nemo

ontology & conceptual modeling research group

Web Development in Java
Part II

Vítor E. Silva Souza

(vitorsouza@inf.ufes.br)

http://www.inf.ufes.br/~vitorsouza

Department of Informatics

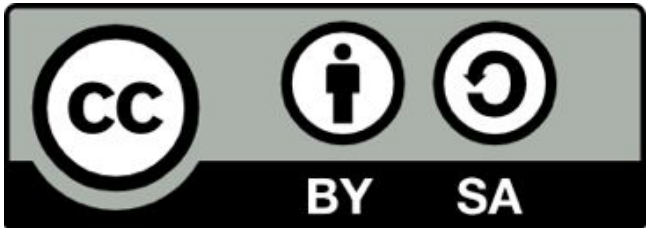
Federal University of Espírito Santo (Ufes),

Vitória, ES – Brazil

License for use and distribution

- This material is licensed under the Creative Commons license Attribution-ShareAlike 4.0 International;
- You are free to (for any purpose, even commercially):
  - Share: copy and redistribute the material in any medium or format;
  - Adapt: remix, transform, and build upon the material;
- Under the following terms:
  - Attribution: you must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use;
  - ShareAlike: if you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

More information can be found at:
http://creativecommons.org/licenses/by-sa/4.0/



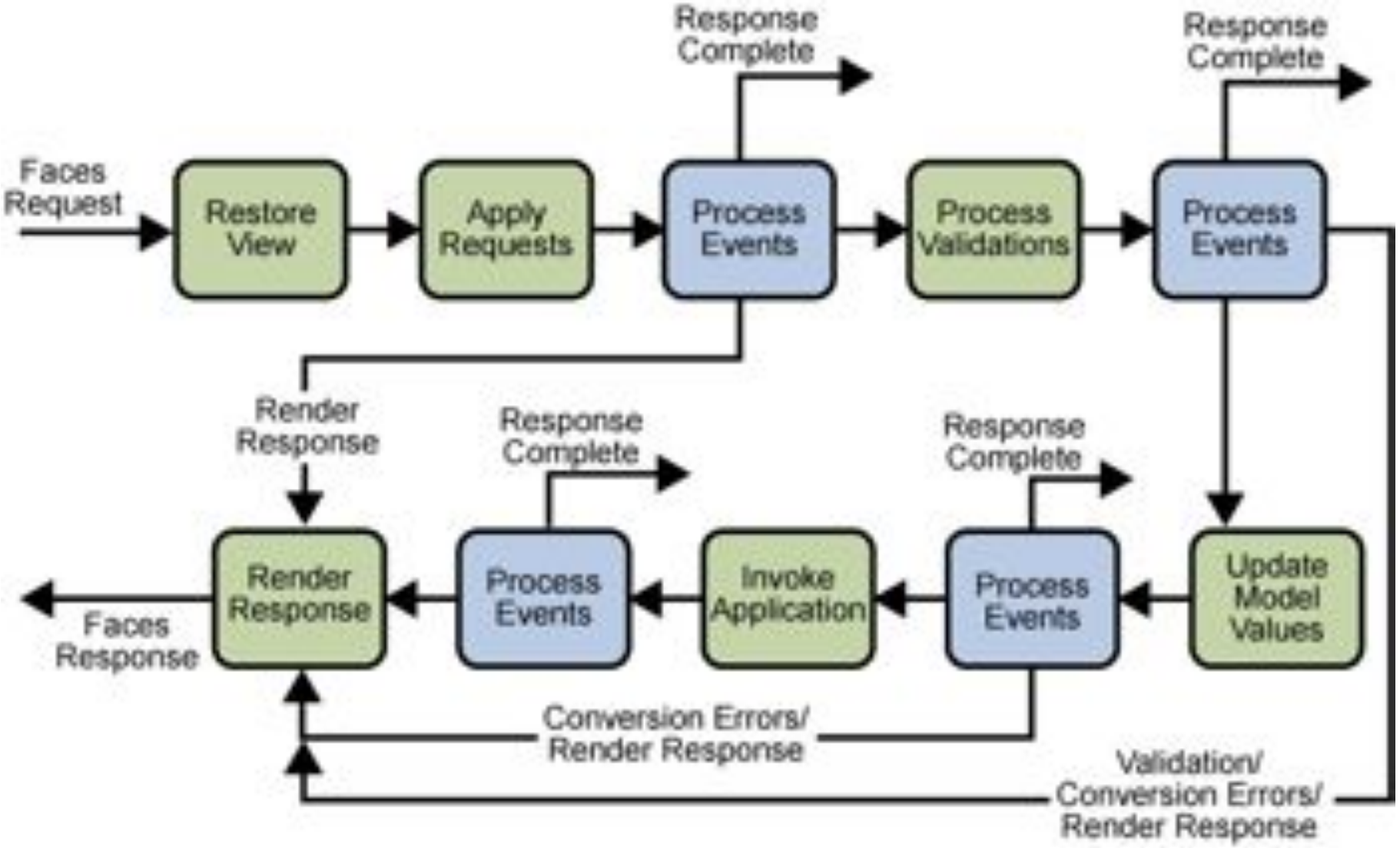
Outline of part II

- More on JSF: AJAX support, converters, formatters, i18n, Facelets components, etc.;
- More on CDI: scopes, producers, qualifiers, interceptors, decorators, events, etc.

Web Development in Java – Part II

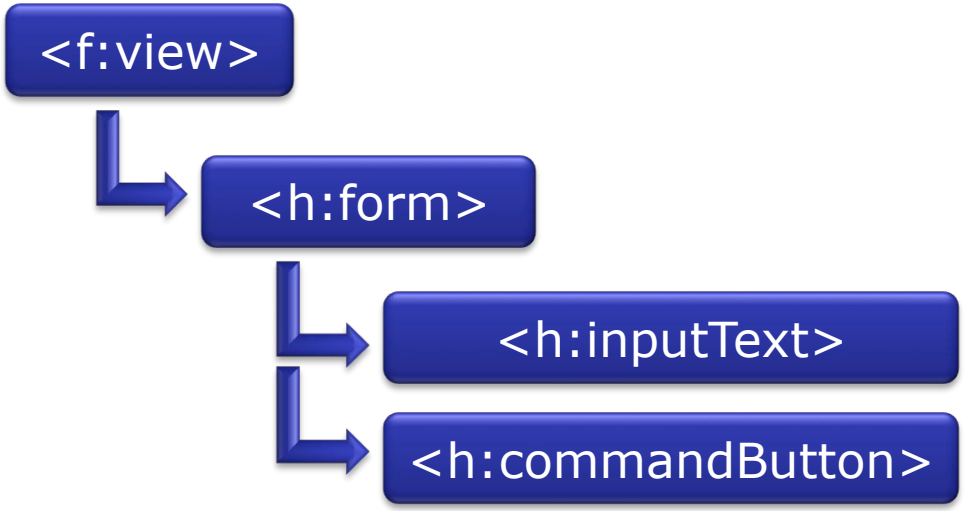
JAVASERVER FACES

The JSF lifecycle



The JSF lifecycle

- **Restore view** phase:
  - Build the view;
  - Wire event handlers and validators to components;
  - Save view in faces context;
  - For initial requests, view is empty, render response;



The JSF lifecycle

- **Apply request values** phase:
  - Extract values from components;
  - Convert them (generate error messages if needed);
  - Process validation, conversion and events of immediate components;
- **Process validations** phase:
  - Processes all components' validators;
  - Generate error messages and advance to render response if any values are invalid;
- **Update model values** phase:
  - Update bean attributes with components' values;
  - Generate error messages if conversion fails;

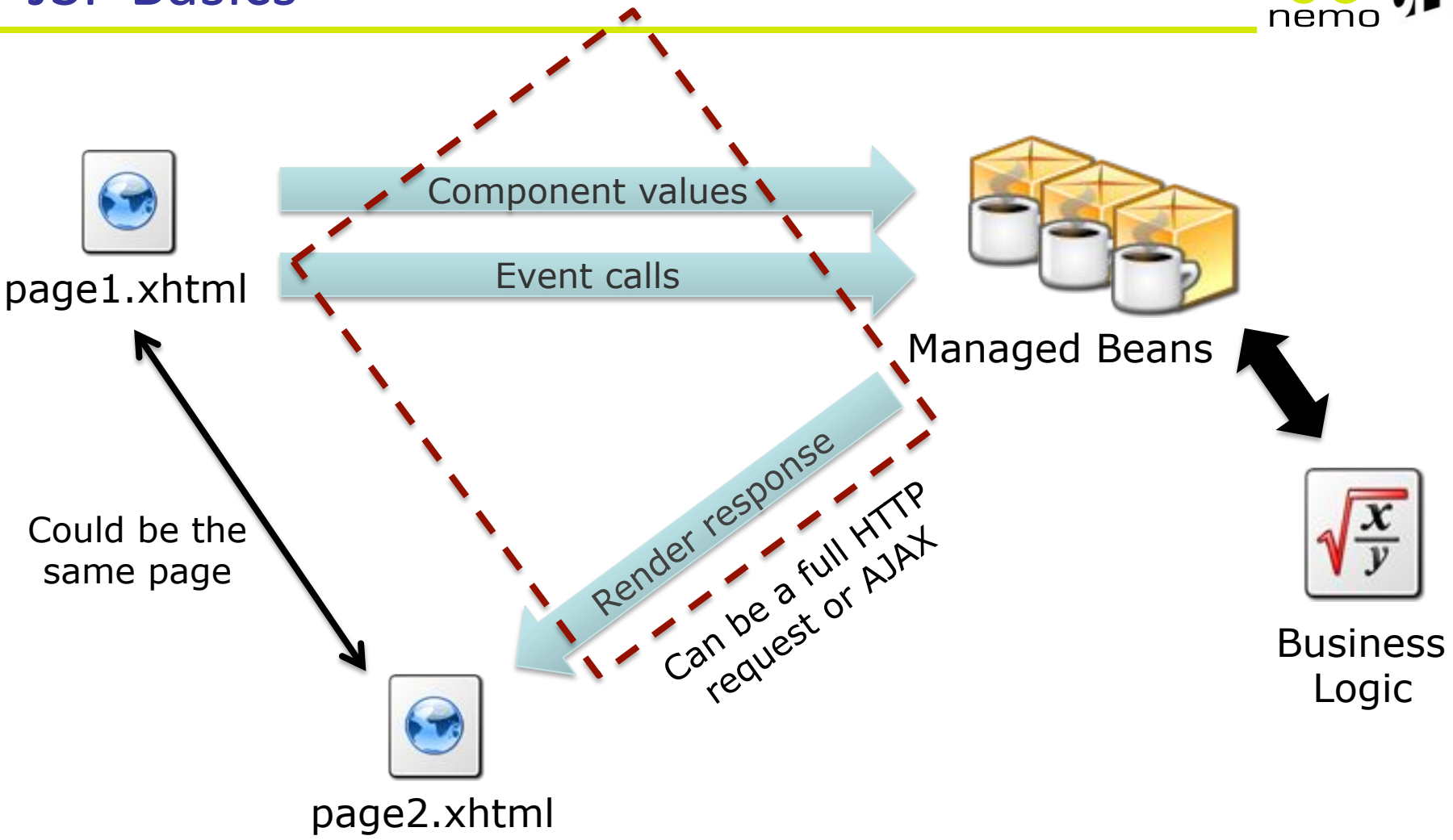
The JSF lifecycle

- **Invoke application** phase:
  - Handle application-level events (e.g. form submit);
  - Fire component events to listeners;
- **Render response** phase:
  - Render the response;
  - If postback request:
    - Response is rendered based on component tree;
    - If errors happened, render the original page with messages (<h:message /> or <h:messages /> required);
  - Save the state of response to be restored later.

The good news is…

- The lifecycle is handled transparently by JSF;
- Most JSF developers don’t need to concern themselves with it.

JSF Basics


page1.xhtml
Component values
Event calls
Managed Beans
Could be the same page
Render response
Can be a full HTTP request or AJAX
Business Logic
page2.xhtml

(Some) JSF standard components

<h:commandButton />

<h:commandLink />

<h:selectOneMenu />

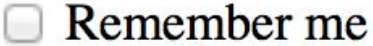
Your Shopping Cart	
Item Description	Price
Apple	1.0
Orange	0.5
Banana	1.25
Total: 2.75	

<h:dataTable />

<h:inputText /> and
<h:inputSecret />

<h:selectManyListbox />

<h:selectOneRadio />



<h:selectBooleanCheckbox />

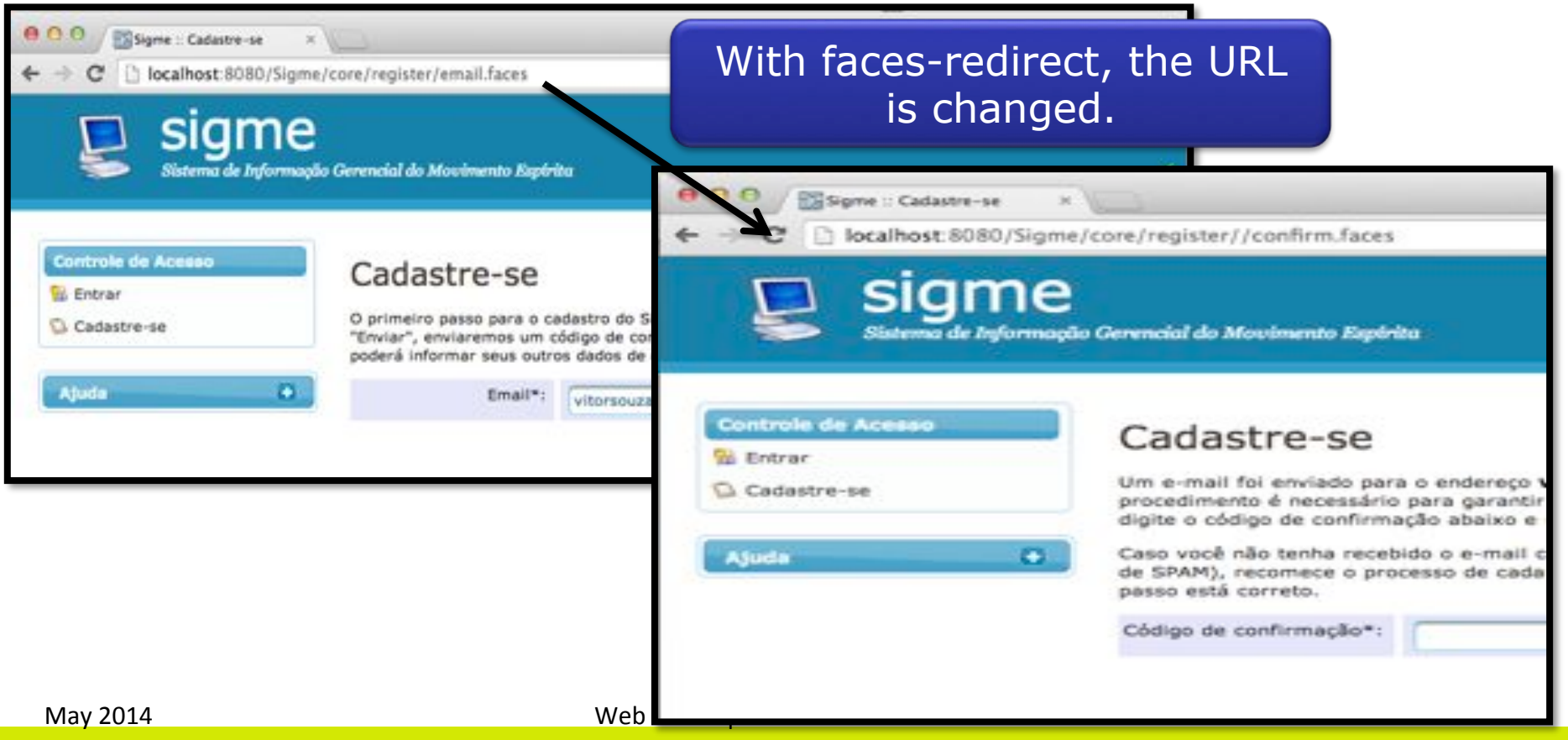
Source: http://www.jsftoolbox.com/documentation/help/12-TagReference/index.jsf

Page rendering vs. redirection

```
public String send() {
   // [...]
   catch (EmailAlreadyRegisteredException e) {
      return "/core/register/taken.xhtml";
   }
}
```

Page rendering vs. redirection

```
public String send() {
   // [...]

   return "/core/register/confirm.xhtml?faces-redirect=true";
}
```



AJAX support

- JSF 2 has AJAX support, no need for external tools;
- Tag **<f:ajax />**. Attributes:
  - **event**: which event should trigger the request: action, blur, change, click, dblclick, focus, keydown, keypress, keyup, mousedown, mousemove, mouseout, mouseover, mouseup, select;
  - **listener**: method to execute when the event occurs;
  - **execute**: data to submit in the request: @all, @none, @this, @form, component IDs;
  - **render**: what should be redrawn: @all, @none, @this, @form, component IDs.

AJAX examples

```
<h:form id="form">
  <h:commandButton action="#{myBean.doSomething()}"
    value="Do Something">
    <f:ajax render=":form:anotherComponent" />
  </h:commandButton>

  <h:panelGroup id="anotherComponent">
    <!-- ... -->
  </h:panelGroup>
</h:form>
```

Using the full name reference of a component.

```
<h:form id="form">
  <h:inputText id="value" value="#{myBean.value}" />

  <h:commandButton action="#{myBean.doSomething()}"
    value="Do Something">
    <f:ajax render=":form:anotherComponent" execute="value" />
  </h:commandButton>
</h:form>
```

Local name reference. Must be in the same form.

AJAX examples

```
<h:form id="form">
  <h:inputText id="name" value="#{myBean.obj.name}">
    <f:ajax event="blur" render="acronym"
      listener="#{myBean.suggestAcronym}" />
  </h:inputText>

  <h:inputText id="acronym" value="#{myBean.obj.acronym}" />

  <!-- ... -->
</h:form>
```

```
@Named
public class MyBean {
  private DomainObject obj = new DomainObject();
  public DomainObject getObj() { return obj; }

  public void suggestAcronym(AjaxBehaviorEvent event) {
    String acronym = /* Calculate an acronym based on obj.name. */
    obj.setAcronym(acronym);
  }
}
```

CDI vs. JSF scopes

- CDI has scopes:
  - javax.enterprise.context.ApplicationScoped;
  - javax.enterprise.context.ConversationScoped;
  - javax.enterprise.context.SessionScoped;
  - javax.enterprise.context.RequestScoped;
- JSF also has scopes:
  - javax.faces.beans.ApplicationScoped;
  - javax.faces.beans.SessionScoped;
  - javax.faces.beans.ViewScoped;
  - javax.faces.beans.RequestScoped;
  - javax.faces.beans.NoneScoped.

We will favor CDI scopes.

Converters

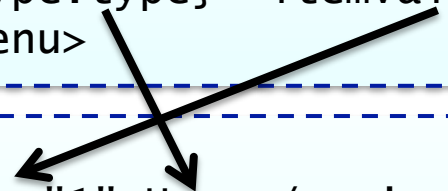
- HTML field values / HTTP parameters are Strings;
- JSF components are bound to objects of various classes;
- Standard converters handle simple types: characters, numbers and dates;
- Some of them can be configured:

```
<h:inputText value="#{myBean.myObject.updateDate}">
  <f:convertDateTime type="both" pattern="dd/MM/yyyy HH:mm:ss" />
</h:inputText>
```

Custom converters

- Sometimes you need to bind a field to an object:

```
<h:selectOneMenu id="telephoneType" value="#{myBean.contactType}"
converter="contactTypeConverter">
  <f:selectItems value="#{myBean.contactTypes}" var="type"
itemLabel="#{type.type}" itemValue="#{type}" />
</h:selectOneMenu>
```



```
<select>
  <option value="1">Home</option>
  <option value="2">Cell</option>
  <option value="3">Work</option>
</select>
```

- The above drop-down list renders an HTML select;
- Something must tell JSF how to convert the objects to their IDs (1, 2, 3) and vice-versa;
- Enter custom converters…

Custom converters

```
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.FacesConverter;

@FacesConverter("contactTypeConverter")
public class ContactTypeConverter implements Converter {
   @Override
   public Object getAsObject(FacesContext context, UIComponent
component, String value) {
      // Gets the element as String value (the ID);
      // Converts it to the object and returns.
   }

   @Override
   public String getAsString(FacesContext context, UIComponent
component, Object value) {
      // Gets the element as Object value;
      // Generates its String representation (ID) and returns.
   }
}
```

Validators

- Validation can start at the UI;
  - But you should also validate at your business logic;
- JSF standard validators:
  - Number range;
  - String length;
  - Regular expression;
  - Required fields.

```
<h:inputText id="email" value="#{myBean.email}" size="30"
     required="true" validatorMessage="Not a valid email address.">
  <f:validateRegex
        pattern="([^.@]+)(\\.[^.@]+)*@([^.@]+\\.)+([^.@]+)" />
</h:inputText>
```

Custom validator

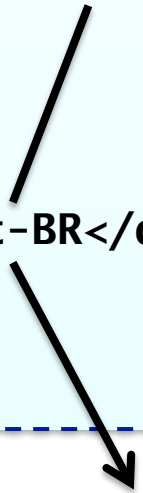
```
<h:inputText id="taxCode" value="#{myBean.client.taxCode}" size="20"
required="true">
  <f:validator validatorId="brazilianTaxCodeValidator" />
</h:inputMask>
```

```
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.*;

@FacesValidator("brazilianTaxCodeValidator")
public class BrazilianTaxCodeValidator implements Validator {
  @Override
  public void validate(FacesContext context, UIComponent component,
Object value) throws ValidatorException {

     // Verifies if value is a valid Brazilian CPF.
     // If not, use context.addMessage(FacesMessage), then
     // throw a new ValidatorException().

  }
}
```

Internationalization (i18n)

- As Java itself, JSF works with resource bundles:

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config ...>
<application>
   <resource-bundle>
      <base-name>br.org.feees.sigme.messages</base-name>
      <var>msgs</var>
   </resource-bundle>

   <locale-config>
      <default-locale>pt-BR</default-locale>
   </locale-config>
</application>
</faces-config>
```



- Given the above configuration, JSF will try to load:
  - /br/org/feees/sigme/messages_pt_BR.properties

Internationalization (i18n)

- You can have translations of your messages:

```
text.backToIndex = Voltar à página inicial
```

messages_pt_BR

```
text.backToIndex = Back to the home page
```

messages_en_US

```
text.backToIndex = Torna alla pagina iniziale
```

messages_it

- Then, at your web page (XHTML):

```
<p align="center"><a
href="#{facesContext.externalContext.requestContextPath}/">
<h:outputText value="#{msgs['text.backToIndex']}" /></a></p>
```

- Or at the bean/controller (Java):

```
ResourceBundle bundle = FacesContext.getCurrentInstance()
                .getApplication().getResourceBundle(context, "msgs");
String message = bundle.getString("text.backToIndex");
```

Facelets components

- Generic components reusable across projects should be developed as JSF components (e.g., PrimeFaces);
- Components reusable in a single project can be created using Facelets compositions (mini-templates):

```
<ui:composition ...>
  <div class="formField #{(fieldName == null or empty facesContext
     .getMessageList(fieldName)) ? '' : 'formFieldError'}">
    <div class="label">
     <ui:insert name="label" /><h:panelGroup styleClass="star"
rendered="#{(fieldName != null and facesContext.viewRoot
        .findComponent(fieldName).required)}">*</h:panelGroup>:
    </div>
    <div class="field">
     <p:message for="#{fieldName}" rendered="#{fieldName != null}" />
     <ui:insert /><p:tooltip for="#{fieldName}" value="#{tooltip}"
rendered="#{tooltip != null}" showEvent="focus" hideEvent="blur" />
    </div><div class="clear"></div>
  </div>
</ui:composition>
```

Facelets components

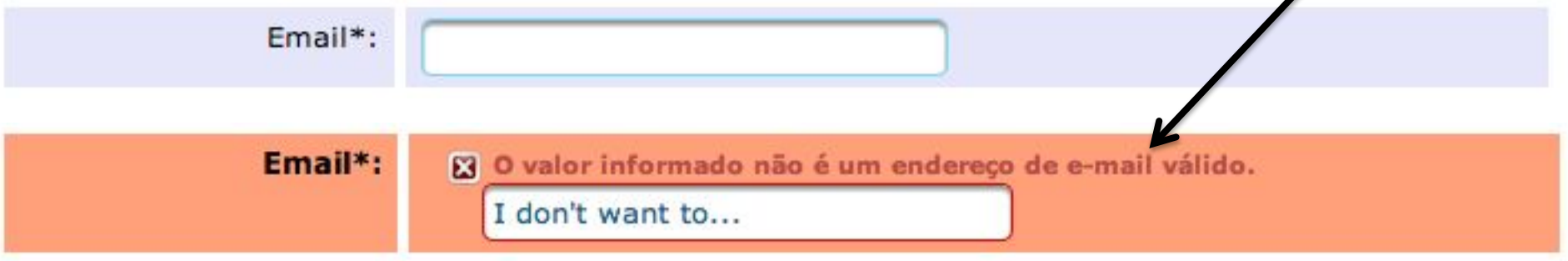
- Then, at the web page:

```
<h:panelGroup id="emailField">
  <ui:decorate template="/resources/templates/formfield.xhtml">
    <ui:param name="fieldName" value="form:email" />
    <ui:param name="tooltip" value="Type your e-mail." />
    <ui:define name="label"><h:outputText value="Email" /></ui:define>
    <p:inputText id="email" value="#{myBean.email}" size="30"
required="true" validatorMessage="...">
      <f:validateRegex pattern="..." />
      <p:ajax event="blur" update="emailField" />
    </p:inputText>
  </ui:decorate>
</h:panelGroup>
```

See slide 19

- The result (CSS not shown):

i18n set to pt_BR...

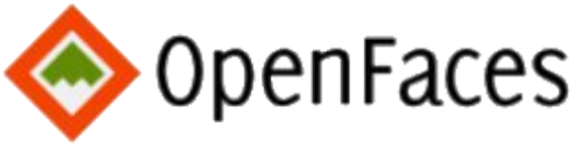


Error messages

- Validators / converters produce error messages;
  - Bound to a specific component ID;
  - Global (no component specified).
- We can display these in our pages with (respectively):
  - <h:message for="componentId" />
  - <h:messages globalOnly="true" />
- I suggest you place them at (respectively):
  - Your Facelets component for form fields;
  - Your Facelets decorator for the entire page.

Going forward…

- Study the standard JSF components;
- Choose an external JSF component library and familiarize yourself with their components;
- Start building your Web application and learn from practice!



Web Development in Java – Part II

CONTEXTS AND DEPENDENCY INJECTION FOR THE JAVA PLATFORM

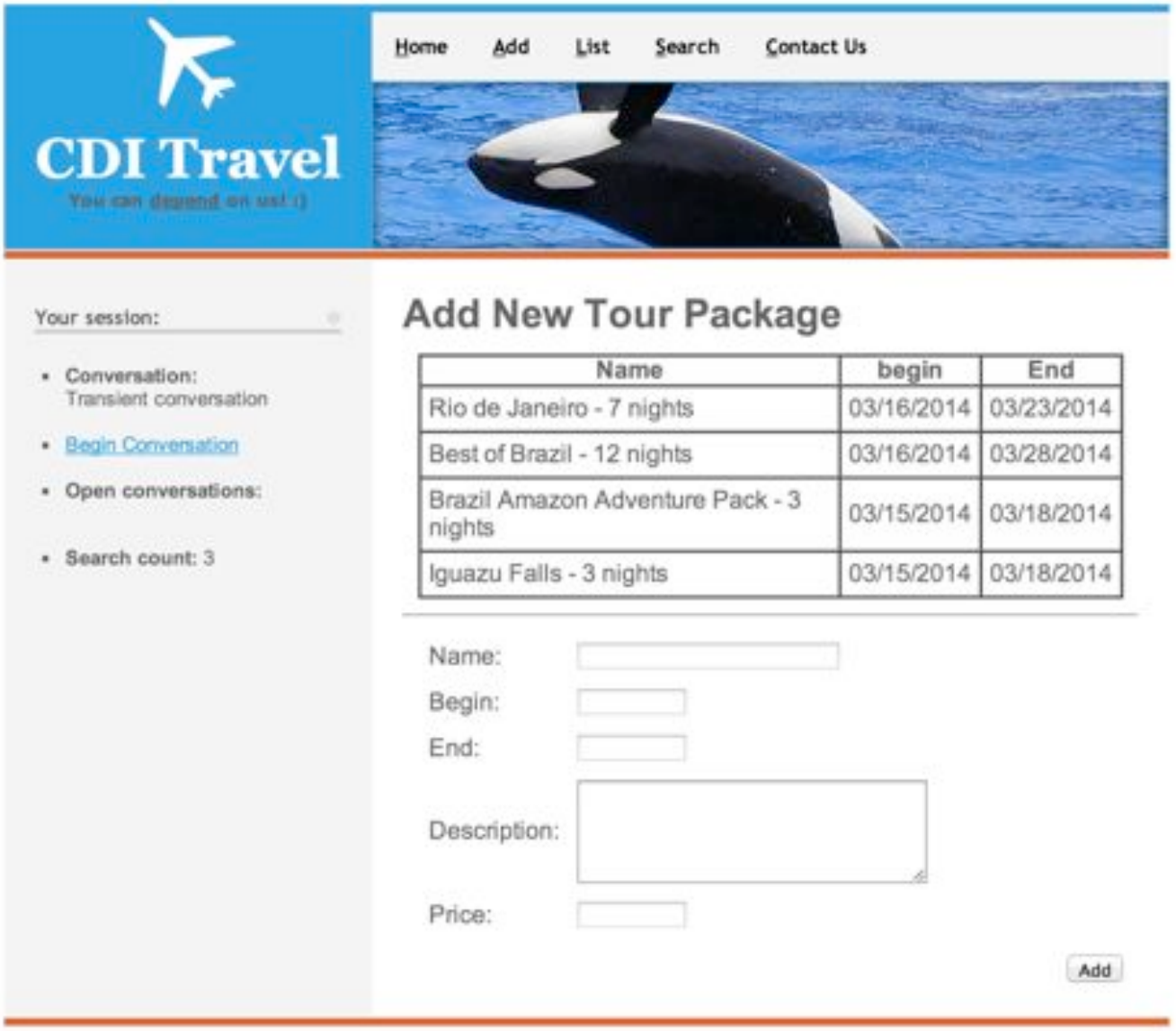
CDI goals (from its spec)

- A well-defined lifecycle for stateful objects bound to lifecycle contexts, where the set of contexts is extensible;
- A sophisticated, type-safe dependency injection mechanism, including the ability to select dependencies at either development or deployment time, without verbose configuration;
- Support for Java EE modularity and the Java EE component architecture—the modular structure of a Java EE application is taken into account when resolving dependencies between Java EE components;

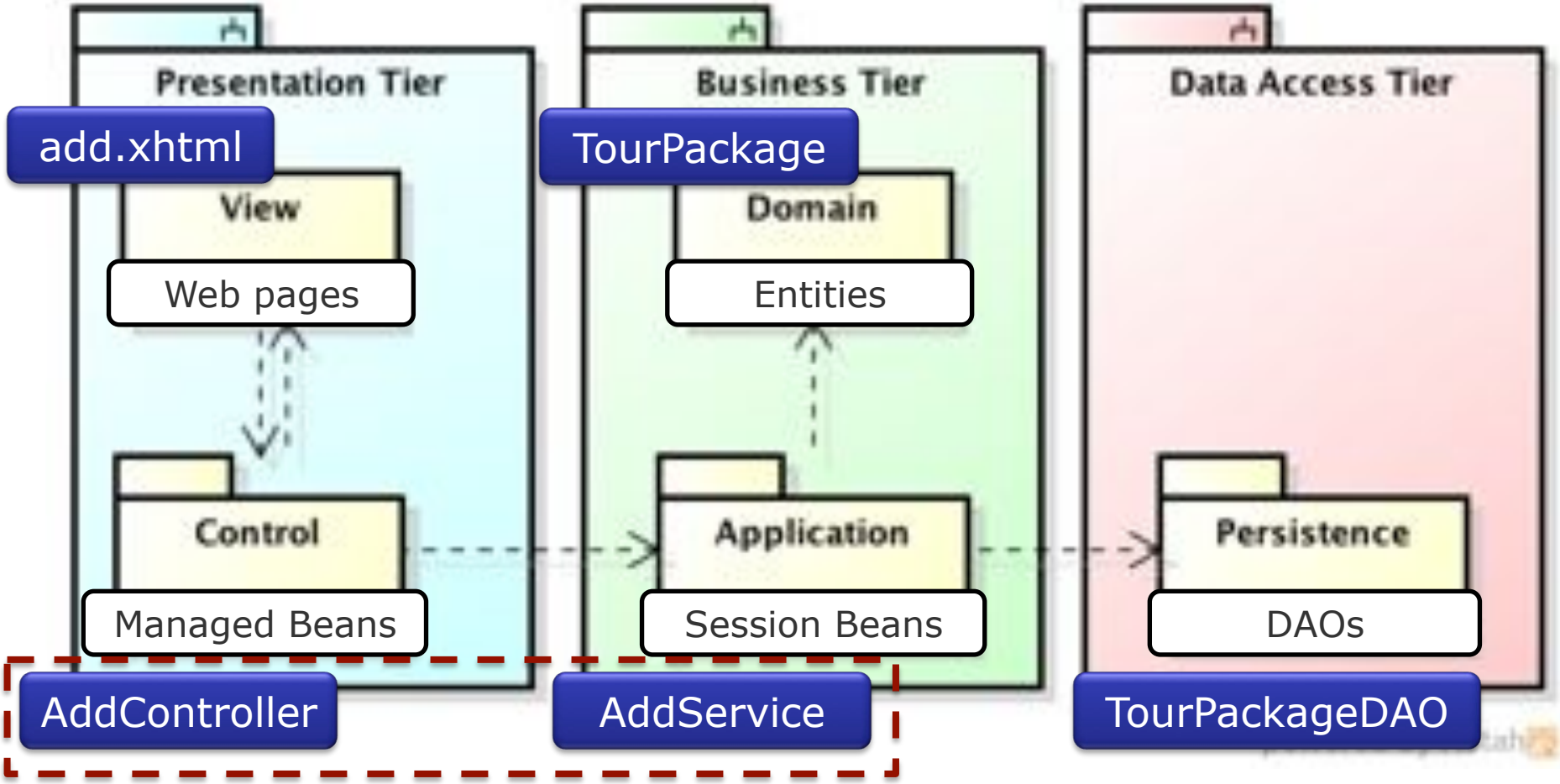
CDI goals (from its spec)

- Integration with the Unified Expression Language (EL), allowing any contextual object to be used directly within a JSF or JSP page;
- The ability to decorate injected objects;
- The ability to associate interceptors to objects via typesafe interceptor bindings;
- An event notification model;
- A web conversation context in addition to the three standard web contexts defined by the Java Servlets specification;
- An SPI allowing portable extensions to integrate cleanly with the container.

Imagine an information system…



Remember the proposed architecture…



For now, these two will be merged…

Start from the bottom: the DAO

```
@Local
public interface TourPackageDAO {
   long retrieveCount();
   List<TourPackage> retrieveAll();
   List<TourPackage> retrieveSome(int[] interval);
   TourPackage retrieveById(Long id);
   TourPackage save(TourPackage  object);
   void delete(TourPackage object);
   List<TourPackage> findByName(String name);
 }
```

Resource (EJB) declaration

```
@Stateless
public class TourPackageJPADAO implements TourPackageDAO {
   @PersistenceContext
   private EntityManager em;

   /* Implementation of the methods declared in the interface. */
}
```

Dependency Injection

DI annotations

- @PersistenceContext: the entity manager;
- @PersistenceUnit: the persistence unit;
- @WebServiceRef: reference to Web services;
- @Resources: miscellaneous Java EE resources;
- @EJB: Enterprise Java Beans;
- @Inject: your own objects.

The service + controller (buy 1, take 2)

```
@Stateful @LocalBean @Model
public class AddPackage {
   @EJB private TourPackageDAO tourPackageDAO;
   @Inject private LocaleBean loc;

   private List<TourPackage> packages;             // + getter
   private TourPackage pack = new TourPackage();  // + getter

   @Inject
   void loadPackages() {
      packages = tourPackageDAO.retrieveAll();
   }

   public String add() {
      tourPackageDAO.save(pack);
      packages.add(pack);
      pack = new TourPackage();
      return null;
   }
}
```

The @EJB annotation

- When an instance of AddPackage is needed:

```
public class AddPackage {
   @EJB private TourPackageDAO tourPackageDAO;
   // ...
```

- An instance of the only (default) implementation of the TourPackageDAO EJB is created/retrieved and injected:

```
@Local
public interface TourPackageDAO {
   // ...

@Stateless
public class TourPackageJPADAO implements TourPackageDAO {
   // ...
```

- No code needed. CDI does the job!

@Inject at an attribute

- Works like @EJB, but for POJOs (Plain Old Java Objects):

```
public class AddPackage {
   @Inject private LocaleBean loc;
   // ...
```

```
@ApplicationScoped
public class LocaleBean implements Serializable {
   // ...
```

- POJOs are associated with scopes…

CDI scopes

- javax.enterprise.context.ApplicationScoped:
  - One instance for the entire application;
- javax.enterprise.context.ConversationScoped:
  - An instance per conversation. Conversations are controller by the programmer (later we'll see how);
- javax.enterprise.context.SessionScoped:
  - One instance per user session (each connected browser gets a session, expires in time);
- javax.enterprise.context.RequestScoped:
  - One instance per HTTP request (AJAX or not).

@Inject at a method

- Defines an initializer method:

```
public class AddPackage {
  private List<TourPackage> packages;   // + getter

  @Inject
  void loadPackages() {
     packages = tourPackageDAO.retrieveAll();
  }

  // ...
```

  - This method is called every time an instance of AddPackage is created;
  - If it had parameters, they would have been injected. The DAO could have been a parameter, for instance.

@Inject at a constructor

- Like an initializer method, but using the constructor:

```
public class ListPackages {
   @Inject
   public ListPackages(NumberFormat nf) {
      this.nf = nf;
   }
   // ...
```

  - A NumberFormat (a class from the Java API!) must be **produced** and injected. Hold that thought…
  - Only one constructor can have @Inject;
  - If no constructors have it, CDI uses the default one to create the object.

Mind buffer:
- Conversations
- Producers

Being service and controller

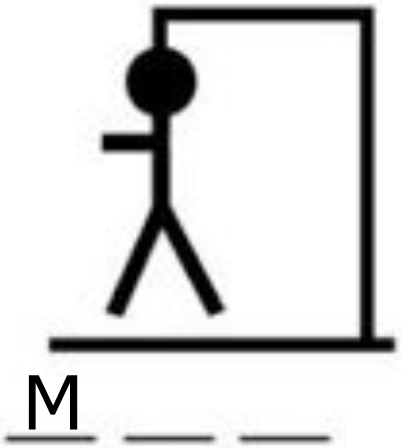
Service

Controller

```
@Stateful @LocalBean @Model
public class AddPackage {
   // ...
```

- @Stateful, @LocalBean, @Local, etc. are EJB stuff;
- @Model reminds you of anything?
- @Model is a stereotype:

```
@Named
@RequestScoped
@Documented
@Stereotype
@Target({ TYPE, METHOD, FIELD })
@Retention(RUNTIME)
public @interface Model {
}
```



The @Named annotation

- Allows a bean to be referred to via EL at a web page:

```
<h:outputText value="No package added yet."
rendered="#{addPackage.packages.size() == 0}" />
```

```
@Model
public class AddPackage {

   public List<TourPackage> getPackages() {
      return packages;
   }

// ...
```

If you don't like the default name, you can specify one in @Named

  - Looks for an instance at the specified scope;
  - If not found, creates a new one in the scope;
  - At the end of the scope, the instance is discarded.

Producers

- Beans can be injected (as seen);
- They can also produce the thing that will be injected:

Mind buffer:
- Conversations
- ~~Producers~~

```
@Stateless @Named("lp")
public class ListPackages {
   @Inject
   private List<TourPackage> packages;
   // ...
```

```
@ApplicationScoped
public class PackageProducer {
   @EJB private TourPackageDAO tourPackageDAO;

   @Produces
   public List<TourPackage> getAllPackages() {
      List<TourPackage> packages = tourPackageDAO.retrieveAll();
      return packages;
   }

   public void dispose(@Disposes List<TourPackage> packages) { }
}
```

Producers

- Attributes can also be producers:

```
@Stateless @Named("lp")
public class ListPackages {
  private NumberFormat nf;

  @Inject
  public ListPackages(NumberFormat nf) { this.nf = nf; }
  // ...
```

```
@ApplicationScoped
public class FormatterProducer {
  private static final Locale LOCALE = new Locale("en", "US");

  @Produces
  private final NumberFormat CF =
        NumberFormat.getCurrencyInstance(LOCALE);
```

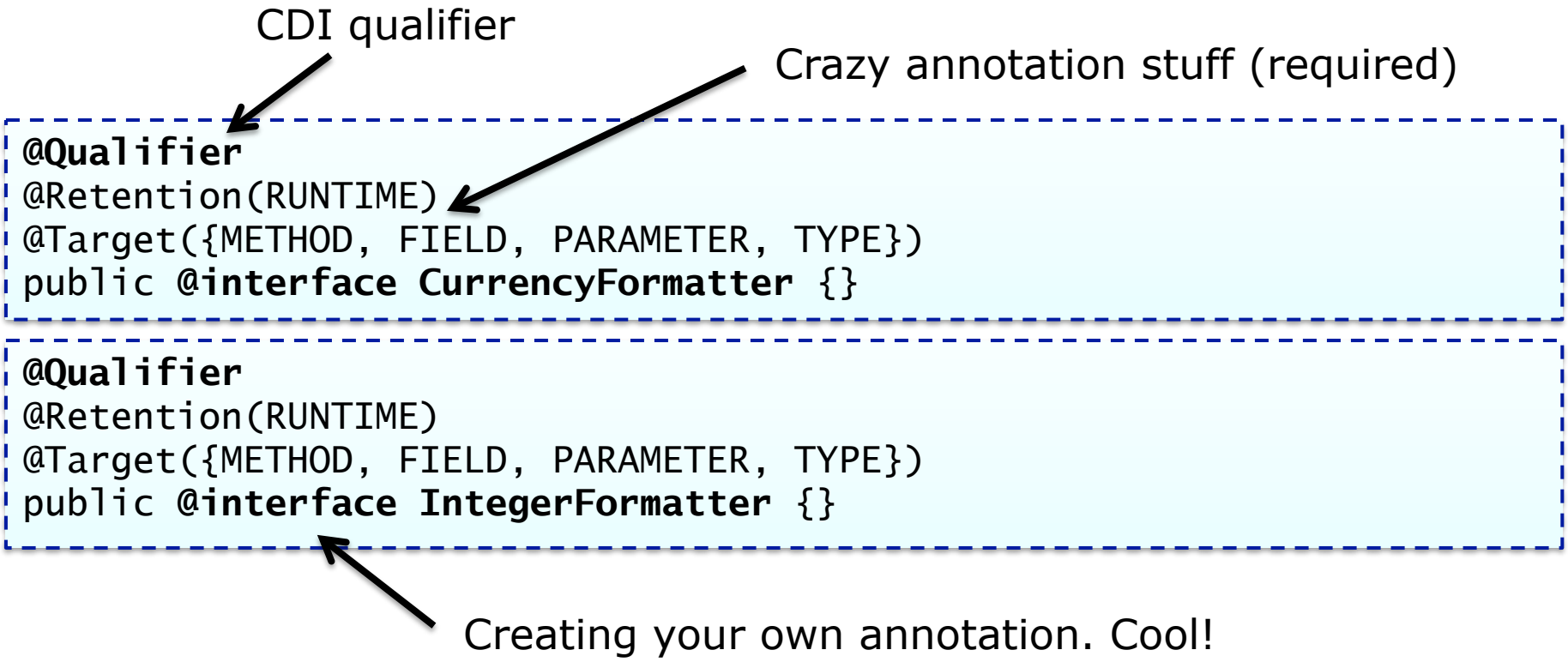
Recap: how does CDI finds things to inject?

- @Named things are found by their name in the EL;
- @EJBs are found by their interface and implementation;
- @Inject'ed objects are found:
  - If their class is directly associated with a scope, i.e., they are beans themselves;
  - If an attribute of their class is annotated with @Produces inside a bean;
  - If a method with their class as return type is annotated with @Produces inside a bean.

What if we need to produce different objects from the same class?

Qualifiers

- Producer methods can use polymorphism and determine which subclass to instantiate;
- However, if the class declaring the dependency knows the subclass it wants, we can use qualifiers:



Qualifiers

- Once you created the qualifiers, use them at both ends:

```
@Stateless @Named("lp")
public class ListPackages {
  private NumberFormat nf;
  private NumberFormat nfi;

  @Inject
  public ListPackages(@CurrencyFormatter NumberFormat nf,
        @IntegerFormatter NumberFormat nfi) { /* ... */ }
  // ...
```

```
@ApplicationScoped
public class FormatterProducer {
  private static final Locale LOCALE = new Locale("en", "US");

  @Produces @CurrencyFormatter private final NumberFormat CF =
     NumberFormat.getCurrencyInstance(LOCALE);

  @Produces @IntegerFormatter private final NumberFormat IF =
     NumberFormat.getIntegerInstance(LOCALE);

  // ...
```

Qualifiers

- Instead of new annotations, one could use:
  - @Named("currencyFormatter") on both ends;
  - @Named without the string, but in this case both variables must have the same name;
- New annotations are checked at compile time;
- @Named is checked at deploy time;
- XML configuration (old style) is checked at runtime;
- You can use as many qualifiers as you want;
  - @Inject @Q1 @Q2 … @Qn: CDI will look for the bean that has all the specified qualifiers.

Qualifiers

- Beans can also be qualified. Imagine, for instance:

```
@Stateless @JPADAO
public class TourPackageJPADAO implements TourPackageDAO {
   @PersistenceContext
   private EntityManager em;

   /* Implementation of the methods using JPA. */
}
```

```
@Stateless @JDBCDAO
public class TourPackageJDBCDAO implements TourPackageDAO {
   @Inject
   private Connection conn;

   /* Implementation of the methods using JDBC directly. */
}
```

Built-in qualifiers

- CDI comes with a few qualifiers ready to use:
  - @Default: optional, specifies the un-qualified dependency;
    - @Inject @Default == @Inject;
    - @Produces @Default == @Produces;
    - If the class produces two dependencies of the same class, only one of them can be the default (non-qualified).
  - @New: if you don't want to use the scope-related dependency. CDI creates a new one in the current scope;
  - @Any: the opposite of @New. Also optional;
    - @Inject @Any == @Inject.

Tired yet? Wait! There's more!

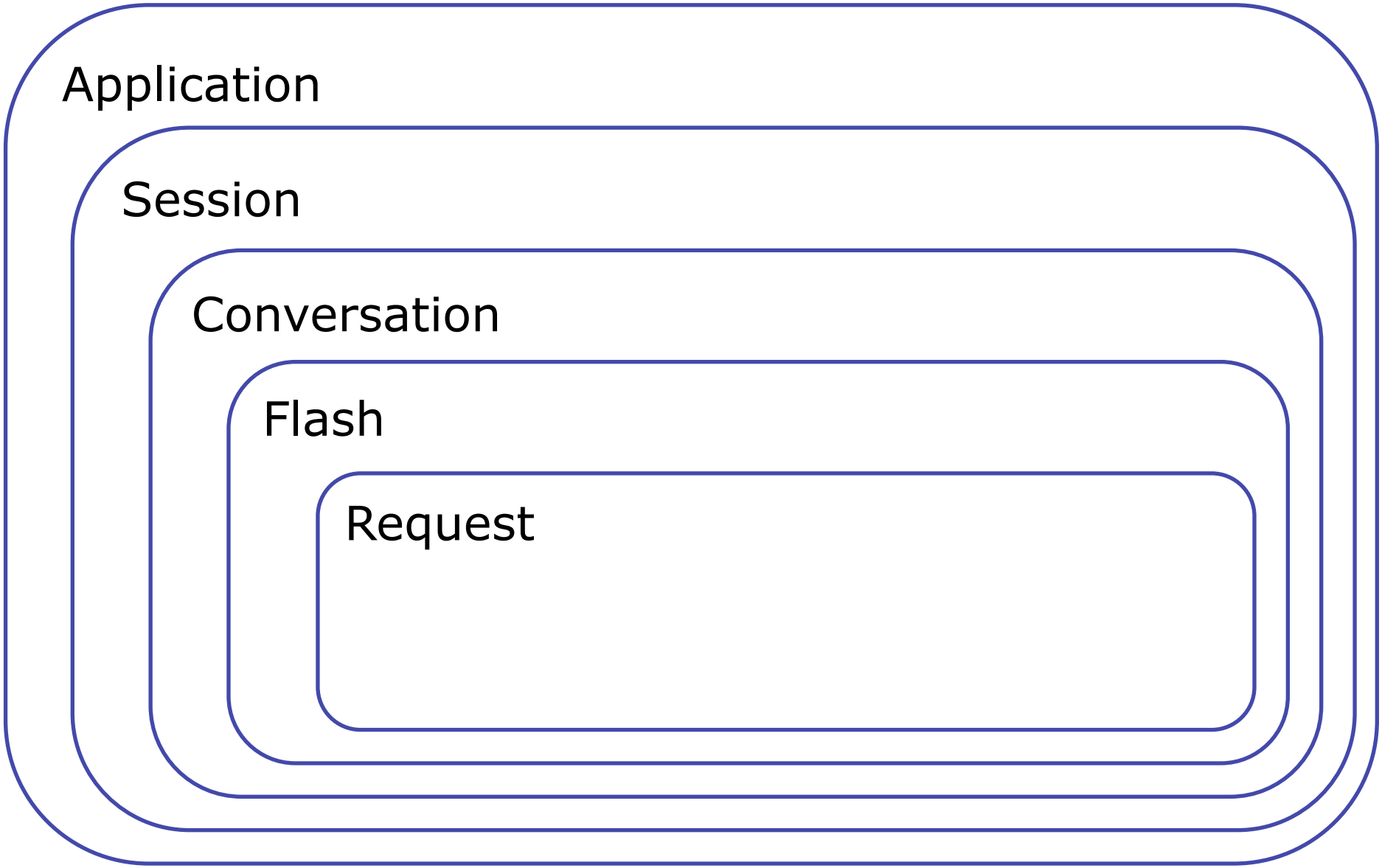
Back to contexts…

- Some components have no context: Singleton EJBs, stateless EJBs, Servlets, classes in general…;
- CDI beans have their lifecycle associated with a context:
  - Created automatically when needed;
  - Destroyed automatically when context ends;
  - Shared with other beans in the same context, if declared as dependencies.

Contexts are defined by the scope

- Some scopes are old friends of ours:
  - @RequestScoped: the HTTP request;
  - @SessionScoped: the HTTP session;
  - @ApplicationScoped: the entire application.
- Some are new, introduced by CDI:
  - @ConversationScoped: a conversation delimited by the programmer (specific calls to begin/end);
  - @Dependent pseudo-scope (and default): use the same scope as the bean declaring the dependency.
- JSF also created the Flash "scope";
- You can define new scopes (not included here).

Scopes inside other scopes



Passivation

- Some contexts hold user-related objects in memory for long periods of time (conversation, session);
- In case of need, the server will passivate these objects: serialize them to disk to free memory;
- For this reason, beans in this context must be serializable.

```
@SessionScoped
@Named
public class ConversationList implements Serializable {

   // ...

}
```

Conversation scope

- Every JSF request is bound to a conversation;
- Conversations are, by default, transient: begin and end with the request;
- Long-running conversations are more interesting:
  - Begin when begin() is called at the current transient conversation;
  - End (go back to transient) when end() is called at the current long-running conversation.
- Conversations belong to a single user session and are destroyed if the session is invalidated;
- A session can have multiple conversations.

Conversation handling

Your session:

- **Conversation:** Transient conversation
- Begin Conversation
- **Open conversations:**
- **Search count:** 3

Add New Tour Package

Name	begin	End
Rio de Janeiro - 7 nights	03/16/2014	03/23/2014
Best of Brazil - 12 nights	03/16/2014	03/28/2014
Brazil Amazon Adventure Pack - 3 nights	03/15/2014	03/18/2014
Iguazu Falls - 3 nights	03/15/2014	03/18/2014

Name:

Begin:

End:

Description:

Price:

Conversation handling

```
@RequestScoped
@Named("cvmg")
public class ConversationManager implements Serializable {
   @Inject private Conversation conversation;
   @Inject private ListConversations list;

   public Conversation getConversation() {
      return conversation;
   }

   public String beginConversation() {
      conversation.begin();
      list.add(conversation);
      return null;
}

   public String endConversation() {
      list.remove(conversation);
      conversation.end();
      return null;
   }
}
```

Conversation handling

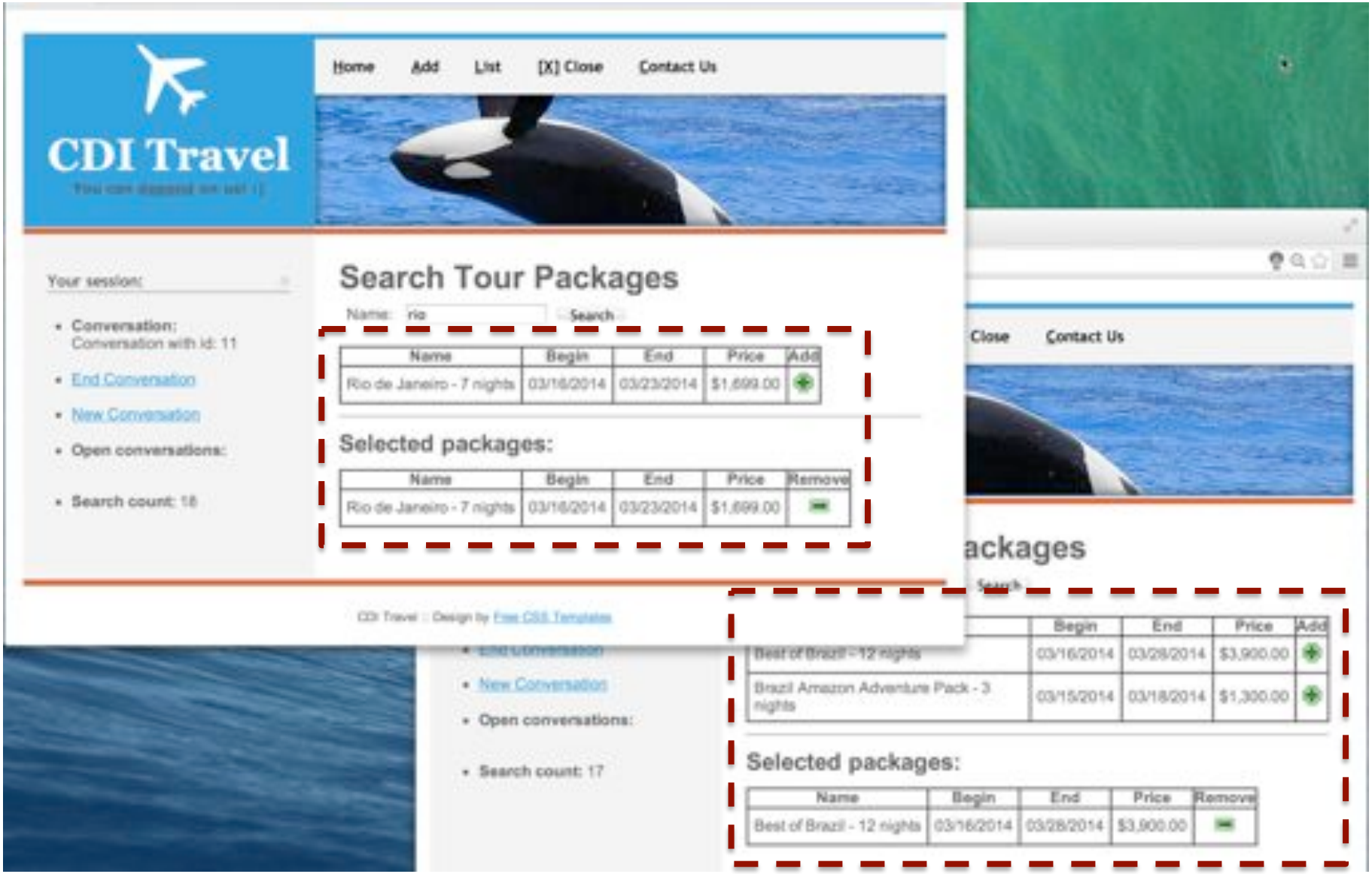
```
@SessionScoped
@Named
public class ListConversations implements Serializable {
   private List<String> conversationIds = new ArrayList<String>();

   public void add(Conversation conversation) {
      conversationIds.add(conversation.getId());
   }

   public void remove(Conversation conversation) {
      conversationIds.remove(conversation.getId());
   }

   public List<String> getList() {
      return conversationIds;
   }
}
```

Conversation handling

```
<ul>
  <li><strong>Conversation:</strong><br /><h:outputText
value="#{cvmg.conversation}" /></li>
  <h:panelGroup rendered="#{cvmg.conversation.transient}">
     <li><h:commandLink action="#{cvmg.beginConversation}"
value="Begin Conversation" /></li>
  </h:panelGroup>
  <h:panelGroup rendered="#{not cvmg.conversation.transient}">
     <li><h:commandLink action="#{cvmg.endConversation}" value="End
Conversation" /></li>
     <li><a href="index.faces">New Conversation</a></li>
  </h:panelGroup>
  <li><strong>Open conversations:</strong>
     <h:dataTable value="#{listConversations.list}" var="id">
        <h:column><a href="index.faces?cid=#{id}"><h:outputText
value="ID #{id}" /></a></h:column>
     </h:dataTable>
  </li>
  <!-- ... -->
</ul>
```

Note that when using links, ?cid= has to be used to keep in the same conversation. JSF components (e.g., buttons) keep the conversation.

Multiple conversations



Multiple conversations (decorator)

```
<h:form id="formMenu">
  <div id="menu">

     <!-- ... -->

     <li>
          <h:commandLink action="#{searchPackages.begin}"
accesskey="S" title="Search" target="_blank"
rendered="#{cvmg.conversation.transient}"><b>S</b>earch</
h:commandLink>
          <h:panelGroup rendered="#{not
cvmg.conversation.transient}"><a href="javascript:window.close();"
accesskey="X" title="[X] Close">[<b>X</b>] Close</a></h:panelGroup>
     </li>

     <!-- ... -->

  </div>
</h:form>
```

Opens a new window and begins a conversation.

Multiple conversations (bean)

```
@ConversationScoped @Named
public class SearchPackages implements Serializable {
   @Inject
   private Conversation conversation;

   // ...

   public String begin() {
      conversation.begin();
      return "/searchPackages.xhtml";
   }

   // ...
}
```

One could open multiple windows with different conversations.

Multiple conversations (web page)

```
<h1>Search Tour Packages</h1>

<h:form id="searchPackages">
  <h:panelGrid columns="3">
     <h:outputLabel for="name" value="Name:" />
     <h:inputText id="name" value="#{searchPackages.name}"
size="20" />
     <h:commandButton action="#{searchPackages.search}"
value="Search" />
  </h:panelGrid>

  <!-- ... -->
</h:form>
```

Multiple conversations (back to bean)

```
@ConversationScoped @Named
public class SearchPackages implements Serializable {
   @EJB private TourPackageDAO tourPackageDAO;
   private String name;                              // + getter/setter
   private List<TourPackage> searchResults;          // + getter

   // ...

   public String search() {
      searchEvent.fire(new SearchEvent(name));
      searchResults = tourPackageDAO.findByName(name);
      return null;
   }

   // ...
}
```

Search results are different in different conversations.

Multiple conversations (back to page)

```
<h:dataTable value="#{searchPackages.searchResults}" var="pack"
rendered="#{(searchPackages.searchResults != null) and (not
searchPackages.searchResults.isEmpty())}" border="1" cellspacing="0">
   <h:column>
      <f:facet name="header">Name</f:facet>
      <h:outputText value="#{pack.name}" />
   </h:column>
   <h:column>
      <f:facet name="header">Begin</f:facet>
      <h:outputText value="#{pack.begin}"><f:convertDateTime
pattern="MM/dd/yyyy" /></h:outputText>
   </h:column>

   <!-- ... -->

</h:dataTable>
```

And so on to select a package, deselect a package, etc.

Flash “scope”

- Not a CDI scope (there’s no @FlashScoped);
- Inspired on Ruby on Rails, included in JSF 2;
- Motivation:
  - Use of redirect after JSF processing;
  - Request data is lost due to redirect;
  - Promoting the bean to session scope is too much.

Flash “scope” (link and form)

```
<!-- In the decorator -->
<li><h:commandLink action="sendMessage.faces" accesskey="C"
title="Contact Us"><b>C</b>ontact Us</h:commandLink></li>
```

```
<h1>Contact Us</h1>                <!-- sendMessage.xhtml -->

<h:form id="sendMessage">
  <h:panelGrid columns="2">
     <h:outputLabel for="name" value="Name:" />
     <h:inputText id="name" value="#{sendMessage.name}" size="40" />

     <h:outputLabel for="email" value="E-mail:" />
     <h:inputText id="email" value="#{sendMessage.email}" size="30”/>

     <h:outputLabel for="message" value="Message:" />
     <h:inputTextarea id="message" value="#{sendMessage.message}"
cols="40" rows="4" />

     <h:commandButton action="#{sendMessage.send}" value="Send" />
  </h:panelGrid>
</h:form>
```

Flash “scope” (bean and result web page)

```
@Model
public class SendMessage {
   private String name;              // + getters/setters
   private String email;             // + getters/setters
   private String message;           // + getters/setters

   public String send() {
      /* Code that sends the message via e-mail (not implemented). */

      // Stores the bean in the FLASH context.
      Flash flash =
FacesContext.getCurrentInstance().getExternalContext().getFlash();
      flash.put("sendMessage", this);

      // Redirects to the thank you page.
      return "thanksForYourMessage.xhtml?faces-redirect=true";
   }
}
```

```
<h1>Contact</h1>

<p>Dear <h:outputText value="#{flash.sendMessage.name}" />, thank you
for your message!</p>
```

Alternatives

- Scenario: during testing I want to inject “mock” versions of beans, but when deploying use the real versions;
- Should we use producer methods?
  - Unnecessary processing for the deployed version;
- Should we use qualifiers?
  - Manual work of placing qualifiers, could lead to errors;
- Solution: alternatives.

Alternatives: how does it work?

- Program to interfaces, not classes (good practice):
  - The injected bean should have an interface;
  - The attribute that receives the injection should be defined as being of the interface type.
- All implementing classes but one receive the @Alternative annotation;
- The one without the annotation is the one used in real deployments;
- The alternatives can be “turned on” via a configuration in beans.xml.

Alternatives: example

```
public interface Message extends Serializable {
   String getMessage();
}
```

```
public class MessageBR implements Message {
   @Override
   public String getMessage() {
      return "A CDI Travel é uma agência com muita tradição...";
   }
}
```

```
@Alternative
public class MessageUS implements Message {
   @Override
   public String getMessage() {
      return "CDI Travel is a travel agency with years of...";
   }
}
```

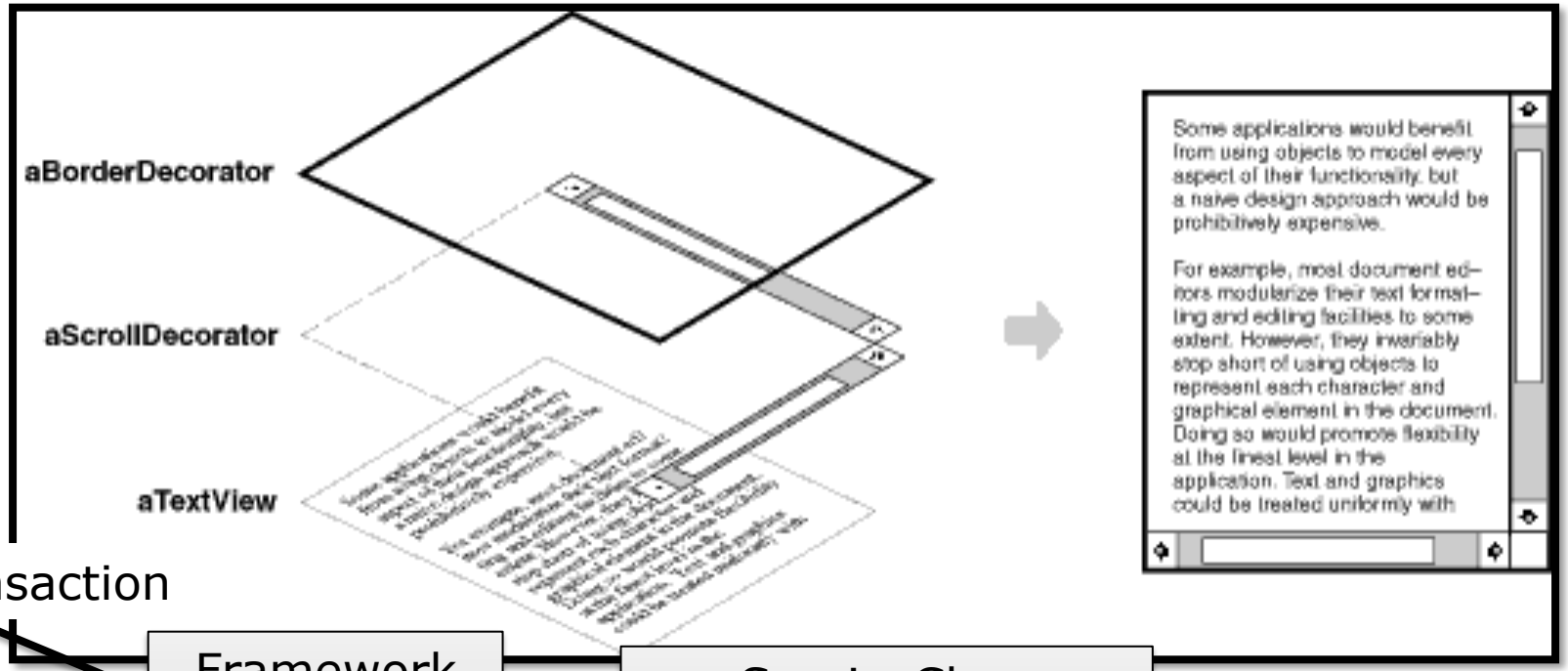
Alternatives: example

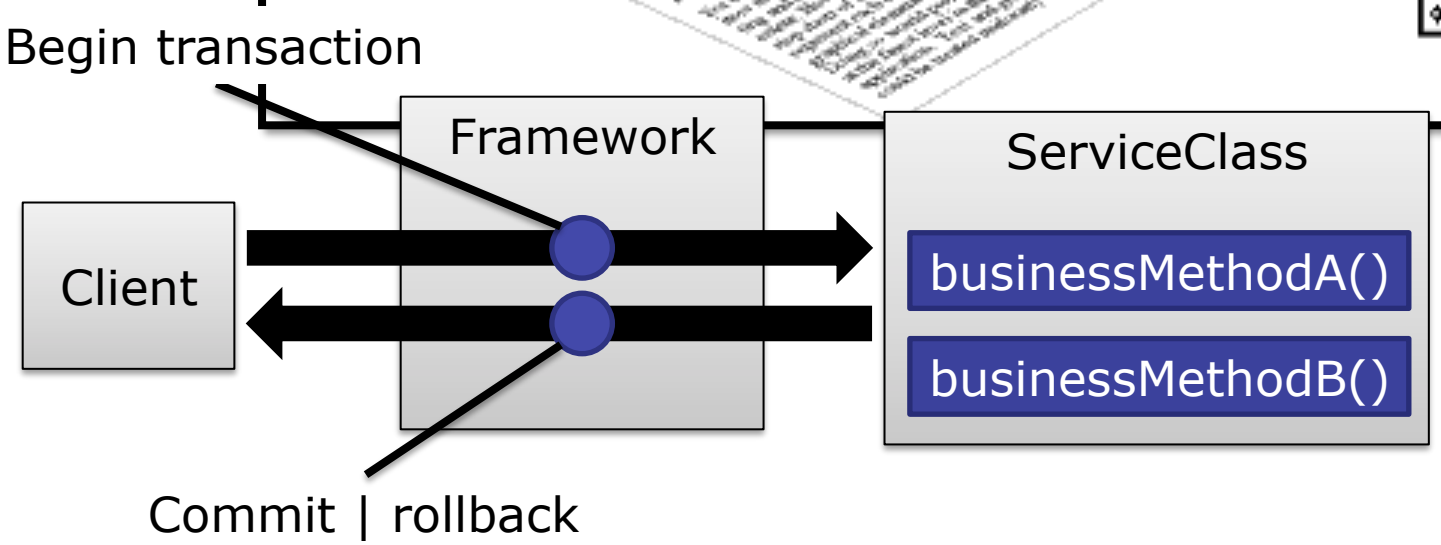
```
@ApplicationScoped @Named
public class ShowMessage implements Serializable {
   @Inject
   private Message message;

   public String getMessage() {
      return message.getMessage();
   }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://
java.sun.com/xml/ns/javaee/beans_1_0.xsd">

   <alternatives>
      <class>br.ufes.inf.nemo.dev.cditravel.beans.MessageUS</class>
   </alternatives>

</beans>
```

Interceptors and decorators

- Design patterns, commonly used in AOP.





Interceptors: how does it work?

- Create an annotation that will represent your interceptor;
- Develop one (or more) implementations of your interceptor;
- Annotate one of your beans (the whole class or a specific method) or an entire stereotype with the interceptor annotation;
- Activate one of the interceptor's implementations in beans.xml.

Interceptors: example

```
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Timed {}
```

```
@Timed @Interceptor
public class TimerInterceptor implements Serializable {
   private static final Logger logger =
Logger.getLogger(TimerInterceptor.class.getCanonicalName());

   @AroundInvoke
   public Object time(InvocationContext ctx) throws Exception {
      long start = System.currentTimeMillis();
      Object result = ctx.proceed();
      long time = System.currentTimeMillis() - start;
      logger.log(Level.INFO, "Time taken by method {0}: {1} ms", new
Object[] {ctx.getMethod().getName(), time});
      return result;
   }
}
```

Interceptors: example

```
@Stateless @Named("lp")
public class ListPackages {
   // ...

   @Timed
   public String getAveragePrice() {
      // ...
   }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

   <interceptors>
      <class>br.ufes....TimerInterceptor</class>
   </interceptors>
</beans>
```

Interceptors vs. decorators

- Interceptors look a lot like decorators (“around” invoke);
- In CDI, there are two main differences:
  - Decorators implement the same interface as the bean they are decorating;
  - Can have the decorated bean injected (in an attribute, constructor or initializer method).
- Also the process is simpler:
  - Create the decorator class;
  - Activate it in beans.xml.

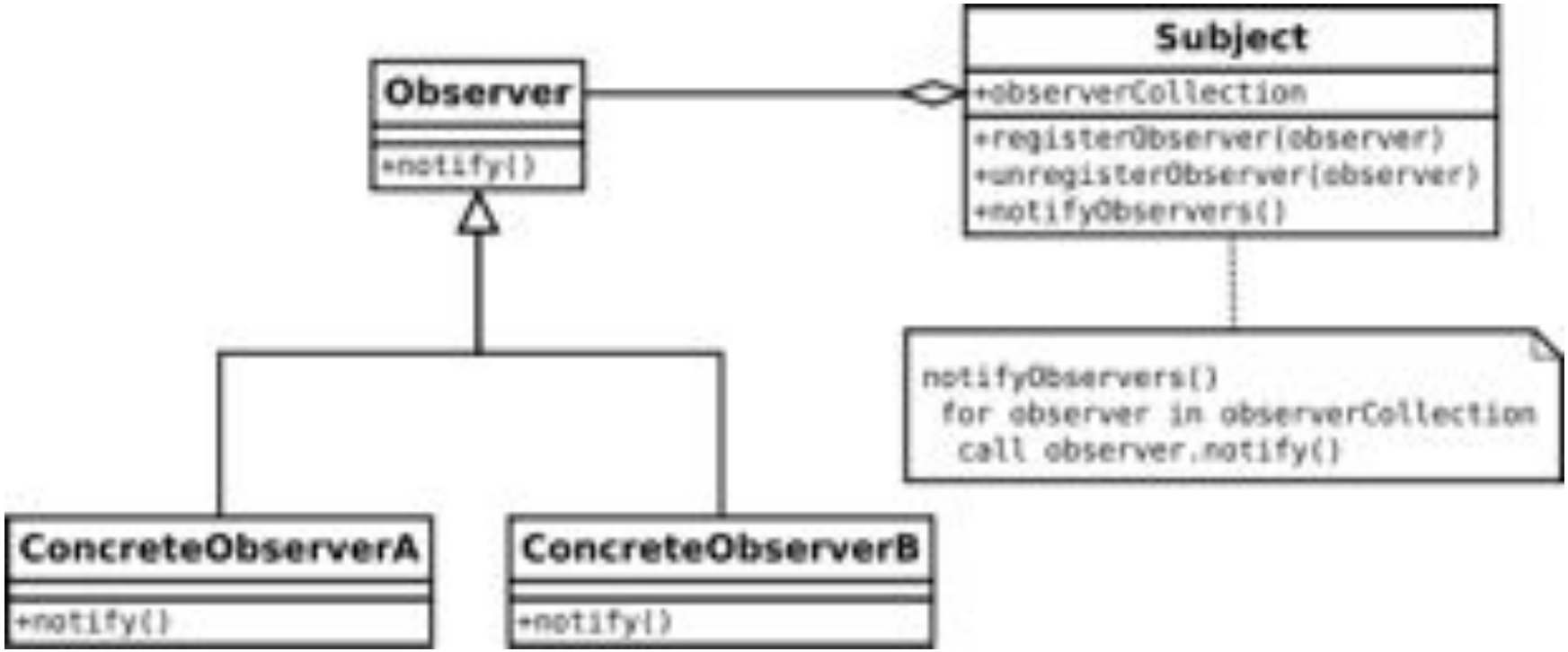
Decorators: example

```
@Decorator
public class QuotedMessage implements Message {
   @Inject @Delegate
   private Message message;

   @Override
   public String getMessage() {
      return "\"" + message.getMessage() + "\"";
   }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

   <decorators>
      <class>br.ufes....QuotedMessage</class>
   </decorators>
</beans>
```

Events

- Event-based development (e.g., like in AWT/Swing);
- Beans can be producers and consumers of events;
- CDI implements the observer design pattern.



Events: example

```
public class SearchEvent {
   private String name;

   public SearchEvent(String name) { this.name = name; }
   public String getName() { return name; }
   public void setName(String name) { this.name = name; }
}
```

```
@ConversationScoped @Named
public class SearchPackages implements Serializable {
   // ...

   @Inject
   private Event<SearchEvent> searchEvent;

   public String search() {
      searchEvent.fire(new SearchEvent(name));
      searchResults = tourPackageDAO.findByName(name);
      return null;
   }
}
```

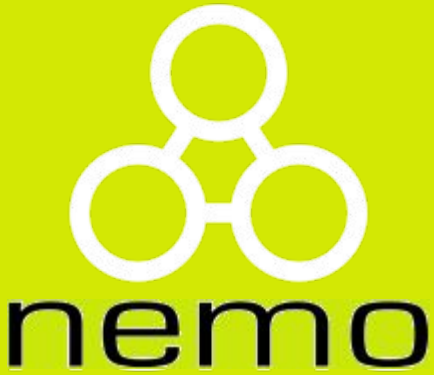
Events: example

```
@SessionScoped @Named
public class ShowNumberOfSearches implements Serializable {
  private static final Logger logger =
Logger.getLogger(ShowNumberOfSearches.class.getCanonicalName());

  private int qty = 0;

  public int getQty() {
     return qty;
  }

  public void countSearch(@Observes SearchEvent event) {
     logger.log(Level.INFO, "Counting one more search. Visitor
searched by name: {0}", event.getName());
     qty++;
  }
}
```

```
<!-- In the decorator -->
<strong>Search count:</strong>
<h:outputText value=" #{showNumberOfSearches.qty}" />
```

Next

- More on JPA: element collections, bean validation, JPQL, criteria API, etc.;
- More on EJBs: authorization, local/remote interfaces, singleton EJBs, asynchronous invocations, etc.



http://nemo.inf.ufes.br/