automation of component communication for this model has not been much addressed. Note that communication via JMS does not require further automation.

Solution

There is a generic solution to the problem of automating implementation of component communication in Java at the level of one JVM. It applies to J2EE components tied with one client as well. J2EE components may be remote objects (EJBs) or non-Java objects (JSPs). J2EE components having a common client are usually represented by their proxies (stubs) on the same JVM as the client. First, we introduce class JSPProxy representing JSPs on the side of a client servlet. Second, we introduce a generator for automated construction of wrapper classes for EJBs. Ironically, EJBs are not designed for dynamic binding; these wrapper classes fill the gap. A single wrapper class represents each EJB on the client side as opposed to a combination of EJBObject and EJBHome.

This approach allows users to specify interactions for a component collection, and then, a communication adapter is automatically generated from the specifications. The communication specifications are comprised of both declarative specifications and procedural ones. Dynamic dispatching of incoming calls is a primary function of generated adapters. Depending on the specifications, adapters facilitate synchronous or asynchronous communications. These adapters support binding of all properties including ones not announcing their changes.

This solution takes advantage of component hierarchies. The hierarchies regulate component participation in communication as well as call dispatching. A component is active when it listens to events or service requests and possibly generates events or requests. In communication hierarchies, a sub-component may only be active when its parent is. Events and other calls are always dispatched to the innermost relevant component. If several components belonging to different branches of a communication tree implement the same action, then the call is dispatched to all of them.

Features and Benefits

- Automatic generation of complex multi-threaded component communication code
- Declarative and procedural styles combined:
  Declarative specifications are supported by methods
- Generic approach to program specification and code generation directly in Java
  No additional specification languages, no extra tools required
- Component hierarchies enable highly compact specifications
  They regulate component participation in communications as well as call dispatching
- Thread safety for components not designed to be such if these components used with the adapters
- Dynamic inter-component call dispatching, filtering, multiplexing, demultiplexing
  with parameter/result type conversions
- Binding of all component properties including ones whose changes are not announced
- Solution for the J2EE platform: it is applicable to EJB or JSP components

When to Use This Approach

- Thread-safe / run-to-completion communication required

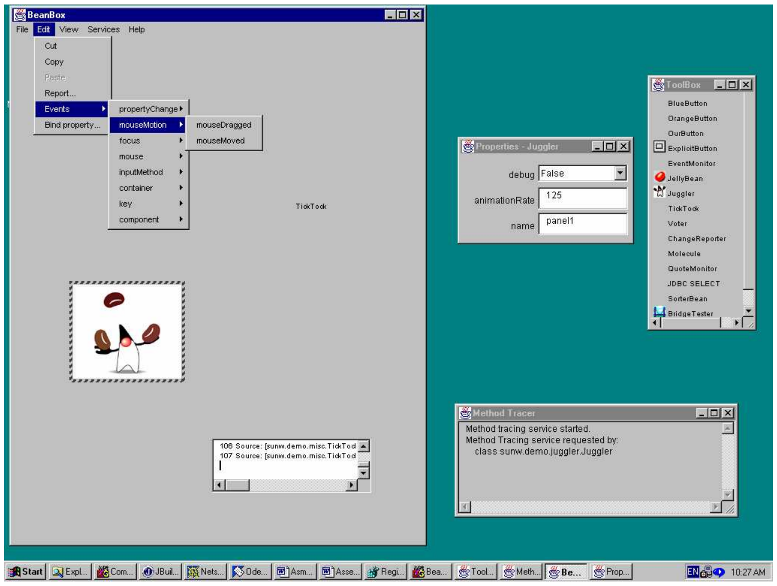
  or
- Asynchronous communication has to be implemented

  or
- Calls have to be dispatched (multiplexed) to different components

  or
- Properties whose changes are not announced have to be bound

  or
- You want the benefit of automatic concurrency management by application servers
  in a non-clustered environment while sticking to servlets and JSPs only

BeanBox and ContextBox



```
106 Source: [sunw.demo.misc.TickTod
107 Source: [sunw.demo.misc.TickTod
```

Properties - Juggler

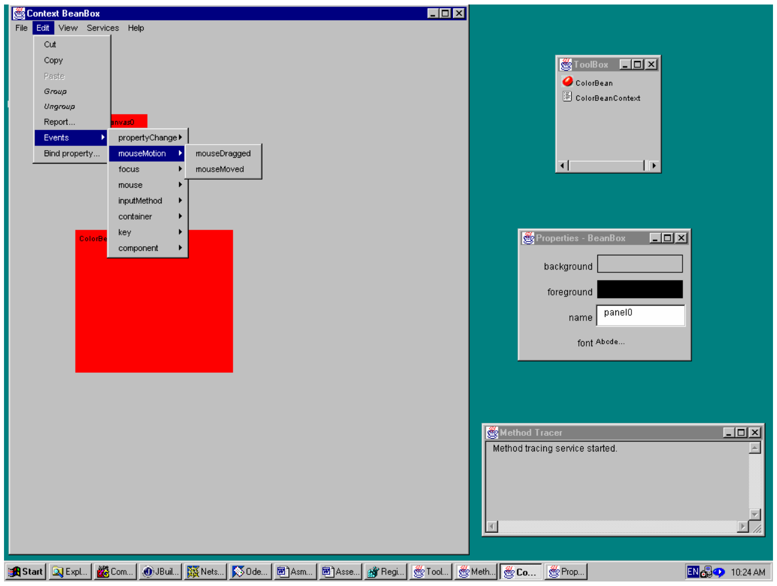
debug	False
animationRate	125
name	panel1

Method Tracer

```
Method tracing service started.
Method Tracing service requested by:
  class sunw.demo.juggler.Juggler
```

BeanBox

BeanBox and ContextBox let users bind event sources with event listeners. They generate an adapter for synchronous event delivery between the two components. These tools also allow users to bind properties of different components given that property changes are announced.



Properties - BeanBox

background	
foreground	
name	panel0
font	Abcde...

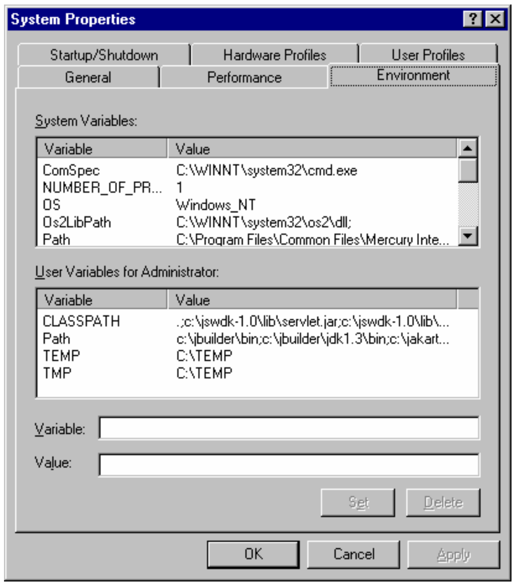
Method Tracer

```
Method tracing service started.
```

ContextBox

Component Hierarchy Examples

Visual hierarchy: System Properties in MS Windows



System Properties

Startup/Shutdown | Hardware Profiles | User Profiles
General | Performance | Environment

System Variables:

Variable	Value
ComSpec	C:\WINNT\system32\cmd.exe
NUMBER_OF_PR...	1
OS	Windows_NT
Os2LibPath	C:\WINNT\system32\os2\dll;
Path	C:\Program Files\Common Files\Mercury Inte...

User Variables for Administrator:

Variable	Value
CLASSPATH	.;c:\jswdk-1.0\lib\servlet.jar;c:\jswdk-1.0\lib\...
Path	c:\jbuilder\bin;c:\jbuilder\jdk1.3\bin;c:\jakart...
TEMP	C:\TEMP
TMP	C:\TEMP

Variable:

Value:

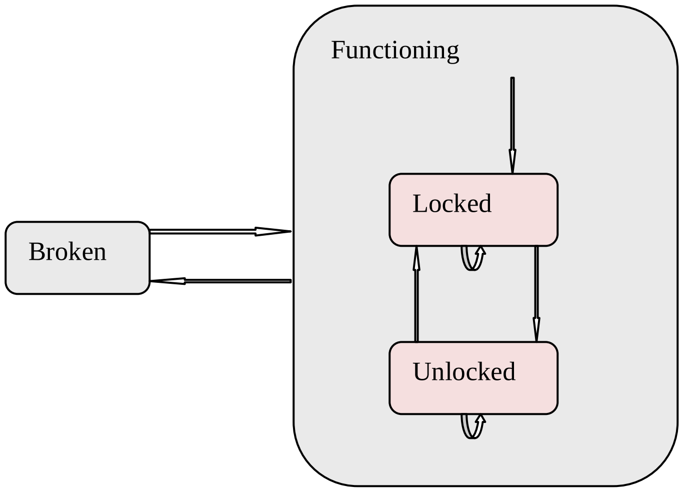
Set | Delete

OK | Cancel | Apply

Each tab is a sub-component of the entire ‘System Properties’ window. Two windows ‘System Variables’ and ‘User Variables’, and two text boxes are sub-components of the ‘Environment’ tab.

Hierarchical state machine: Slot Machine

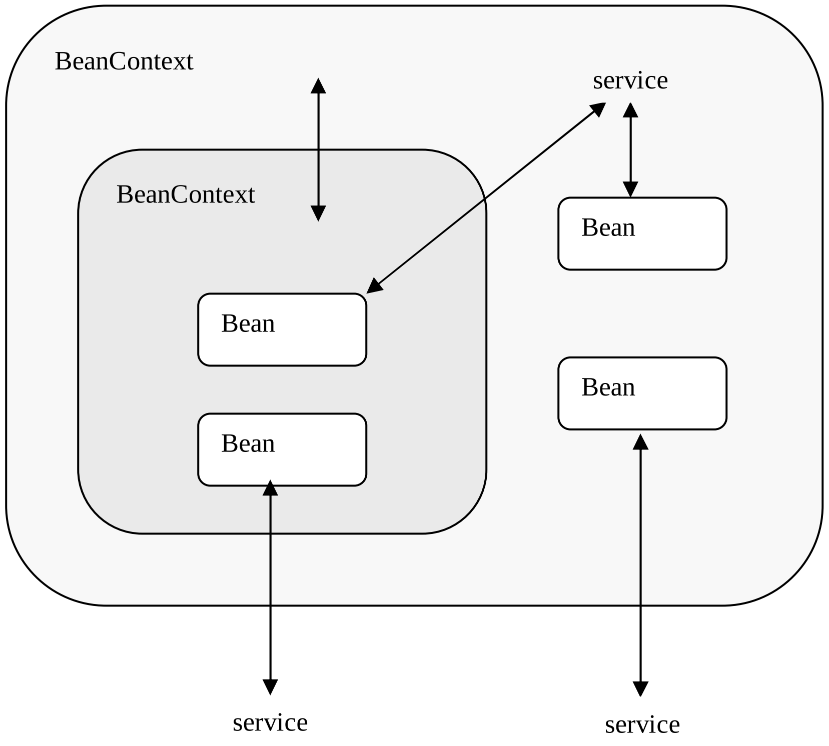
State ‘Functioning’ has two sub-states: ‘Locked’ and ‘Unlocked’.



Component Hierarchy Examples (continued)

Sample BeanContext hierarchy

BeanContext provides API for hierarchical nesting of BeanContext and JavaBean instances.



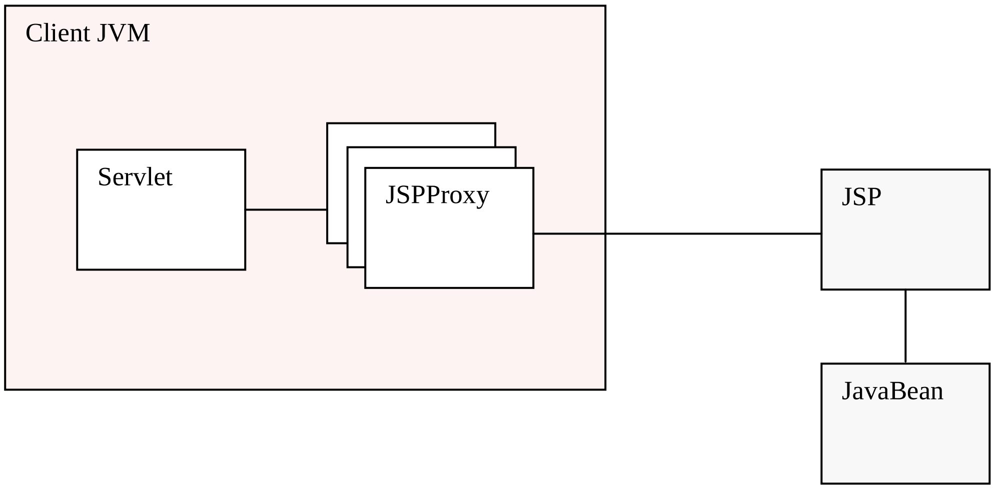
JSP components

JSPs may be collocated with their client servlet or may reside remotely. JSPProxy instances are components that represent JSPs on the client JVM. The JSP is JSPProxy constructor's parameter:

```
public JSPProxy(String page) {
    this.jsp = page;
}
```

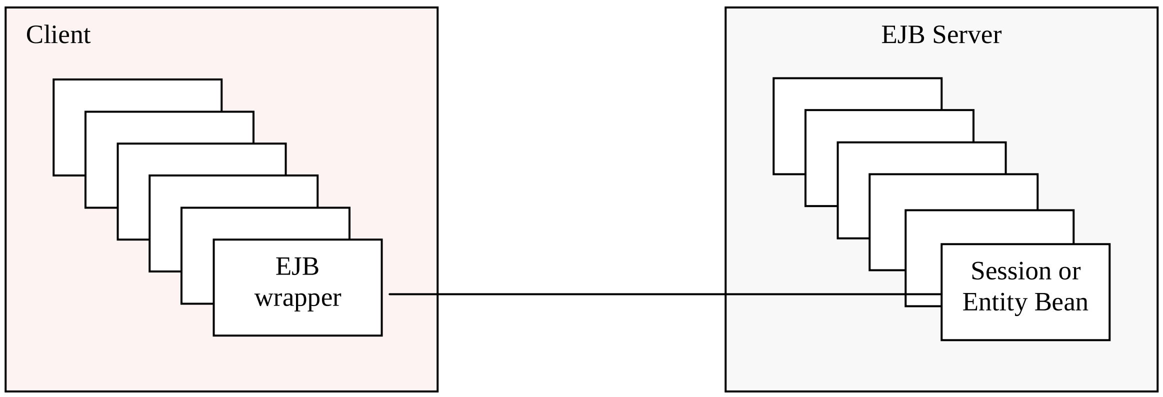
JSPProxy connects to the JSP:

```
public void forward(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException {
    RequestDispatcher dispatcher= req.getRequestDispatcher(this.jsp);
    dispatcher.forward(req, resp);
}
```

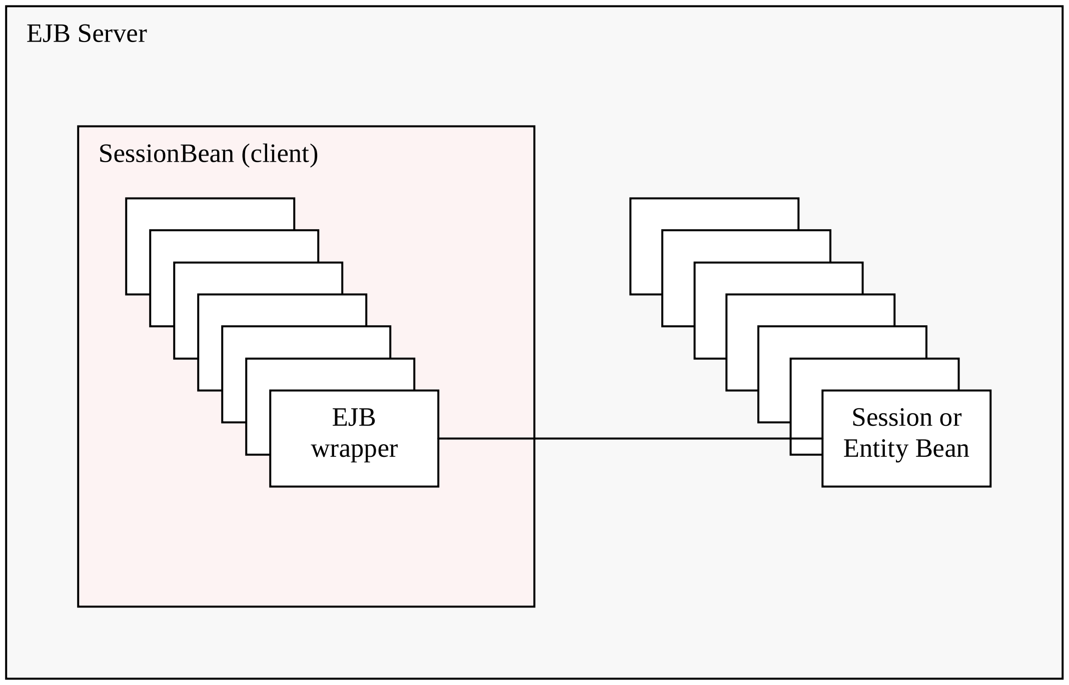


EJBs tied with one client - examples

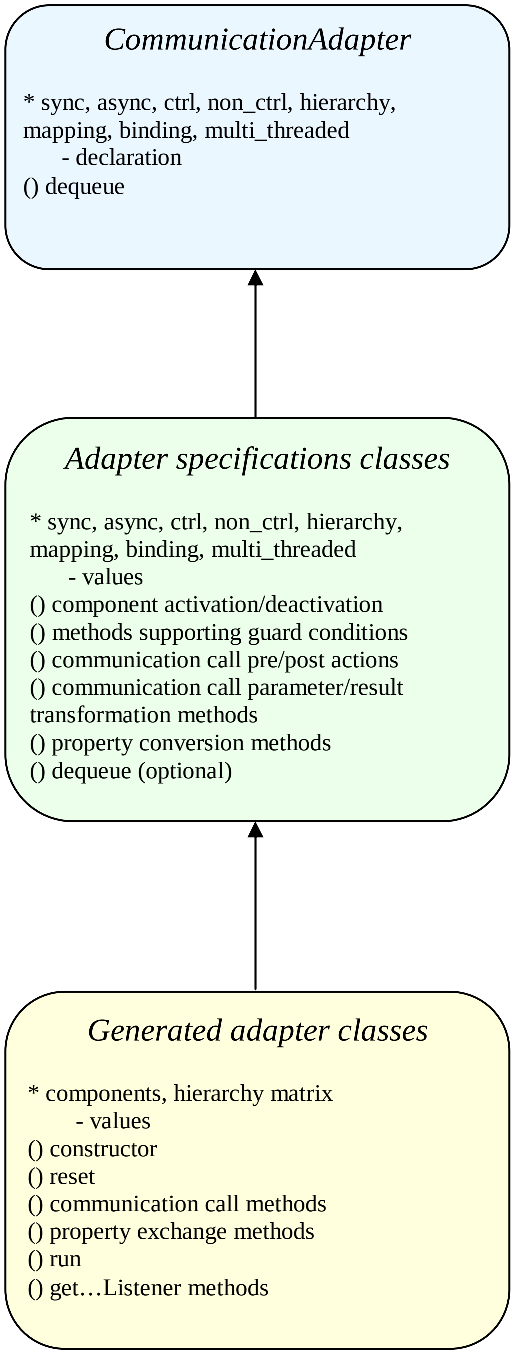
A client application such as a servlet may access EJBs via EJBObject wrapper class.



Session beans may access other EJBs in the same manner via EJBObject wrapper class.



Adapter Class Hierarchy



CommunicationAdapter
* sync, async, ctrl, non_ctrl, hierarchy, mapping, binding, multi_threaded
- declaration
() dequeue
Adapter specifications classes
* sync, async, ctrl, non_ctrl, hierarchy, mapping, binding, multi_threaded
- values
() component activation/deactivation
() methods supporting guard conditions
() communication call pre/post actions
() communication call parameter/result transformation methods
() property conversion methods
() dequeue (optional)
Generated adapter classes
* components, hierarchy matrix
- values
() constructor
() reset
() communication call methods
() property exchange methods
() run
() get…Listener methods

Communication Specifications

Declarative:

- Interface classification: synchronous vs asynchronous
- Component classification: controlled vs non-controlled
- Hierarchy description
- Event/method mappings
- Property binding specifications
- Multi or single-thread flag

Procedural:

- Methods referred to from the declarative specifications
- Code substantiating expressions from the declarative specifications

Sample communication specification:

```
public class SampleAdapterSpecification extends CommunicationAdapter { ...
    static { ...
        sync = new String[] { “Interface11”, “Interface12.Id”, … };
        async = new String[] { “Interface21”, “Interface22”, … };
        ctrl = new String[] { “Class11.Id1”, “Class12”, … };
        non_ctrl = new String[] { “Class21.Id2”, “Class22”, … };
        hierarchy = new String[][] { { “Class11.Id1”, “isId1Active() ” “activate”, “passivate”, “Class12”, … },
            { “Class12”, “”, “”, “”, “Class13”, … },
            ... };
        mapping = new String[][] { { “Interface11”, “method11 ”, “preAction”, “postAction”,
                “{“, “Class12”, “method2”, “filter()==0”, “transformParams”, “transformResult” “}”, … },
                { “Interface21”, “method21 “”,
                “{“, “Class21.Id2”, “method21”, “guard11()&&flag”, “”, “”, “}”, … },
            ... };
        binding = new String[][] { { “Class11.Id1”, “property11 ” “adapterProperty11”, “in”, “convertTo”, “convertFrom” },
            ... };
          multi-thread = true;
    }
    // methods
    protected boolean isId1Active() { ... )
    ...
    // data
    private boolean flag;
    private integer adapterProperty11;
    ...
}
```

Adapter Generation

Class AdapterGenerator is a code generator that builds communication adapter classes as extensions of their specification classes. The adapter's constructor takes a hash table of component objects keyed on their identifiers as the only argument. The adapter has a method to reset these objects. The generated adapters implement listener and other communication interfaces. The adapter should be registered as a sole communication listener via add…EventListener, BeanContextServiceSupport, etc. If there are listener method name conflicts, inner classes are generated. The respective identifiers become their names, and instances of these inner classes should be registered instead. Adapter's methods get<Identifier>Listener return the instances to register.

The adapters queue events (and other calls) of asynchronous communication interfaces. The adapters are Runnable in presence of asynchronous interfaces. The adapters construct and fire multiple threads when one call goes to several components simultaneously. The generated code mostly consists of fragments dispatching and filtering calls. The adapters also serve as property containers; they perform and monitor property updates induced by the specifications. Property changes take place right after fully processing each communication call. The adapters convert property types and communication call parameters/results when transformation methods are specified.

An adapter method implementing a method of an asynchronous communication interface:

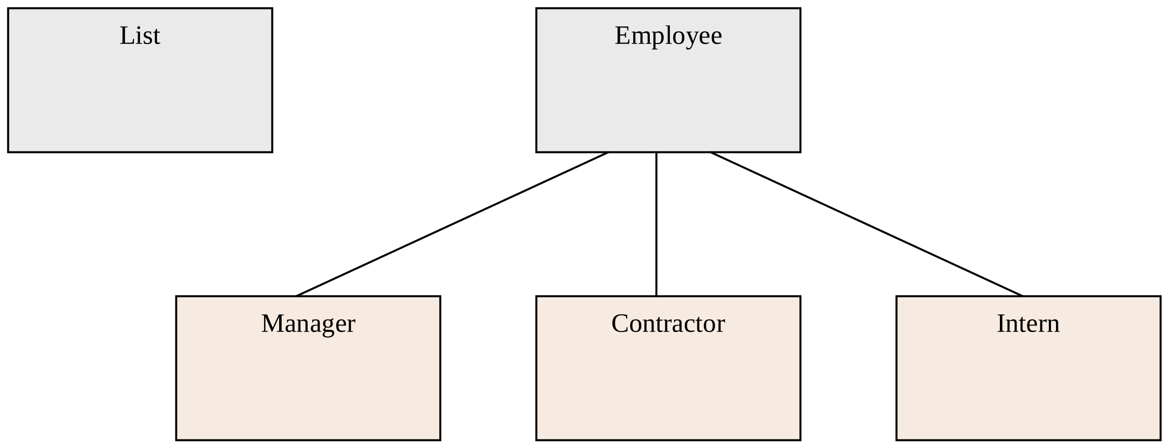
```
public synchronized void foo(Parameter param) {
    eventQueue.addElement(param);                        // parameter queuing for controlled
components
     eventIndexQueue.addElement(new Integer(N));

    ro = new RunNCComponentFoo(param);                // non-controlled component invocation
    th = new Thread(ro);
    th.start();
    ...
}
```

The core of method run of a generated adapter:

```
switch( callIndex ) { ...
    case M :                              // controlled components only
        if ( !blockComponentX ) {
            blockComponentY = true; ...
            componentX.foo(param);        or              roX = new RunComponentXFoo(param);
                                          |               thX = new Thread(roX);
                                          |               thX.start();
        }
                                           ...
                                          |               thX.join();
    break;
}
updateProperties();
```

JSP example from '*Web Development with Java Server Pages' (modified)*



Specifications:

```
sync = new String[] { "ForwardRequest" };
ctrl = new String[] { "List", "Employee", "Manager", "Contractor", "Intern" };
hierarchy = new String[][] { { "List" },
    { "Employee", "", "", "", "Manager", "Contractor" , "Intern" },
    { "Manager" }, { "Contractor" }, { "Intern" }, };
mapping = new String[][] { { "ForwardRequest", "forward ", "getId",
    "{", "List", "forward", "id==null" "}", "{", "Employee", "forward", "id!=null", "}",
    "{", "Manager", "forward", ", "id.equals(\"MNGR\")", "}",  "{", "Contractor", "forward", ", "id.equals(\"CNTR\")", "}",
    "{", "Intern", "forward", ", "id.equals(\"INTR\")", "}" } };
…
public void getId() { … }
```

Servlet's core:

```
doPost(HttpServletRequest req, HttpServletResponse res) {
     adapter.forward(req, res);
}
```

EJB Wrapper Generation

Sample home interface methods:

```
   public Foo create(int id) throws CreateException, RemoteException;
   public Foo findByPrimaryKey(FooPK pk) throws FinderException, RemoteException;
```

Sample remote interface methods:

```
   public String getName() throws RemoteException;
   public void setName(String str) throws RemoteException;
   …
   public void process() throws RemoteException;
```

Sample wrapper:

```
public class FooWrapper implements FooHome, Foo {
   private FooHome home = null;
   private Foo remote = null;
   public FooWrapper(String name) throws RemoteException, NamingException;
   // home interface methods
   // remote interface methods
}
```

Client application employing EJB without a wrapper:

```
         Context jndiContext = getInitialContext();
         Object obj = jndiContext.lookup("ejb/FooHome");
         FooHome home = (FooHome) javax.rmi.PortableRemoteObject.narrow(obj, FooHome.class);
         Foo remote = home.create(1);
         remote.setName("…");
```

Client application using the wrapper:

```
         FooWrapper wrapper = new FooWrapper("ejb/FooHome”);
         wrapper.create(1);
         wrapper. setName("…");
```