U N I V E R S I T Y   O F   T A R T U
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Chair of Theoretical Computer Science

Jevgeni Kabanov

Aranea—A Web Development and Integration Framework

Master thesis (40 AP)

Supervisor: Varmo Vene, PhD

TARTU 2007

Contents

Acknowledgements

First of all I'd like to thank Webmedia, Ltd. and specially Taavi Kotka and Ivo Mägi for believing in Aranea project success and thus making it possible.

Secondly, I'd like to thank my supervisor, Varmo Vene, who consistently gave me new challenges, without which this thesis would never come to be.

Of course a great deal of gratitude goes to my reviewers. Mark Fishel spent 8 hours one Sunday afternoon, helping me to polish and finish this thesis. Anton Litvinenko, Vesal Vojdani and Ragne Hanni found a lot of last moment mistakes.

Aranea was not a one person effort. Oleg Mürk contributed to the core ideas behind the framework. Maksim Boiko developed the initial prototype. Konstantin Tretjakov, Toomas Römer, Taimo Peelo, Alar Kvell, Rein Raudjärv and Andrei Ivanov have all contributed to the development of the open source implementation.

This work was partially supported by Estonian Science Foundation grant No. 6713.

Finally the greatest gratitude goes to the Flying Spaghetti Monster that kept me in the reach of his noodly appendages at all time, blessing my pasta and beer, allowing me to complete this work.

Introduction

During the last 10 years we have witnessed immense activity in the area of web framework design. Currently, there are more than 30 actively developed open source web frameworks in Java [33], not to mention commercial products, other platforms like .NET and dynamic languages and in-house corporate frameworks that never saw public light. Many different and incompatible design philosophies are used, but even within one approach there are multiple frameworks that have small implementation differences and are consequently incompatible with each other.

In this thesis we propose a minimalistic component model "Aranea" aimed at constructing and integrating server-side web controller frameworks in Java. It allows assembling most of available web programming models out of reusable components and patterns. We also show how to integrate different existing frameworks using Aranea as a common protocol. In its default configuration Aranea supports both developing sophisticated user interface using stateful components and nested processes, and high-performance stateless components.

We propose to use this model as a platform for framework development, integration, and research. This would allow using different ideas together and avoid reimplementing the same features repeatedly.

Motivation

One of the main problems with enterprise applications is their size. After some point in time the application becomes so large and so complex that a lot of work is required just to maintain it. Even more work is required to improve existing or add new functionality.

The best known way to combat this problem is to decompose the application in many modules or components. Most wide-spread way of doing it is Object-Oriented Programming [5].

Object-Oriented Programming employs object encapsulation and abstraction to reduce the dependencies between components, thus also reducing the effort required to understand and maintain the application.

However web applications have long suffered from lack of Object-Oriented approach. From all the web frameworks only one known to us provides good Object-Oriented abstractions—Wicket [38]. Even Wicket, however, does not provide first-class abstractions for pages and flows, which means that applications are still assembled using ad-hoc approach.

Supporting Object-Oriented Programming was a major goal of Aranea project. Everything is a first-class abstraction and large enterprise applications have been successfully built and deployed, employing Object-Oriented Development and Design to great benefit.

Another problem with enterprise applications is their age. Enterprises commonly accumulate a number of different applications developed for different business requirements over some time. They may use a number of different technologies, often outdated. These applications constitute the enterprise ecosystem, that software developers have to take into account.

However it is not usually economically feasible to just start a completely fresh application that fulfills all of those business requirements. It is more common to create an application fulfilling only a part of business requirements (mostly not covered by existing applications) and integrate it with the ecosystem.

Although there is a lot of solutions for integrating business logics, there is almost nothing for integrating web applications. It is often the case, that users have to use several separate web applications resorting to techniques like *copy-paste* to transfer information between them.

The second major Aranea goal is to enable integration on the web level. This is achieved by extending object encapsulation to allow implementing Aranea components in arbitrary web frameworks and technologies. This approach enables Aranea-based application to integrate enterprise ecosystem into a single coherent component-based application. It also enables to painlessly migrate from outdated technologies, by allowing to rewrite the applications component-by-component, instead of all-at-once.

Ecosystem

Before starting to describe Aranea we would like to give an overview of the web framework ecosystem at the time of writing. A number of different approaches has been implemented by various web frameworks, some of them only marginal and some highly popular.

The first important approach is using stateless or reentrant components for high performance and low memory footprint, available in such frameworks as Struts [19] and WebWork [37].

An alternative one is using hierarchical composition of stateful non-reentrant components with event-based programming model, available in such frameworks as JSF [30], ASP.NET [25], Seaside [6], Wicket [38], Tapestry [27], FSM [12]. This model is often used for developing rich user interfaces (UIs), but generally poses higher demands on server's CPU and memory.

The next abstraction useful especially for developing rich UIs is nested processes, often referred to as *modal* processes, present for instance in such web frameworks as WASH [22], Cocoon [24], Spring Web Flow [35], and RIFE [2]. They are often referred to by the name of implementation mechanism—*continuations*. The original idea comes from Scheme [11, 21].

All of these continuation frameworks provide one top-level call-stack—essentially flows are like function calls spanning multiple web requests. A significant innovation can be found in framework Seaside [6], where continuations are combined with component model and call stack can be present at any level of component hierarchy.

Yet another important model is using asynchronous requests and partial page updates, referred to as Ajax [23]. It allows decreasing server-side state representation demands and increases responsiveness of UI. At an extreme, it allows creating essentially fat client applications with sophisticated UI within browser.

Finally there are different forms of metaprogramming such as domain specific language for describing UI as a state machine in Spring Web Flow [35] and domain-driven design as implemented in Ruby on Rails [34] or RIFE/Crud [3]. This often requires the framework to allow dynamic composition and configuration of components at run-time.

Prerequisites

This thesis implies the knowledge of basic software development and web practices, technologies and approaches. It also implies some basic experience in web development in Java and familiarity with at least one MVC web framework. A good introduction can be found e.g. in [4].

Contributions and Outline

The first chapter is a tutorial introducing the reader to the application of OOP practices in Aranea. It also serves as a case study for the rest of the thesis. It is based on a tutorial [15] written for first-time Aranea users.

The second chapter introduces the reader to the components, abstractions and assembly of the Aranea framework. It is less concerned with the implementation details and more with the ideas behind the framework. It is based on [20].

The third chapter describes how to use the framework components to enable web application integration.

The fourth and final chapter describes Aranea extensions. The first section is based on [20]. The second section is based on [16].

The main contribution is the open source Aranea project, comprising over 100 thousand lines of code and over 200 pages of documentation with several examples and subprojects. The project was co-founded with Oleg Mürk and built under author's leadership by the Aranea team. The project distribution is available on the accompanying CD and from the project website http://www.araneaframework.org.

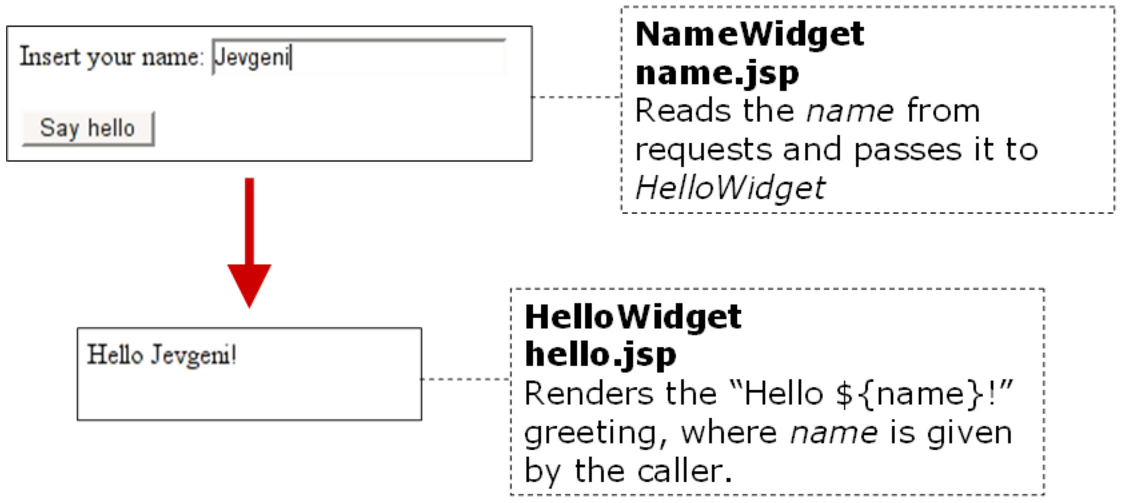
Chapter 1

Tutorial

This chapter demonstrates the way Aranea applications are assembled from independent widgets and how we can use these widgets to apply Object-Oriented idioms.

1.1 Hello World!

For the first introduction to Aranea, we present the example that has become classical in computer literature for learning new languages and technologies–“Hello World!” [17]. Our variant of “Hello World!” will display a form where one can insert his name and then display a personalized greeting. We will assume that all Java classes in the example are in the `example` package.



Insert your name: Jevgeni
Say hello
NameWidget
name.jsp
Reads the *name* from requests and passes it to *HelloWidget*
Hello Jevgeni!
HelloWidget
hello.jsp
Renders the “Hello ${name}!” greeting, where *name* is given by the caller.

Figure 1.1: Insert your name display

As we can see on Figure 1.1, we will split the example into two separate components (that we refer to hereon as widgets, see 2.1.3). Generally one component would be more than enough (for that matter a single JSP would be enough). The starting widget is `NameWidget` that has an associated `name.jsp` and displays the form for inserting your name. The source code of the widget follows:

```
public class NameWidget extends BaseUIWidget {
  protected void init() throws Exception {
    setViewSelector("name");
  }

  public void handleEventHello() throws Exception {
    String name =
      (String) getCurrentInput().getGlobalData().get("name");
    getFlowCtx().replace(new HelloWidget(name), null);
  }
}
```

The `init()` method is called when the widget is initialized by the framework and can be used to perform initialization tasks. In this case the widget selects the JSP that will be used for rendering by calling the `setViewSelector("name")`, which by default will be `/WEB-INF/jsp/name.jsp`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:ui="http://araneaframework.org/tag-library/standard"
  version="1.2">
  <ui:root>
    <ui:viewPort>
      <head>
        <ui:importScripts/>
        <title>Hello World!</title>
      </head>
      <ui:body>
        <ui:systemForm method="GET">
            Insert your name: <input type="text" name="name"/><br/><br/>
            <ui:eventButton labelId="#Say hello" eventId="hello"/>
        </ui:systemForm>
      </ui:body>
    </ui:viewPort>
  </ui:root>
</jsp:root>
```

`<ui:systemForm/>` renders the actual HTML form and `<ui:eventButton>` renders a button with a label “Say hello!” that will send the `hello` event to the `NameWidget`, which calls its `handleEventHello()` method.

As soon as that occurs the first line of `handleEventHello()` method

```
String name = (String) getCurrentInput().getGlobalData().get("name");
```

will read the data submitted by the form input field ”name” (this is not the way it is done usually in Aranea, but using forms would complicate this example). The second line of this method

```
getFlowCtx().replace(new HelloWidget(name), null);
```

will create a new instance of `HelloWidget` passing it the read `name` as a constructor argument. Then we replace the current widget in the flow with a new one, which causes the `NameWidget` instance to be destroyed and the `HelloWidget` instance to become active. The `HelloWidget` source follows:

```
public class HelloWidget extends BaseUIWidget {
  private String name;

  public HelloWidget(String name) {
    this.name = name;
  }

  public String getName() {
    return this.name;
  }

  protected void init() throws Exception {
    setViewSelector("hello");
  }
}
```

`HelloWidget` saves the constructor argument and makes it visible to the JSP via the getter `getName()`. It will then delegate the rendering to the following JSP:

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:ui="http://araneaframework.org/tag-library/standard"
  version="1.2">
  <ui:root>
    <ui:viewPort>
      <ui:widgetContext>
        <head>
          <ui:importScripts/>

          <title>Hello World!</title>
        </head>
        <ui:body>
          Hello <c:out value="${widget.name}"/>!
        </ui:body>
      </ui:widgetContext>
    </ui:viewPort>
  </ui:root>
</jsp:root>
```

Finally we will need to deploy the example as a web application. The simplest way to configure Aranea is to pass the application starting widget to its dispatcher servlet in the Servlet `web.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- Aranea session cleanup listener -->
  <listener>
    <listener-class>
org.araneaframework.http.core.StandardSessionListener
    </listener-class>
  </listener>

  <!-- Aranea dispatcher servlet -->
   <servlet>
      <servlet-name>araneaServlet</servlet-name>
      <servlet-class>
org.araneaframework.integration.spring.AraneaSpringDispatcherServlet
      </servlet-class>
      <init-param>
        <param-name>araneaApplicationStart</param-name>
        <param-value>example.NameWidget</param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <!-- Aranea dispatcher servlet mapping -->
   <servlet-mapping>
      <servlet-name>araneaServlet</servlet-name>
      <url-pattern>/main/*</url-pattern>
  </servlet-mapping>
</web-app>
```

1.2 Reusing Widgets

Having presented a simple example we can go on to examine more complex topics. Every widget can also be freely reused as a component inside any other widget. Let's try to reuse `HelloWidget` inside of `NameWidget` as shown on Figure 1.2.

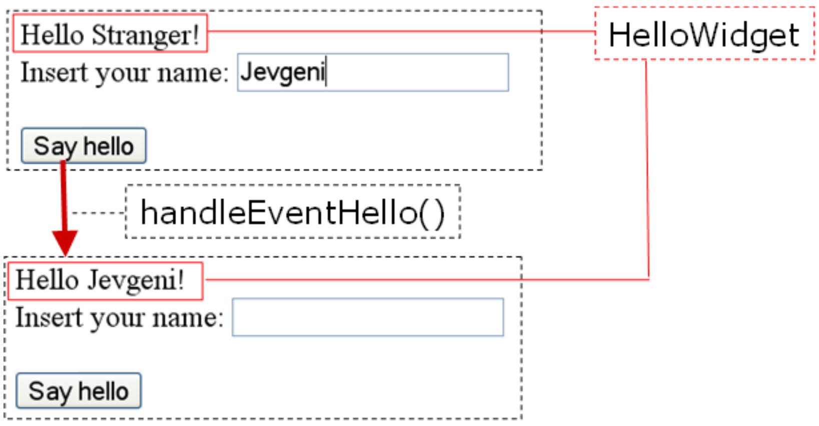


Figure 1.2: `HelloWidget` in `NameWidget`

First we have to modify the `HelloWidget`, adding it a setter in addition to the getter:

```
public class HelloWidget extends BaseUIWidget {
  private String name;

  public String getName() {
    return this.name;
  }

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

Then we modify the `NameWidget` to use `HelloWidget` directly instead of navigating to it:

```
public class NameWidget extends BaseUIWidget {
  private HelloWidget helloWidget;

  protected void init() throws Exception {
    helloWidget = new HelloWidget("Stranger");
    addWidget("hello", helloWidget);

    setViewSelector("name");
  }
  ...
}
```

This associates an instance of `HelloWidget` under name “hello” with the `NameWidget` and passes it the default greeting of “Stranger”. We save this instance in a field to be able to use it directly, which we do in `handleEventHello()`:

```
public class NameWidget extends BaseUIWidget {
  ...
 public void handleEventHello() throws Exception {
    String name = (String) getInputData().getGlobalData().get("name");
    helloWidget.setName(name);
  }
}
```

We use the setter to update the `HelloWidget` state and display the submitted name. The last thing we need to update is the `name.jsp`[1]:

```
...
<ui:systemForm method="GET">
  <ui:widgetInclude id="hello"/><br/>
  Insert your name:   <input type="text\ name="name"/><br/><br/>
  <ui:eventButton labelId="#Say hello" eventId="hello"/>
</ui:systemForm>
...
```

An interesting fact to notice is that `NameWidget` doesn't really have to depend on `HelloWidget`. In fact all we need is a widget that has a `setName(String name)` method as captured in the following interface:

```
public interface IHelloWidget extends Widget {
  public void setName(String name);
}
```

and make `NameWidget` accept it into its constructor:

[1]This is a simplification, since `hello.jsp` also includes `head`, `body` and other general tags. However in the next section we will show how to get rid of this repeating boilerplate and make leave only relevant tags in widget JSPs. This way we would also put the current example to work.

```
public class NameWidget extends BaseUIWidget {
  private IHelloWidget helloWidget;
  public NameWidget(IHelloWidget helloWidget) {
     this.helloWidget = helloWidget;
  }

  protected void init() throws Exception {
    addWidget("hello", helloWidget);
    setViewSelector("name");
  }
  ...
}
```

Then we could easily implement e.g. a widget that doesn't just display the greeting, but also tells how many hits does Google give for this name as shown on Figure 1.3. For

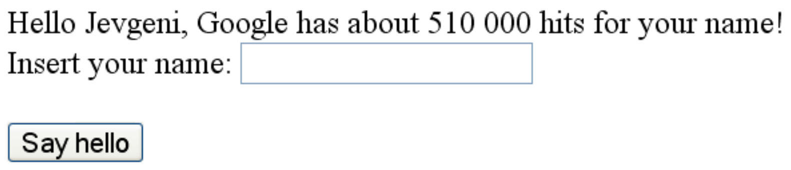

Hello Jevgeni, Google has about 510 000 hits for your name!
Insert your name:

Figure 1.3: `HelloWidget` with Google hits.

that we would just need a widget implementing `IHelloWidget` that searches Google on `setName()` calls:

```
public class GoogleHelloWidget implements IHelloWidget {
  private int hits;
  public void setName(String name) {
    //Connect to Google...
    this.hits = //Get number of hits...
  }
  //... select JSP and render
}
```

Another interesting fact is that since our widgets are objects through and through it is easy to make e.g. a widget that will use three independent `HelloWidgets` to display greetings to three different people:

```
public class ThreeGreetingsWidget extends BaseUIWidget {
  protected void init() throws Exception {
    addWidget("hello1", new HelloWidget("Jevgeni Kabanov"));
    addWidget("hello2", new HelloWidget("Taimo Peelo"));
    addWidget("hello3", new HelloWidget("Alar Kvell"));
    setViewSelector("threeGreetings");
  }
}
```

We would of course also need to include those widgets in the JSP:

```
...
<ui:widgetContext>
  <ui:widgetInclude id="hello1"/><br/>
  <ui:widgetInclude id="hello2"/><br/>
  <ui:widgetInclude id="hello3"/><br/>
</ui:widgetContext>
...
```

1.3 Flows

Flows can be introduced by modifying the same example. First of all we can notice that after the name is inserted, there is no going back—the widget on the flow has been exchanged and unless we create a new browser session we will always get the same "Hello Jevgeni!" message no matter how much we refresh. To remedy that let's modify the `HelloWidget` to have a back button as shown on figure 1.4.

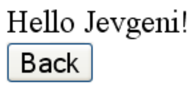


Figure 1.4: `HelloWidget` with a back button.

We start by adding the button to `hello.jsp`:

```
...
Hello <c:out value="${widget.name}"/>! <br/>
<ui:eventButton labelId="#Back" eventId="back"/>
...
```

This button maps to the `handleEventBack()` in the `HelloWidget`:

```
public class HelloWidget extends BaseUIWidget {
  ...
  public void handleEventBack() throws Exception {
    getFlowCtx().replace(new NameWidget(), null);
  }
}
```

Although this solution does the work, it has a number of drawbacks:

1. A new instance of `NameWidget` is constructed every time the back button is pressed. This means that `NameWidget` state is lost and the inserted name is not preserved.

2. `HelloWidget` doesn't really send us back, it just replaces itself with the `NameWidget`. Thus `HelloWidget` has to know where to send us, and if more than one widget would be able to create it we would return wrongly.

To solve this problem we have to update `NameWidget` and `HelloWidget` as follows:

```
public class NameWidget extends BaseUIWidget {
  ...
  public void handleEventHello() throws Exception {
    ...
    getFlowCtx().start(new HelloWidget(name), null);
  }
}

public class HelloWidget extends BaseUIWidget {
  ...
  public void handleEventBack() throws Exception {
    getFlowCtx().finish(null);
  }
}
```

Before the widgets replaced each other inside one and the same flow as shown on Figure 1.5.

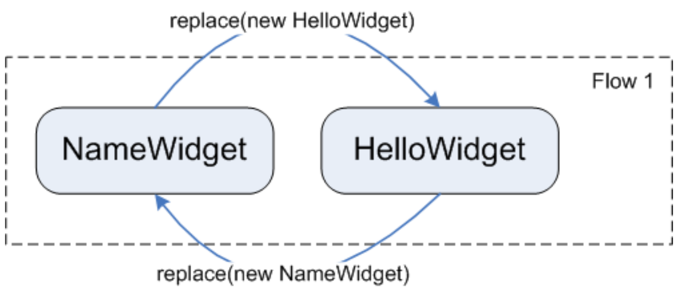


Figure 1.5: Widgets in same flow.

Now however widgets are in different flows as shown on Figure 1.6.

By calling `getFlowCtx().start()` we cause a new flow to be created containing a `HelloWidget` instance. The previous flow containing a `NameWidget` instance becomes inactive, and we can interact with the `HelloWidget` flow freely. After we call `getFlowCtx().finish()` the current flow finishes execution freeing the `HelloWidget` instance and the `NameWidget` flow becomes active again.

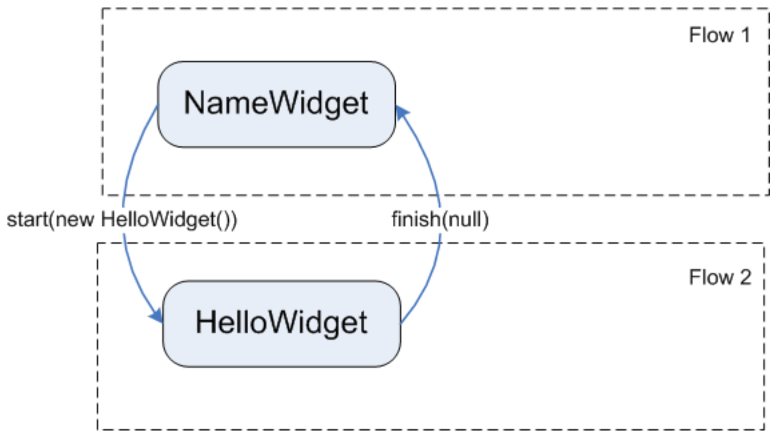


Figure 1.6: Widgets in different flows.

Now that the flows are covered, let's try to get the JSP files smaller. Currently we duplicate all of the root tags like head, body, etc in both widgets. In a real application this can be handled by introducing a `RootWidget`:

```
public class RootWidget extends BaseUIWidget {
  protected void init() throws Exception {
    setViewSelector("root");
    addWidget("flowContainer",
      new StandardFlowContainerWidget(new NameWidget()));
  }
}
```

`StandardFlowContainerWidget` is a widget that provides the `FlowContext` used for navigation in the previous examples. It handles a stack of child widgets and implements `start()` and `finish()` by pushing on and popping from the stack. And since it is just a usual widget we can associate it with our `RootWidget` as usual and pass it a `NameWidget` instance to be the starting flow.

Now we can put all of the boilerplate tags into `root.jsp`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:ui="http://araneaframework.org/tag-library/standard"
  version="1.2">
  <ui:root>
    <ui:viewPort>
      <ui:widgetContext>
        <head>
          <ui:importScripts/>
          <title>Hello World!</title>
        </head>

        <ui:body>
          <ui:widgetInclude id="flowContainer"/>
        </ui:body>
      </ui:widgetContext>
    </ui:viewPort>
  </ui:root>
</jsp:root>
```

Then the junk from other JSPs can also be removed. After that the `name.jsp` will look as follows:

```
<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:ui="http://araneaframework.org/tag-library/standard"
  version="1.2">

  <ui:widgetContext>
    Insert your name:   <input type="text\ name="name"/><br/><br/>
    <ui:eventButton labelId="#Say hello" eventId="hello"/>
  </ui:widgetContext>
</jsp:root>
```

and the `hello.jsp` as follows:

```
<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:ui="http://araneaframework.org/tag-library/standard"
  version="1.2">

  <ui:widgetContext>
    Hello <c:out value="${widget.name}"/>! <br/>
    <ui:eventButton labelId="#Back" eventId="back"/>
  </ui:widgetContext>
</jsp:root>
```

`<ui:widgetContext>` is the only required tag that declares that this JSP belongs to a widget.

The last thing to note, is that since `StandardFlowContainerWidget` is a usual widget, it is possible to have several of them running in parallel:

```
public class RootWidget extends BaseUIWidget {
  protected void init() throws Exception {
    setViewSelector("root");
    addWidget("flowContainer1",
      new StandardFlowContainerWidget(new NameWidget()));
    addWidget("flowContainer2",
      new StandardFlowContainerWidget(new NameWidget()));
    addWidget("flowContainer3",
      new StandardFlowContainerWidget(new NameWidget()));
  }
}
```

Of course we would also need to include them from the corresponding JSP, but this would allow three independent instances of "Hello World" application to run unhindered on a single page, with each application being completely independent of the rest [2] as shown on Figure 1.7.

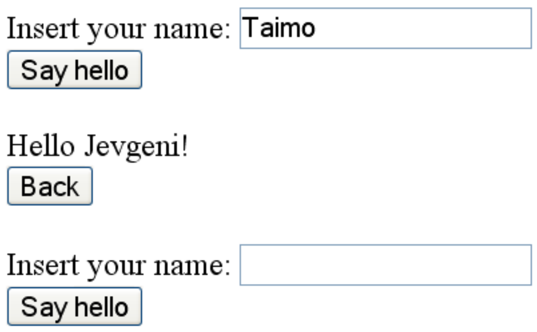


Figure 1.7: Three flow containers.

1.4 Putting It Together

Now that we have covered both widgets and flows let's try to approach this example as we would in a usual application, where each component can be a valuable reuse entity later. We'd like to reuse the widgets both as resuable components and flows.

However if we include `NameWidget` as a child widget it will start a new flow with HelloWidget in it, which might not be what we really want. We'd rather have `NameWidget` notify us somehow. To do that we need it to declare a callback:

[2]This is a simplification, as all three widgets have a textbox with one and the same name, which would conflict when submitted. In reality we would have to scope the name, adding a `${widgetId}` prefix to it and change `getGlobalData()` to `getScopedData()` in the parameter reading code. However since in real-life use cases forms scope parameters automatically, we choose to ignore this problem in our example as well.

```
public class NameWidget extends BaseUIWidget {
  interface Callback extends Serializable {
    void nameSelected(String name);
  }

  private Callback callback;

  public NameWidget(Callback callback) {
    this.callback = callback;
  }

  protected void init() throws Exception {
    setViewSelector("name");
  }

  public void handleEventHello() throws Exception {
    String name =
      (String) getCurrentInput().getGlobalData().get("name");
    callback.nameSelected("name");
  }
}
```

Now we can just include `NameWidget` as a child and it will let us know when the user clicks the button selecting the name. We can also wrap it as a flow easily:

```
public class NameFlowWidget
  extends BaseUIWidget implements NameWidget.Callback{

  protected void init() throws Exception {
    addWidget("name", new NameWidget(this));
    setViewSelector("nameFlow");
  }

  public void nameSelected(String name) {
    getFlowCtx().start(new HelloFlowWidget(name));
  }
}
```

where the `nameFlow.jsp` should just include the "name" widget.

We need `HelloWidget` to have a setter and its JSP should not have a back button (just like in 1.2). Then the `HelloFlowWidget` wrapper class will look as follows:

```
public class HelloFlowWidget extends BaseUIWidget{
  private HelloWidget helloWidget;

  public HelloFlowWidget(String name) {
    this.helloWidget = new HelloWidget(name);
  }

  protected void init() throws Exception {
    addWidget("hello", helloWidget);
    setViewSelector("helloFlow");
  }
}
```

Finally we add the back button in the `helloFlow.jsp`:

```
<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:ui="http://araneaframework.org/tag-library/standard" version="1.2">

  <ui:widgetContext>
    <ui:widgetInclude id="hello"/><br/>
    <ui:eventButton labelId="#Back" eventId="back"/>
  </ui:widgetContext>
</jsp:root>
```

Chapter 2

Web Development Framework

This chapter introduces components, abstractions and assembly of the Aranea framework.

2.1 Abstractions

Aranea framework is based on the abstraction of components arranged in a dynamic hierarchy and two component subtypes: services that model reentrant controllers and widgets that model non-reentrant stateful controllers. In this section we examine their interface and implementation ideas. We omit the checked exceptions and some other details from the interfaces for brevity.

2.1.1 Components

At the core of Aranea lies a simple notion of components arranged into a dynamic hierarchy that follows the *Composite* pattern [9] with certain mechanisms for communicating between children and parents. This abstraction is captured by the following interface:

```
interface Component {
  void init(Environment env);
  void enable();
  void disable();
  void propagate(Message msg);
  void destroy();

  Scope getScope();
  Environment getEnvironment()
}
```

A component is an entity that

- Has a life-cycle that begins with an `init()` call and ends with a `destroy()` call.
- Can be signaled to be disabled and then enabled again.
- Has an `Environment` that is passed to it by its parent or creator during initialization and can be retrieved using `getEnvironment()` method.
- Can propagate `Message`s to its children.
- Has a scope that signifies its position in the component hierarchy.

We imply that a component will have a parent and may have children. Aranea actually implies that the component would realize a certain flavor of the *Composite* pattern that requires each child to have a unique identifier in relation to its parent. These identifiers are combined to create a scope that allows tracing the component starting from the hierarchy root. The hierarchy is in no way static and can be modified at any time by any parent.

The hierarchy we have arranged from our components so far is inert. To allow some communication between different components we need to examine in detail the notions of `Environment` and `Message`.

`Environment` is captured by the following interface:

```
interface Environment {
  Object getEntry(Object key);
}
```

Environment is a discovery mechanism allowing children to discover services (named *contexts*) provided by their parents without actually knowing, which parent provides it. Looking up a context is done by calling the environment `getEntry()` method passing some well-known context name as the key. By a convention this well-known name is the interface class realized by the context. The following example illustrates how environment can be used:

```
L10nContext locCtx = (L10nContext)
  getEnvironment().getEntry(L10nContext.class);
String message = locCtx.localize("message.key");
```

Environment may contain entries added by any of the current component ancestors, however the current component direct parent has complete control over the exact entries that the current component can discover. It can add new entries, override old ones as well as remove (or rather filter out) entries it does not want the child component to access. This is done by wrapping the grandparent `Environment` into a proxy that will allow only specific entries to be looked up from the grandparent.

`Message` is captured in the following interface:

```
interface Message {
  void send(Object key, Component comp);
}
```

While the environment allows communicating with the component parents, messages allow communicating with the component descendants (indirect children). `Message` is basically an adaptation of the *Visitor* pattern [9] to our flavor of *Composite*. The idea is that a component method `propagate(Message m)` will just call the message method `m.send(...)` for each of its children passing the message both their instances and identifiers. The message can then propagate itself further or call any other component methods.

It is easy to see that messages allow constructing both broadcasting (just sending the message to all of the components under the current component) and routed messages that receive a relative "path" from the current component and route the message to the intended one. The following example illustrates a component broadcasting some message to all its descendants (`BroadcastMessage` will call `execute` for all component under current):

```
Message myEvent = new BroadcastMessage() {
  public void execute(Component comp) {
    if (comp instanceof MyDataListener)
      ((MyDataListener) comp).setMyData(data);
  }
}
myEvent.send(null, rootComponent);
```

2.1.2 Services

Although component hierarchy is a very powerful concept and messaging is enough to do most of the communication, it is comfortable to define a specialized component type that is closer to the *Controller* pattern [9]. We call this component `Service` and it is captured by the following interface:

```
interface Service extends Component {
  void action(
         Path path,
         InputData input,
         OutputData output
         );
}
```

`Service` is basically an abstraction of a reentrant controller in our hierarchy of components. The `InputData` and `OutputData` are simple generic abstractions over, correspondingly, a request and a response, which allow the controller to process request data and generate the response. The `Path` is an abstracted representation of the full path to the service from the root. It allows services to route the request to the one service it is intended for.

However since service is also a component it still can enrich the environment with additional contexts that can be used by its children.

2.1.3 Widgets

In the next section we will examine in more detail how we can use services to put a framework together. However although services are very powerful they are not too comfortable for programming stateful non-reentrant applications. To do that as well as to capture GUI abstractions we will introduce the notion of a `Widget`, which is captured in the following interface:

```
interface Widget extends Service {
  void update(InputData data);
  void event(Path path, InputData input);
  void render(OutputData output);
}
```

Widgets extend services, but unlike them widgets are usually stateful and are always assumed to be non-reentrant. The widget methods form a request-response cycle that should proceed in the following order:

1. `update()` is called on all the widgets in the hierarchy allowing them to read data intended for them from the request.
2. `event()` call is routed to a single widget in the hierarchy using the supplied `Path`. It allows widgets to react to specific user events.
3. `render()` calls are not guided by any conventions. If called, widget should render itself (though it may delegate the rendering to e.g. template). The `render()` method should be idempotent, as it can be called arbitrary number of times before the next `update()` call.

Widgets inherit an `action()` method from the services. It may be used to interact with a single widget, e.g. for the purposes of making an asynchronous request through Ajax [23]. `action()` calls are allowed to

So far we called our components stateful or non-stateful without discussing the *persistence* of this state. A typical framework would introduce predefined scopes of persistence, however in Aranea we have very natural scopes for all our components—their lifetime. In Aranea one can just use the component fields and assume that they will persist until the component is destroyed. If the session router is used then the root component under it will live as long as the user session. This means that in Aranea state management is non-intrusive and invisible to the programmer, as most components live as long as they are needed.

2.1.4 Flows

To support flows (nested processes) we construct a flow container widget that essentially hosts a stack of widgets (where only the top widget is active at any time) and enriches their environment with the following context:

```
interface FlowContext {
  void start(Widget flow, Handler handler);
  void replace(Widget flow);

  void finish(Object result);
  void cancel();
}
```

This context is available in standard flow implementation by calling `getFlowCtx()`. Its methods are used as follows:

- Flow `A` running in a flow container starts a child flow `B` by calling `start(new B(...), null)`. The data passed to the flow `B` constructor can be thought of as incoming parameters to the nested process. The flow `A` then becomes inactive and flow `B` gets initialized.
- When flow `B` is finished interacting with the user, it calls `finish(...)` passing the return value to the method. Alternatively flow `B` can call the `cancel()` method if the flow was terminated by user without completing its task and thus without a return value. In both cases flow `B` is destroyed and flow `A` is reactivated.
- Instead of finishing or canceling, flow `B` can also replace itself by flow `C` by calling `replace(new C(...))`. In such case flow `B` gets destroyed, flow `C` gets initialized and activated, while flow `A` continues to be inactive. When flow `C` will finish flow `A` will get reactivated.

`Handler` is used when the calling flow needs to somehow react to the called flow finishing or canceling:

```
interface Handler {
  void onFinish(Object returnValue);
  void onCancel();
}
```

It is possible to use continuations to realize synchronous (blocking) semantics of flow invocation, as shown in the section 4, in which case the `Handler` interface is be redundant.

2.1.5 Protecting Framework Abstractions

Several problems come up in framework design, when the objects that application programmers use and extend also have a specific framework contract:

- Application programmer can call a framework method in a way that will break the contract, e.g. in wrong order, which is hard to enforce.
- Application programmer may extend a framework object overriding the framework method and again breaking the contract (even harder to enforce).
- Framework programmer may inadvertently call a method that is application-specific, since framework and application methods share the same namespace.
- Since all methods are in the same namespace it may be hard to find the one you need. There are frameworks that have 50 to 100 methods in core interfaces, some of which have to be extended, others called.

Java allows to solve "breaking the contract by overriding framework method" problem by declaring this method `final`. However this also has its drawbacks, as sometimes we would want framework programmers to still be able to override or extend some of the framework logic. Java however does not provide any good means to restrict visibility based on namespace (unless the classes are in the same package).

The solution chosen for Aranea is to hide the framework interfaces in an inner class behind an additional method call:

```
interface Component {
  Component.Interface _getComponent();

  interface Interface {
    void init(Environment env);
    void destroy();
    void propagate(Message message);
    void enable();
    void disable();
  }
}
```

The idea is that although we can't enforce the contract onto application programmers we can ensure that programmer is fully aware when he is calling a system method:

```
widget._getComponent().init(childEnvironment);
widget._getWidget().update(input);
```

Note that this also breaks the methods into *namespaces* with one global namespace for public custom application methods and separate named namespaces for each of the framework interfaces. This allows to document and use them in a considerably clearer manner.

2.2 Assembly

Now that we are familiar with the core abstractions we can examine how the actual web framework is assembled. First of all it is comfortable to enumerate the component types that repeatedly occur in the framework:

Filter A component that contains one child and chooses depending on the request parameters whether to route calls to it.

Router A component that contains many children, but routes calls to only one of them depending on the request parameters.

Broadcaster A component that has many children and routes calls to all of them.

Adapter A component that translates calls from one protocol to another (e.g. from service to a widget or from Servlet [28] to a service).

Container A component that allows some type of children to function by enabling some particular protocol or functionality.

Of course of all of these component types also enrich the environment and send messages when needed.

Aranea framework is nothing, but a hierarchy (often looking like a chain) of components fulfilling independent tasks. There is no predefined way of assembling it. Instead we show how to assemble frameworks that can host a flat namespace of reentrant controllers (á la Struts [19] actions), a flat namespace of non-reentrant stateful controllers (á la JSF [30] components) and nested stateful flows (á la Spring Web Flow [35]). Finally we also consider how to merge all these approaches in one assembly.

2.2.1 Reentrant Controllers

The reentrant controller model is easily implemented by arranging the framework in a chain by containment (similar to pattern *Chain-of-Responsibility* [9]), which starting from the root would look as follows:

1. Servlet [28] adapter component that translates the servlet `doPost()` and `doGet()` to Aranea service `action()` calls.

2. HTTP filter service that sets the correct headers (including caching) and character encoding. Generally this step consists of a chain of multiple filters.

3. URL path router service that routes the request to one of the child services using the URL path after servlet. One path will be marked as default.

4. A number of custom application services, each registered under a specific URL to the URL path router service that correspond to the reentrant controllers. We call these services *actions*.

The idea is that the first component object actually contains the second as a field, the second actually contains the third and so on. Routers keep their children in a `Map`. When `action()` calls arrive each component propagates them down the chain.

The execution model of this framework will look as follows:

- The request coming to the root URL will be routed to the default service.
- When custom services are invoked they can render the HTML response (optionally delegating it to a template language) and insert into it URL paths of other custom services, allowing to route next request to them.
- A custom service may also issue an HTTP redirect directly sending the user to another custom service. This is useful when the former service performs some action that should not be repeated (e.g. money transfer).

Note that in this assembly `Path` is not used at all and actions are routed by the request URL.

Of course in a real setup we might need a number of additional filter services that would provide features like file uploading, but this is enough to emulate the model itself. Further on we will also omit the optional components from the assembled framework for brevity.

In general, steps 3-4 could be extended to be composed out of:

- Filter services that enrich `InputData` and `OutputData` based on some criteria and then delegate work to the single child service.
- Router services that route request to one of their children based on remaining part of URL, accessible from InputData.

Both filter and router services are stateful and reentrant. Router services could either create a new stateless action for each request (like WebWork [37] does) or route request to existing reentrant actions (like Struts [19] does). Router services could allow adding and removing (or enabling and disabling) child actions at runtime, although care must be taken to avoid destroying action that can be active on another thread.

We have shown above how analogues of Struts and WebWork actions fit into this architecture. WebWork interceptors could be implemented as a chain of filter services that decide based on `InputData` and `OutputData` whether to enrich them and then delegate work to the child service. There could be filter services both before action router and after. The former would be shared between all actions while the latter would be private for each action instance. A disadvantage of such approach is that each request must pass through all shared filters, although which filters are needed for particular action might be possible to decide statically before the request arrives.

If this turns out to be a problem, we could introduce a new concept of interceptor:

```
interface Interceptor extends Component {
  void intercept(
         Service service,
         InputData,
         OutputData
         );
}
```

Interceptor is a stateful reentrant component that does modifications to `InputData` and `OutputData` and then calls `action()` method of the service. When creating a new action, it could be wrapped into interceptors:

```
Interceptor i1 = ...
Interceptor i2 = ...
..
Service action = ...
Service p1 = new InterceptingProxy(i1, action);
Service p2 = new InterceptingProxy(i2, p1);
...
this.addService(pN);
```

Here `InterceptingProxy(Interceptor, Service)` proxies all method invocations to the service except for the one method:

```
void action(InputData in, OutputData out) {
  iterceptor.intercept(service, in, out)
}
```

There is no need to create more than one instance of each interceptor kind because they can be shared between wrapped actions. Interceptors allow mimicking WebWork interceptors more directly and are more space and time efficient as compared to chains of filters performing the same role.

2.2.2 Stateful Non-Reentrant Controllers

To emulate the stateful non-reentrant controllers we will need to host widgets in the user session. To do that we assemble the framework as follows:

1. Servlet [28] adapter component.
2. Session router that creates a new service for each new session and passes the `action()` call to the associated service.
3. Synchronizing filter service that let's only one request proceed at a time.
4. HTTP filter service.

5. Widget adapter service that translates a service `action()` call into a widget `update()/event()/render()` request-response cycle.

6. Widget container widget that will read from request the path to the widget that the event should be routed to and call `event()` with the correct path.

7. Page container widget that will allow the current child widget to replace itself with a new one.

8. Application root widget which in many cases is the login widget.

This setup is illustrated on Figure 2.1.

A real custom application would most probably have login widget as the application root. After authenticating the user, login widget would replace itself with the actual root widget, which in most cases would be the application menu (which would also contain another page container widget as its child).

The menu would contain a mapping of menu items to widget classes (or more generally factories) and would start the appropriate widget in the child page container when the user clicks a menu item. The custom application widgets would be able to navigate among each other using the page context added by the page container to their environment.

The execution model of this framework will look as follows:

- The request coming to the root URL will be routed to the application root widget. If this is a new user session, a new session service will be created by the session router.

- Only one request will be processes at once (due to synchronizing filter). This means that widget developers should never worry about concurrency.

- The widget may render the response, however it has no way of directly referencing other widgets by URLs. Therefore it must send all events from HTML to itself.

- Upon receiving an event the widget might replace itself with another widget (optionally passing it data through the constructor) using the context provided by the page container widget. Generally all modification of of widget hierarchy (e.g. adding/removing children) can be done during event part of the request-response cycle only.

- The hierarchy of widgets under the application root widget (e.g. GUI elements like forms or tabs) may be arranged using usual *Composite* widget implementations as no special routing is needed anymore.

In the real setup page container widget may be emulated using flow container widget that allows replacing the current flow with a new one.

Such an execution model is very similar to that of Wicket [38], JSF [30], or Tapestry [27] although these frameworks separate the pages from the rest of components (by declaring a special subclass) and add special support for markup components that compose the actual presentation of the page.

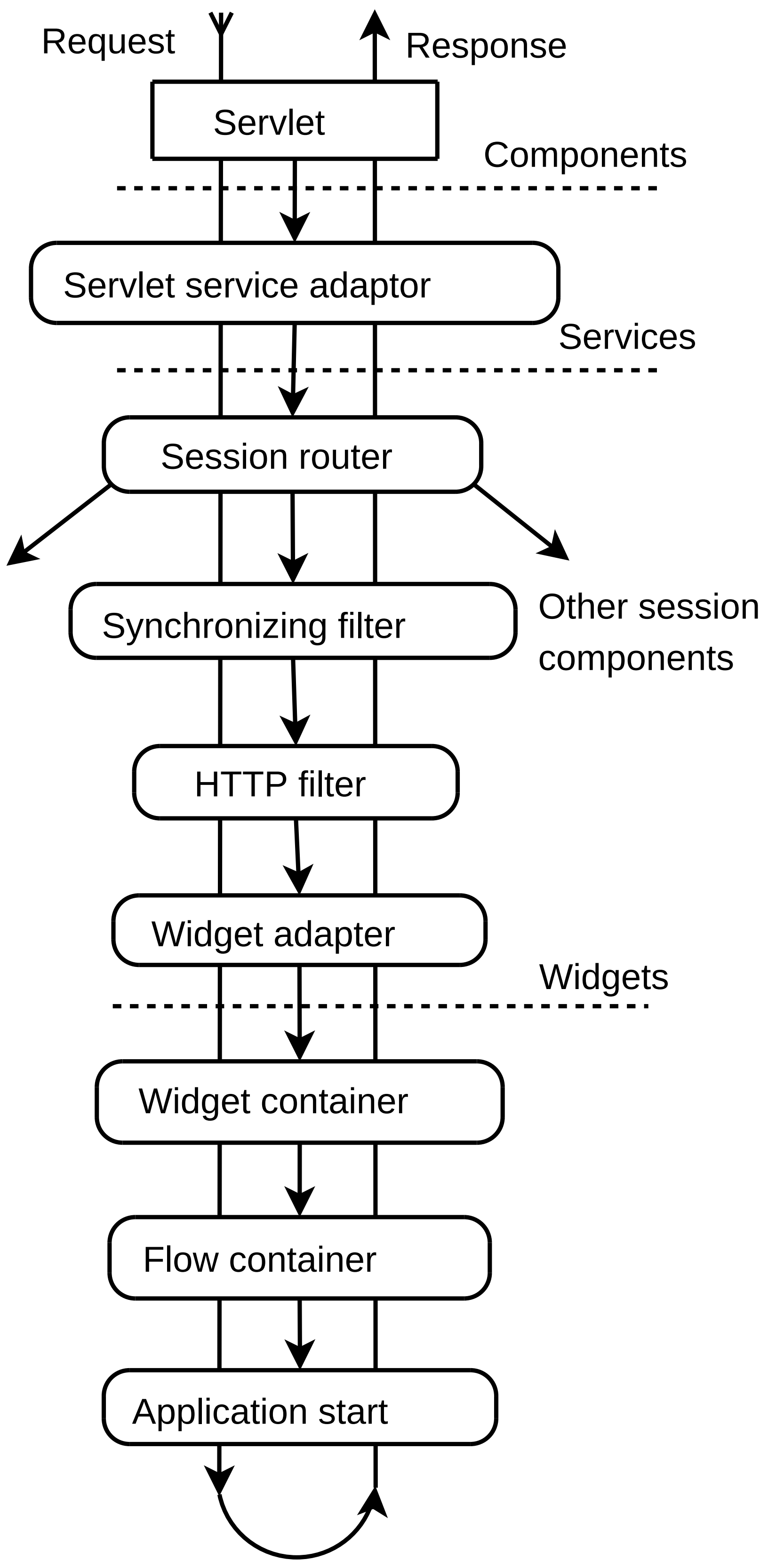


Figure 2.1: Framework assembly for hosting pages

2.2.3 Stateful Non-Reentrant Controllers with Flows

To add nested processes we basically need only to replace the page container with a flow container in the previous model:

1. Servlet [28] adapter component.
2. Session router service.
3. Synchronizing filter service.
4. HTTP filter service.
5. Widget adapter service.
6. Widget container widget.
7. Flow container widget that will allow to run nested processes.
8. Application root flow widget which in many cases is the login flow.

The execution model here is very similar to the one outlined in Subsection 2.2.2. The only difference is that the application root flow may start a new subflow instead of replacing itself with another widget.

This model is similar to that of Spring WebFlow [35], although Spring WebFlow uses Push-Down Finite State Automaton to simulate the same navigation pattern and consequently it has only one top-level call stack. In our model call stacks can appear at any level of widget composition hierarchy, which makes our model considerably more flexible.

2.2.4 Combining the Models

It is also relatively easy to combine these models, modifying the model shown on figure 2.1 by putting a URL path router service before the session router, map the session router to a particular URL path and put a flow container in the end.

The combined model is useful, since reentrant stateless services allow to download files from database and send other semi-static data comfortably to the user. They can also be used to serve parts of the application that has the highest demand and thus load.

It is also worth noting that such a model allows cooperation between the flows and reentrant services—e.g. widgets can dynamically add/remove them on need.

2.3 Aspects

Next we examine some typical web framework aspects and how they are realized in Aranea.

2.3.1 Configuration

The first aspect that we want to examine is *configuration.* We have repeated throughout the thesis that the components should form a dynamic hierarchy, however it is comfortable to use a static configuration to wire parts of that hierarchy that form the framework core.

To do that one can use just plain Java combining a hierarchy of objects using setter methods and constructors. But in reality it is more comfortable to use some configuration mechanism, like an IoC container [8]. We use in our configuration examples Spring [14] IoC container and wire the components together as beans. Note that even such static configuration contains elements of dynamicity, since some components (á la root user session service) are wired not as instances, but via a factory that returns a new service for each session.

```
SessionRouterService srs =
  new SessionRouterService();
srs.setSessionServiceFactory(
  new ServiceFactory() {
     Service buildService(Environment env) {
       Service result = ...
       //Build a new session service...
       return result;
     }
  }
);
```

2.3.2 Security

The most common aspect of security that frameworks have to deal with is *authorization.* A common task is to determine, whether or not the current user has enough privileges to see a given page, component or GUI element. In many frameworks the pages or components are mapped to a particular URL, which can also be accessed directly by sending an HTTP request. In such cases it is also important to restrict the URLs accessible by the user to only those he is authorized to see.

When programming in Aranea using stateless re-entrant services they might also be mapped to particular URLs that need to be protected. But when programming in Aranea using widgets and flows (a stateful programming model) there is no *general* way to start flows by sending HTTP requests. Thus the only things that need protection are usually the menu (which can be assigned privileges per every menu item) and the active flow and widgets (which can only receive the events they subscribe to).

This simplifies the authorization model to checking whether you have enough privileges to start the flow *before* starting it. Since most use-cases should have enough privileges to start all their subflows it is usually enough to assign coarse-grained privileges to use-cases that can be started from the menu as well as fine-grained privileges for some particular actions (like editing instead of viewing).

2.3.3 Error Handling

When an exception occurs the framework must give the user (or the programmer) an informative message and also provide some recovery possibilities. Aranea peculiarity is that since an exception can occur at any level of hierarchy the informing and recovery may be specific to this place in the hierarchy. Default behavior for Aranea components is just to propagate the error up the hierarchy to the first exception handler component

For example it might be required to be able to cancel a flow that has thrown an exception and return back to the flow that invoked the faulty flow. A logical solution is to let the flow container (and other similar components) to handle their children's exceptions by rendering an informative error subpage instead of the flow. The error page can then allow canceling flows by sending events to the flow container.

With such approach when we have several flow containers on one HTML page, then if two or more flows under different containers fail, they will independently show error subpages allowing to cancel the particular faulty flows. Note also that such approach will leave the usual navigation elements like menus intact, which will allow the user to navigate the application as usual.

Alternatively we may want to render the error page outside the flow, hiding the usual navigation element. To do that the flow container needs to re-throw the exception further upwards to the top-level exception handler, accompanied by a service that will be used to render the error page. This service will be given the flow container environment, thus allowing it to cancel flows.

With such approach only one flow can generate exception at one time, since it will escape the exception to the top-level exception handler. Both approaches have their merits and Aranea allows the particular flow container to choose the suitable strategy. Additionally Aranea provides a critical error handler that will render the exception stack for an error occurring high in the framework part of the hierarchy.

It should also be noted that to handle exceptions occurring after some data has been written to the response stream (e.g. during a `render()` call) we need to roll back this data altogether and render an informative error page instead. This is easily accomplished by wrapping the response stream with a buffer.

Certainly these approaches don't cover all possible use cases and custom exception handlers may be needed for new type of containers. However the approach is general enough to be applied similarly in new use cases.

2.3.4 Concurrency

Execution model of Aranea is such that each web request is processed on one Java thread, which makes system considerably easier to debug. By default Aranea does not synchronize component calls. It does, however, protect from trying to destroy a working component. If a service or widget currently in the middle of some method call will be destroyed, the destroyer will wait until it returns from the call. To protect from deadlock and livelock, after some time the lock will be released with a warning.

When we want to synchronize the actual calls (as we need for example with widgets) we can use the synchronizing service that allows only one `action()` call to take place simultaneously. This service can be used when configuring the Aranea framework to synchronize calls on e.g. browser window threads. This will allow to program assuming that only one request per browser window is processed at any moment of time. Note that widgets should *aways* be behind a synchronizing filter and cannot process concurrent calls.

2.4 Implementation

The previously described components are more-or-less straightforward to implement, however to actually develop applications one needs considerably more functionality than just a controller framework. In this section we present the components and extensions that make Aranea a full-fledged web framework usable for productive development of large applications.

2.4.1 Standard Components

Aranea includes standard implementations of the core abstractions: component, service, widget, environment and message.

The standard component, service and widget implementations (named `StandardComponent`, `StandardService` and `StandardWidget`) are similar to each other and mainly provide children management and synchronization of destruction. The children are managed using following methods (with "Component" substituted for accordingly "Service" or "Widget"):

- `addComponent(key, Component)` and `removeComponent(key)` add and remove the child to/from the parent as well as initializing/destroying it.
- `enableComponent(key)` and `disableComponent(key)` allow to enable/disable child blocking it from receiving calls and notifying it via `enable()`/`disable()` calls.
- `getChildComponentEnvironment()` that can be overridden to supply the child additional entries to its environment.

The standard component classes also implement call routing according to the *Composite* pattern described in Section 2.1.

In addition to this, standard service and standard widget implement event listener management that enable further discrimination between action/event calls to the same service/widget. This allows for truly event-driven programming.

There are two standard implementations of message: `RoutedMessage` and `BroadcastMessage`. The first one allows to send a message to a component with a known full path, while the second one broadcasts the message to all components under current.

2.4.2 Reusable Widget Library

While standard widget and service implementations supply the base classes for custom and reusable application coarse-grained controllers, the reusable widget library implements the fine-grained GUI abstractions like form elements, tabs, wizards and lists.

One of the most common tasks in Web is data binding/reading the values from the request, validation and conversion. Aranea *Forms* provide a GUI abstraction of the HTML controls and allows to bind the data read from the request to user beans. Forms support hierarchical data structures, change detection and custom validation and conversion.

Another common programming task is to display user a list (also called grid) that can be broken in pages, sorted and filtered. Aranea *lists* support both in-memory and database back-ends, allowing to generate database queries that return the exact data that is displayed. This allows to make lists taking memory for the currently displayed items only, which support tables with many thousands of records.

2.4.3 Presentation Framework

Finally Aranea also contains a JavaServer Pages [31] custom tag library that not only allows to access services and widgets, but also facilitates expressing user interface with less redundancy than W3C XHTML [36] and W3C CSS [26].

The core idea is to break the application UI into logical parts and capture them using reusable custom tags. Then one can program using a higher-level model than XHTML, operating with UI logical entities. The framework contains standard implementations for the reusable widget library tags and base implementations for the custom application tags together with specific examples.

```
<html>
    <body>
      <ui:systemForm method="POST">
        <h1>Aranea Template Application</h1>

        This renders the child widget
        with id "myChild":<br/>
        <ui:widgetInclude id="myChild"/>
      </ui:systemForm>
  </body>
</html>
```

Since Aranea controller in no way enforces particular rendering mechanism, every other component may be rendered by a different view framework, so this particular JSP-based rendering engine is in now way obligatory.

Chapter 3

Web Integration Framework

This chapter describes how to use the framework components to enable web application integration.

3.1 Requirements

One of the main design decisions in Aranea was to enforce Object-Oriented principles like encapsulation. This means that Aranea components are first-class objects that can be used abstractly, without any care for their implementation. This immediately gives raise to a question—is this abstraction powerful enough that we could implement Aranea components using third-party frameworks?

Application integration has two main aspects[1]:

Encapsulation Ensures that application behavior does not depend on any other applications.

Communication Allows different applications to interact with each other.

It is obvious that these aspects are conflicting with each other, since the need to communicate breaks encapsulation. To solve this problem we introduce a level of indirection—encapsulation will be provided by specific Aranea components that are aware of the application implementation. All communication between different applications will only be done using Aranea API. This way we provide full encapsulation from the Aranea point of view as well as enable arbitrary communication when needed.

Let's examine how our integrated application will be structured and what requirements encapsulation must satisfy.

First of all it is a logical step to embed encapsulation into component base classes. Although both services and widgets can provide integration, we will examine widgets more, since they are used more often and have more issues. For example Struts encapsulation may have a `StrutsWidget` base class and JSF encapsulation a `JsfWidget`. These widgets

[1]We could also identify a third one—*coherence*—that ensures all applications are using the same basic set of services, but this is less about application integration and more about framework integration.

should encapsulate a particular application, or rather an application module, by taking a starting URI into their constructor. For example `new StrutsWidget("/Logon.do")`.

To encapsulate a particular module we may create a widget that is tied to a particular starting URI and takes any necessary extra parameters as constructor arguments. This way we can use this widget as we do usually, both including it as a reusable component and starting it in a flow. We can also include widgets from the encapsulated module, including widgets implemented in yet another third-party framework. In fact we should provide access to Aranea API also to the native integrated application modules, so that they could make use of Aranea components and flows.

From this we can deduce several requirements:

- Several widgets can appear on one HTML page. Even more important those widgets could be instances of one and the same class, thus implemented by one and the same integrated module. In such a case encapsulation becomes all the more important, since the integrated modules will definitely have overlapping namespace, state and so on.

- All of the requests made from the integrated module must go through the Aranea component hierarchy to the encapsulating component that originated the request and only then the module should be included and rendered. Otherwise, if we let the request go to the originating application, the response will display only the integrated application, breaking the encapsulation.

- Since all inter-module communication takes place via Aranea components, we must implement application basic features using Aranea. This includes such features as authentication (usually implemented by a login screen) and module starting points (menus and similar).

- To integrate the applications (or modules) they need to undergo some amount of conversion. At the very least they will depend on Aranea API for communication and use Aranea authentication services. This means that they will not be able to run standalone anymore.

3.2 Implementation

Our approach allows to integrate several applications written using different web frameworks inside a single web container[2] application. The applications are inside the same JVM and they share access to Java Servlet API. Thus we can simulate parts of this API to enable independence of integrated applications and their modules.

[2]Since we use the term *web applications* generically we will use the term *web container applications* to refer to the Java web application deployment unit that contains a single `web.xml` mapping.

3.2.1 Request and State

First of all let's examine how servlets can interact with the user and among themselves:

Request URL. The basic URL that request was submitted to. Different servlets are mapped to different URLs and the specific path after the servlet URL is used for internal application communication. Since we want to fully encapsulate the applications in Aranea components we need to ensure that all requests will go through the Aranea component hierarchy to the correct component before being processed.

Request parameters. Carry the information submitted by a user to the server. Since different applications share the same namespace of parameter names we must make sure that applications get access only to parameters submitted by themselves.

Request attributes. Allow communication between different components of a web application inside the same request. Since these are also shared by all applications involved in the request we must make sure that they do not influence each other.

Session attributes. Allow storing user session-specific data between requests. Also shared by all applications, so we need to ensure that they can only access their own attributes.

Servlet attributes. Allow communication between servlets in the same web container application. Not as important as the rest, since web applications must anyway assume this is shared and provide their own namespace.

It seems that most of the important interactions is request-based[3]. HTTP Request is represented in Java by the interface `HttpServletRequest`, which includes methods related to:

- Request URL, like `getRequestURL()`, `getServletPath()`.
- Request parameters, like `getParameter()`, `getParameterMap()`.
- Request attributes, like `setAttribute()`, `getAttribute()`.
- Getting associated session: `getSession()`.

Since `HttpServletRequest` is an interface, it can be wrapped using the *Decorator* pattern [9]. Servlet API even provides a default wrapper implementation—`HttpServletRequestWrapper`. Using this pattern we override the default behavior of the methods by making them *local* as follows:

[3]Sessions in Servlet API are accessible only via the request

- We make the wrapper take the request URI as the constructor parameter. It should return it and its derivatives from all associated methods. We can safely assume that the URL part besides the URI (that is host, port, etc) is the same for all applications.
- We assume that all of the request parameters have been prefixed with some namespace. We take this namespace as a constructor parameter and make the application see only the request parameters inside this namespace.
- We create a local request attribute map. All attribute modification methods act only on this map, while all attribute query methods act first on the local map then globally. This ensures that although application can see the global attributes (e.g. set by Aranea) it cannot influence other applications.

Additionally, since `HttpSession` is an interface and can only be accessed from the `HttpServletRequest`, we can also wrap it and provide a local attribute map acting same way as the request attributes. A logical place to put the attribute map is the Aranea component that encapsulates the module, since its scope should exactly correspond to the user session with the integrated application.

The resulting wrapper will look something like this:

```
class RequestWrapper extends HttpServletRequestWrapper {
 private Map localRequestAttributes = new HashMap();
 private String prefix;
 private String requestURI;
 private SessionWrapper session;

 public RequestWrapper(HttpServletRequest request,
   String requestURI, String prefix, Map sessionAttributeMap) {

   super(request);

   this.prefix = prefix;
   this.requestURI = requestURI;
   this.session = new SessionWrapper(sessionAttributeMap);
 }

 public HttpSession getSession() {
   return session;
 }

 public String getParameter(String name) {
   if (name.startsWith(prefix))
     return super.getParameter(name);
```

```
    return null;
  }

  public void setAttribute(String name, Object o) {
    localRequestAttributes.put(name, o);
  }

   public Object getAttribute(String name) {
    if (localRequestAttributes.containsKey(name))
      return localRequestAttributes.get(name);
    return super.getAttribute(name);
  }

  public String getRequestURI() {
    return requestURI;
  }

  //...
}
```

This wrapper encapsulates all of the basic servlet-to-servlet communication, and we can proceed with implementing the rest of application integration.

3.2.2 Navigation

In most of the web frameworks request URL is used for navigation. In all of the web frameworks that we are aware of request URL is relied upon by the framework for some of it functionality.

Thus we must preserve the original URL when passing the request to the integrated module. At the same time we must make sure that Aranea handles the request first and includes the integrated module in its proper component. There are two problems with these requirements:

1. Most of the web frameworks will have their servlet URLs hard-coded into their HTML output, thus producing requests to their own servlets.

2. Even if they would produce requests to Aranea we would still need to know both the original servlet path and some Aranea-specific data like the identifier of the component hosting the integrated module.

To solve the first problem we make use of servlet *filters*. We implement the `IntegrationFilter` filter that saves the current URL as a request parameter and after that redirects to the Aranea servlet, then map this filter to all of the integrated application servlets.

To solve the second one we need to wrap the `HttpServletResponse` in the encapsulating component. `HttpServletResponse` is the Java Servlet API abstraction for the HTTP response to the submitted request, and besides the output stream and headers it has a method `encodeURL()`. This method is used by the Servlet API to allow for cookieless session tracking. To support this it must be called on all the URLs embedded in the generated HTML output.

By overriding its default implementation we can add encoded Aranea-specific information that allows `IntegrationFilter` to route the request to the intended Aranea component. The resulting `ResponseWrapper` class may look something like this:

```
class ResponseWrapper extends HttpServletResponseWrapper {
  //...

  public String encodeURL(String url) {
    StringBuffer urlB = new StringBuffer(url);

    urlB.append(url.indexOf('?') != -1 ? '&' : '?');
    urlB.append("araInfo=")
    .append(':').append(topServiceId)
    .append(':').append(threadServiceId)
    .append(':').append(widgetEventPath);

    return super.encodeURL(urlB.toString());
  }

  //...
}
```

The filter can then decode this information and dispatch the request to the Aranea servlet that will cause the hosting component to set up the encapsulation and dispatch the request further to the integrated module. This will appear as if the navigation took place inside the component, since other components will not change.

3.2.3 Rendering

Now let's look into rendering the hosting widgets[4] and the integrated modules. There are two main issues with rendering:

1. It would be logical to dispatch the request to the integrated module on `render()` call. However we have to keep in mind that the integrated module has access to the

[4]Services need no additional support and can just dispatch the encapsulating request to the appropriate URI from the `action()` call.

Aranea API and thus can modify the component hierarchy e.g. by starting a new flow. In this case rendering would be incorrect, as the flow container will render the old flow (possibly failing).

2. Not every framework is capable of just re-rendering the current "page". Since in Aranea only one component at a time receives an event, all the rest must re-render their previous state. But for a framework like Struts this might mean calling an action and possibly producing undesired side-effects.

Both problems admit one and the same solution. To allow component hierarchy modification we do the actual rendering during an event `event()` call. We then cache the resulting output as a character array and write it out during the actual render phase. We save the cached character array between requests and render it again if no event comes. Since at each request only one component gets an event call (typically the one that originated the request), the rest of the components will just display the previously cached response that is identical to the previous one.

To cache the output we must wrap the output stream provided by the `HttpServletResponse`. We can do this by adding a `getOutputStream()` method to the `ResponseWrapper` that returns the `OutputStreamWrapper`[5]:

```
class OutputStreamWrapper extends ServletOutputStream {
  private ByteArrayOutputStream out;

  OutputStreamWrapper() {
    reset();
  }

  public void write(int b) throws IOException {
    out.write(b);
  }

  //...

  public void flush() throws IOException {
    out.flush();
  }

  public void reset() {
    out = new ByteArrayOutputStream(20480);
  }

  public byte[] getData() {
```

[5]We will also need to wrap it in a `PrintWriter` for the `getWriter()` method.

```
    return out.toByteArray();
  }
}
```

Although we fixed one problem, we introduced another. Since we want the integrated modules to be able to include other widgets, they will also re-render by returning the cached response. However widgets are perfectly capable of re-rendering themselves and it is not a good idea to cache their output. The solution is to cache only those parts of response that are produced by the integrated module, not by included widgets. To do that we need to introduce a new tag `<ui:hostedWidgetInclude>` that would call the hosting widget `renderWidget()` method. Then we can create an array of closures that will render either the cached output or the included widget in the correct order as shown on Figure 3.1.

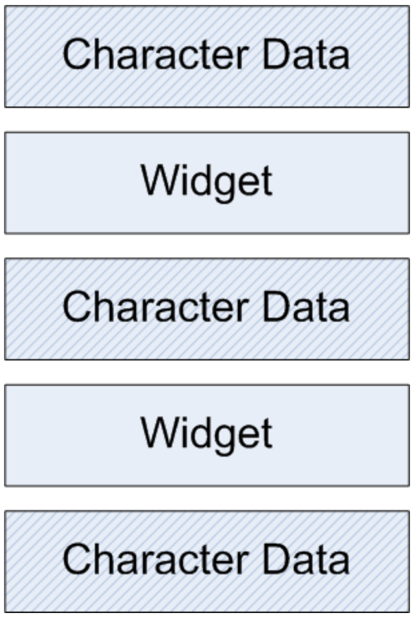


Figure 3.1: Renderer chain

3.2.4 View Sanitizing

Although the response generated by the integrated modules is partially preprocessed as described in subsection 3.2.2, it may still contain some HTML not acceptable in an Aranea component. This is mostly due to HTML limitations:

1. Every HTML page can contain only one `html` tag, `head` tag and `body` tag.
2. `form` tag cannot be nested. Since Aranea defines a single root form, integrated modules are not allowed to define their own forms.
3. Form element names must be prefixed with the component identifier to allow their later separation as described in subsection 3.2.1.

These problems can be solved by one of the two approaches:

Conversion The simplest solution is to change the integrated module view code and introduce the necessary changes (remove root tags like `html`, remove `form` tag and prefix the element names). Most probably some associated changes will also have to be done (Javascript references to form elements, form submission logics, etc). The problem with this solution is that it implies a significant effort unless the logic was already well encapsulated, which is a relatively rare case.

Postprocessing A different approach is to render the response and the postprocess it by parsing the resulting HTML and applying the changes automatically. This approach has the benefit of allowing in simpler cases to just drop the application in and get integration to work in a number of minutes. The main drawback is the inevitable decrease in performance.

In reality full conversion should only be used if the performance penalty is unacceptable. The overhead is not as great as could be expected, since it does not involve Input/Output, which is the main bottleneck in web applications.

Typically postprocessing is combined with some amount of conversion, since some things are simpler to postprocess while other are simpler to just change in place. The ratio of one to another depends on the particular application.

Postprocessing is applied on `render()` call and uses the cached response gather during the `event()` call.

AJAX

The final issue to consider is handling AJAX calls from the integrated module. By itself it does not differ much from usual integration, except that Aranea widgets are by default synchronized, which may cause trouble if the request is expected to be processed asynchronously.

The request will be submitted to some third-party servlet and should be handled by mapping a subclass of the `IntegrationFilter` to the servlet. This subclass should be able to distinguish AJAX requests from common ones (this distinction being, of course, framework-specific) and send AJAX requests to the hosting widget as an unsynchronized action call. The hosting widget should also define this action, which should just dispatch the request to the integrated module after setting up request encapsulation.

3.3 Legacy Migration

Integration allows to integrate several applications together as illustrated by Figure 3.2. However in addition to that it also allows to migrate legacy applications from outdated technologies to newer ones.

To do that we need to prepare the application as follows:

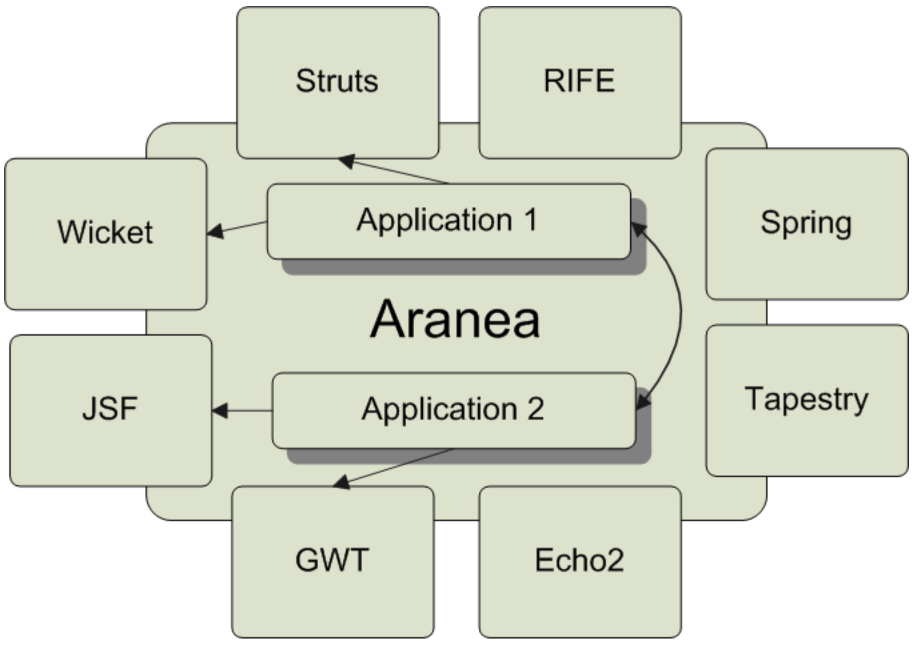


Figure 3.2: Web integration illustration.

1. *Implement adapters for in-house web frameworks.* A typical legacy web application is likely to use some in-house web framework. We need to introduce a specific Aranea adapter component that will handle the framework-specific quirks.

2. *Analyze the application design.* Since most application do not use Object-Oriented technologies we often need to reverse-engineer the design behind them. Mainly we are interested in splitting the application into logical use cases and identifying their interdependencies.

3. *Convert the legacy application to use Aranea integration.* This involves rewriting some basic features and wrapping use cases in Aranea widgets. We also need to lift all communication among use cases to use Aranea API. The result is a fully Object-Oriented application with components implemented using the legacy technology.

After the application is prepared we may start implementing new use cases using any other technology integrated with Aranea. We do it by defining a subclass of the wrapping widget corresponding to the implementation technology. Since all communication among use cases is done in terms of Aranea API the legacy components cannot be distinguished from the non-legacy ones.

To actually migrate away from the outdated technology we apply this simple rule–rewrite legacy use cases only when business requirements change. Since in such a case a significant portion of the use case would have to be rewritten anyway and the development costs using an outdated technology are higher, the migration overhead cost is very low.

We call such an approach *step-by-step legacy migration*, since instead of doing an all-at-once rewriting of the system functionality we run the old and new technologies side-by-side and migrate only when costs and time allow.

In addition to the direct economical benefit the following extra benefits are attained:

- Since development of features requiring a new technology can start immediately the time-to-market decreases.
- Newer frameworks might provide features that increase usability without any extra effort from the developers.
- Object-Oriented design means that development is easier to scale, since particular use cases can be written by any third-party in technology of their choice.
- When a newer and better technology appears the enterprise is somewhat insured against the migration costs, since the application design is already technology-agnostic and migration to the newer beneficial technology can begin immediately.

Chapter 4

Extensions

This chapter describes different extensions to the core Aranea framework.

4.1 Blocking Calls and Continuations

Consider the following scenario:

1. When user clicks a button, a new subflow is started.
2. When this subflow eventually completes we want to assign its return value to some text field.

In event-driven programming model the following code would be typical:

```
OnClickListener listener = new OnClickListener() {
  void onClick() {
    Handler handler = new Handler() {
      void onFinish(Object result) {
        field.setText((String)result);
      }
    }
    getFlowCtx().
      start(new SubFlow(), handler);
  }
}
button.addOnClickListener(listener);
```

What strikes here is the need to use multiple event listeners, and as a result writing multiple anonymous classes that are the Java equivalents of syntactical closures. What we would like to write is:

```
OnClickListener listener = new OnClickListener() {
  void onClick() {
    String result = (String)getFlowCtx().
             call(new SubFlow());
    label.setText(result);
  }
}
button.addOnClickListener(listener);
```

What happens here is that flow is now called using blocking semantics. Going further in this direction, we would like to get rid of all event handlers in this example:

```
button.waitOnClick();
String result = (String)getFlowCtx().
         call(new SubFlow());
label.setText(result);
```

Essentially, we would like to allow waiting for arbitrary events, even ANDs and ORs of events—any monotonous propositions.

Typically blocking behavior is implemented by suspending executed thread and waiting on some concurrency primitive like semaphore or monitor. The disadvantage of such solutions is that operating system threads are expensive and using an extra thread for each user session would be a major overkill—most application servers use a limited pool of worker threads that would be exhausted very quickly. Besides, threads cannot be serialized and migrated to other cluster nodes. A more conceptual problem is that the suspended thread contains information regarding processing of the whole web request, whereas it can be woken up by a different web request. Also, in Java blocking threads would retain ownership of all monitors.

In [11, 21] *continuations* were proposed to solve the blocking problem in web applications, described above. Continuation can be thought of as a lightweight snapshot of thread call stack that can be resumed multiple times. In the context of this problem the differences between continuation and thread is that continuation is much more lightweight in terms of OS resources, can be serialized, and can be run multiple times.

There still remains the problem that both thread and continuation contain information regarding processing of the whole request, but can be woken up by a different web request. To solve it we can use *partial continuations* [13]. Essentially the difference is that the snapshot of the call stack is taken not from the root, but starting from some stack frame that we will call *boundary.* In case of Aranea, the boundary is the stack frame of the event handler invocation that may contain blocking statements. So in case of our previous example the boundary will be the invocation of method `onClick()`:

```
OnClickListener listener = new OnClickListener() {
  @Blocking
  void onClick() {
    String result = (String)getFlowCtx().
             call(new SubFlow());
    label.setText(result);
  }
}
button.addOnClickListener(listener);
```

To wait for an event we should do the following:

1. Take current partial continuation,
2. Register it as an event handler,
3. Escape to the boundary.

A similar approach can be also applied to services though mimicking such frameworks as Cocoon [24] and RIFE [2]. We'd like to stress that by applying continuations to widget event handlers we can create a more powerful programming model because there can be simultaneous linear flows at different places of the same widget hierarchy, e.g. in each flow container. This programming model is similar to that of the Smalltalk web framework Seaside [6] that uses continuations to provide analogous blocking call semantics of flows, but not event handlers in general.

Java does not support continuations in any form, but luckily there is an experimental library [29] that allows suspending the current partial continuation and later resuming it:

```
Runnable myRunnable = new Runnable {
  void run() {
    ...
    Continuation.suspend();
    ...
  }
}
Continuation cont1 =
  Continuation.startWith(myRunnable);
...
Continuation cont2 =
  Continuation.continuteWith(cont1);
```

Altogether we view blocking calls as “syntactic sugar” above the core framework. At the same time we find that combining event-based and sequential programming in a component framework is a powerful concept, since different parts of application logic can be expressed using the most suitable tool.

4.2 Code Reloading

Zero turn-around usually refers to the way changes made to the code are immediately visible in most interpreted languages. Indeed, since there is no compilation and almost no deployment, the time from making a change to seeing it in the browser is a fast as pressing Ctrl+S, Alt+Tab and F5. The situation in the Java language [10] is different—although compilation is relatively fast, together with deployment, cache reloads and framework initialization it might easily take up to a minute to see a change propagate in a large application.

The problem is that JVM specification [18] clearly states that we can't just reload code of a single class after it has been loaded into the JVM. HotSpot JVM offers some way by allowing to replace the code of the methods, but so far it forbids to anyhow alter structure and signatures, so it is only of limited use. What we can do is to load classes using a particular (possibly our own) classloader, and then let it to be garbage collected among with all loaded classes. Then next time an instance is created the class will be loaded anew.

However even this won't give us the desired result, as any existing instance of a class will still hold on to its previous definition (with the old classloader) and now we might actually have two conflicting class definitions in two different classloaders which can lead to all sorts of trouble. So we also need to somehow reload the object state in a new classloader.

We can achieve this with serialization—we just need to modify the `ObjectInputStream` to take a classloader parameter and load the instances using its classes. Thus what we have to do is:

1. Load the classes using our classloader.
2. Serialize the state of the instances
3. Drop the old classloader and create a new one
4. Deserialize state of the instances in the new classloader

Of course if some objects cannot be serialized (e.g do not implement `Serializable` or contain non-serializable fields) the whole scheme fails, since we cannot preserve their state.

Let's start with the reloading procedure that takes an object and reloads it using a fresh classloader. We know it should work through serialization, so it should look approximately like the following:

```
...
private Serializable reload(Serializable child)
  throws Exception {
  return deepCopy(newClassLoader(), child);
}
...
```

`deepCopy` should just serialize and deserialize the object. There is one catch though—it should also resolve the classes using our own classloader, so we have to make a subclass of `ObjectInputStream`:

```
...
private static class ReloadingObjectInputStream
extends ObjectInputStream {
  private ClassLoader cl;

  public ReloadingObjectInputStream(
      ClassLoader cl,
      InputStream in)
    throws IOException {
    super(in);
    this.cl = cl;
  }

  /*
   * This method is used to resolve the classes
   * when creating object instances.
   */
  protected Class resolveClass(ObjectStreamClass desc)
    throws IOException, ClassNotFoundException {
    String name = desc.getName();
    return cl.loadClass(name);
  }
}
...
```

The `deepCopy()` method itself is straightforward and we could have used Apache `SerializationUtils` methods if it weren't for the custom `ObjectInputStream`:

```
...
private  Serializable deepCopy(
    ClassLoader cl,
    Serializable original) throws Exception {
  //Serialize to a byte array
  ByteArrayOutputStream baos =
    new ByteArrayOutputStream(512);
  ObjectOutputStream out =
    new ObjectOutputStream(baos);
  try {
    out.writeObject(original);
  }
  finally {
    out.close();
  }
  byte[] serialized = baos.toByteArray();

  //Deserialize to an instance
  ByteArrayInputStream bais =
    new ByteArrayInputStream(serialized);

  ReloadingObjectInputStream in =
    new ReloadingObjectInputStream(cl, bais);
  Object obj = in.readObject();

  return (Serializable) obj;
}
...
```

We could have used a usual classloader (most common is `URLClassLoader`) and point its classpath to the `/WEB-INF/classes`. However there is also the problem that the same classes are in the classpath of the web application classloader. The Java specification instructs classloaders to try loading classes with the parent classloader first, however in our case we would be able to reload those classes, so we need to invert this preference by delegating to the parent only if we cannot find the class in the classpath:

```
...
private static class ReloadingClassloader
  extends URLClassLoader {

  public ReloadingClassloader(
      URL[] urls,
      ClassLoader parent) {
    super(urls, parent);
  }

  public Class loadClass(String name)
    throws ClassNotFoundException {
    //If already loaded just return
    Class c = findLoadedClass(name);
    if (c != null)
      return c;

    //First try own classpath
    //then delegate to parent
    try {
      return findClass(name);
    }
    catch (ClassNotFoundException e) {
      return super.loadClass(name);
    }
  }
}
...
```

We create the classloader by putting `/WEB-INF/classes` into the classpath:

```
...
private ClassLoader newClassLoader()
  throws MalformedURLException {
  //Get the ServletContext
  ServletContext sctx =
    (ServletContext) getEnvironment().getEntry(
        ServletContext.class);
  //Return a classloader for "/WEB-INF/classes"
  return new ReloadingClassloader(
      new URL[] {sctx.getResource("/WEB-INF/classes")},
      getClass().getClassLoader());
}
...
```

Now we have most of the machinery in place and before we move on to implementing the Aranea filter we can inspect the limitations of the solution. Obviously it will only work on serializable classes. In addition to get actual gain from it all classes should have `serialVersionUID` set to some fixed number (e.g. "0"). This should be done so that small changes in the method and class signatures wouldn't cause a deserialization failure. Even then removing a class (even an inner or anonymous class) will cause serialization to fail as well as changing the field types or order. However it is still much more than HotSwap would allow (at least at the moment).

Now we can go on with implementing the Aranea filter that will do the work. There are several reasons why this solutions suits Aranea so well:

- Aranea is originally assembled from independent components by containment. Therefore one filter can define the classloader to load its children and do the rest of the tricks.
- All application widgets in Aranea are serializable and loaded by the parent classloader (most of the filters are configured by Spring and loaded by its classloader).
- Aranea components can only access their parents through the environment, which can be taken away at any given moment. Moreover parents have references only to

their direct children, and can communicate with the indirect children only through messages.

Thus we can create a filter and put it in appropriate place just above the application widgets and we will be able to seamlessly reload all application widgets code.

At the moment we will assume that creating a new classloader and serializing and deserializing does not visibly affect response time in development, so we will just reload all classes before every request. Thus we do not need to check exactly which classes have changed, since any possible changes will be reloaded.

Let's start by creating the filter service itself. It will have to take the child class name as a string, since we will need to load it reflectively in a freshly created classloader.

```
public class StandardClassReloadingFilterWidget
    extends BaseApplicationWidget {

  private String childClassName;
  private RelocatableWidget child;

  public void setChildClass(String childClass) {
    this.childClassName = childClass;
  }

  protected void init() throws Exception {
    //Create the classloader and use it
    //to load the child class
    ClassLoader cl = newClassLoader();
    Class childClass = cl.loadClass(childClassName);

    //Create an instance of child class and
    //attach it to the filter
    child =
      new RelocatableDecorator(
          (Widget) childClass.newInstance());
    addWidget("c", child)'
  }
...
```

Here we create the child widget reflectively in the classloader and add it under a name "c". The only interesting part is that we also wrap it into a `RelocatableDecorator`, which we will return to later. The following method will do the actual reloading:

```
...
protected void update(InputData input)
```

```
  throws Exception {
  try {
    //Remove all references to parents and
    //reload the child classes
    child._getRelocatable().overrideEnvironment(null);
    child = (RelocatableWidget) reload(child);
  }
  catch (ClassNotFoundException e) {
    log.error("Failed to reload widget classes", e);
  }
  finally {
    //Restore the references to parents
    child._getRelocatable()
      .overrideEnvironment(getEnvironment());
  }

  //Reattach the child new instance to the filter
  _getComposite().attach("c", child);
}
...
```

Here the point of wrapping the child into the `RelocatableDecorator` comes out—as all references from children to parents in Aranea have to go through the environment, by removing the environment from the child we also remove all possible links up allowing to serialize children only.

Now the filter is complete (except for the trivial `render()` method) and can be tested in e.g. main example or any other Aranea application by adding a similar configuration entry:

```
<bean id="araneaApplicationStart" singleton="false"
  class="org.araneaframework.framework.filter.
         StandardClassReloadingFilterWidget">
  <property name="childClass"
    value="org.araneaframework.example.main.web.DevelWidget"/>
</bean>
```

instead of the usual:

```
<bean id="araneaApplicationStart"
  class="org.araneaframework.example.main.web.LoginWidget"
  singleton="false"/>
```

The `DevelWidget` should start the `LoginWidget` in a `StandardFlowContainerWidget`. Otherwise the reloading filter is lost on login.

Conclusions

As the enterprise applications get bigger and older more and more technologies come up that help to fight the arising complexity. In the business layer there is a number of established technologies, both implementation-level (Spring [14] and similar) and protocol-level (Web Services [1], SOA [7] and similar). However in the web layer no such technologies are known to be widely applied[1].

Aranea was created to solve these problems in the web layer. In this thesis we have described Aranea minimalistic component model, the web development framework built around it and shown a case study illustrating Object-Oriented concepts application. Descriptions of several extensions that simplify web development have also been included in the thesis.

We have described the integration facilities built on top of the framework and the component model that allow component level web application integration. We have also described an application of these facilities to allow step-by-step migration from outdated web development technologies and frameworks.

In addition we have developed the open source project *Aranea MVC* implementing the described principles. The project is available from http://www.araneaframework.org and has a stable community. The distribution had more than 2700 downloads since the first release on 22nd February 2006.

The proposed web development approach has been successfully applied in more than seven projects known to the author providing a boost in productivity and simplifying maintenance. The proposed web integration approach is in testing phase, and has so far been applied to one project.

[1]WSRP [32] supports some integration and decomposition, but is specific to portal applications.

Aranea Veebiraamistik

Jevgeni Kabanov

Magistritöö

Kokkuvõte

Kuna ärirakendused kasvavad suuremaks ja järjest vananevad, on vajadus uute tehnoloogiate järele, mis aitavad nendega kaasnavatest keerukustest jagus saada. Ärikihis on tekkinud hulk ennast tõestanud tehnoloogiad, nii implementatsiooni (Spring [14] jms) kui ka protokolli tasemel (*Web Services* [1], SOA [7] jms). Samas ei ole veebikihis sellised tehnoloogiad laialdast kasutust leidnud.

Aranea on objekt-orienteeritud minimalistlik komponentmudel, veebiarenduse- ja veebiintegratsiooniraamistik, mis on suunatud suurtele ärirakendustele. Selles töös kirjeldame Araneat ning näitame, kuidas sellega saab lahendada tüüpilisi arendus- ja integratsiooniprobleeme.

Esimene peatükk on sissejuhatus sellest, kuidas rakendatakse OOP metoodikat Aranea puhul ning tutvustab ka juhtumuuringut, mis illustreerib ülejäänud töö poolt kirjeldatud printsiipe. Peatükk põhineb tutorialil [15], mis on kirjutatud Aranea esmakasutajate tarbeks. Teine peatükk tutvustab lugejale Aranea raamistiku komponente, abstraktsioone ja ülesehitust. Keskendutakse pigem raamistiku põhiideedele, mitte niivõrd implementatsiooni detailidele. Peatükk baseerub konverentsiartiklil [20]. Kolmas peatükk kirjeldab, kuidas raamistiku komponente saab kasutada veebirakenduste integratsiooni võimaldamiseks. Neljas, ja ühtlasi viimane, peatükk käsitleb Aranea laiendusi: blokeeruvad jätkud ning koodi ülelaadimine. Neist esimene põhineb konverentsiartiklil [20] ning teine veebiblogis publitseeritud artiklil [16].

Töö põhipanus on avatud lähtekoodiga Aranea projekt, mis hõlmab üle 100 tuhande koodirea ning üle 200 lehekülje dokumentatsiooni mitmete näidete ja alamprojektidega. Projekti algatajateks olid autor koos Oleg Mürkiga ning seda arendas autori poolt juhitud Aranea meeskond. Projekti distributsioon on kättesaadav tööga kaasneval CD plaadil ning projekti koduleheküljel `http://www.araneaframework.org`.

Pakutud lähenemine veebiarendamisele on leidnud edukalt rakendamist rohkem kui seitsmes autorile teadaolevas projektis, tõstes produktiivsust ning lihtsustades hooldust. Veebiintegratsiooniraamistik on testimisfaasis ning on siiani olnud kasutusel ühes projektis.

Bibliography

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, 2004.

[2] G. Bevin. RIFE. Available at http://rifers.org/, visited on 12th May 2007.

[3] G. Bevin. RIFE/Crud. Available at http://rifers.org/wiki/display/rifecrud/, visited on 12th May 2007.

[4] M. Boiko. Java web controller frameworks. B.Sc. thesis, University of Tartu, 2005.

[5] B. Cox. *Object oriented programming: an evolutionary approach*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1986.

[6] S. Ducasse, A. Lienhard, and L. Renggli. Seaside — a multiple control flow web application framework. *ESUG 2004 Research Track*, pages 231–257, September 2004.

[7] T. Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2004.

[8] M. Fowler. Inversion of Control Containers and the Dependency Injection pattern. *Actualizado el*, 23.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1996.

[11] P. T. Graunke, S. Krishnamurthi, V. der Hoeven, and M. Felleisen. Programming the web with high-level programming languages. In *European Symposium on Programming (ESOP 2001)*, 2001.

[12] B. Henderson. *The Gospel of the Flying Spaghetti Monster*. Villard Books, 2006.

[13] R. Hieb, K. Dybvig, and C. W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110, 1994. Available at http://citeseer.ist.psu.edu/hieb93subcontinuations.html, visited on 12th May 2007.

[14] R. Johnson. Spring. Available at http://springframework.org, visited on 12th May 2007.

[15] J. Kabanov. Aranea Introductory tutorial. Available at http://www.araneaframework.org/docs/intro/html/, visited on 12th May 2007.

[16] J. Kabanov. Zero turn-around in Java? Blog post, available at http://blog.araneaframework.org/2006/11/21/zero-turn-around-in-java/, visited on 12th May 2007.

[17] B. Kernighan and D. Ritchie. *The C programming language*. Prentice-Hall, 1988.

[18] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[19] C. R. McClanahan. Apache Struts project. Available at http://struts.apache.org/, visited on 12th May 2007.

[20] O. Mürk and J. Kabanov. Aranea: web framework construction and integration kit. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 163–172, New York, NY, USA, 2006. ACM Press.

[21] C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 23–33, 2000.

[22] P. Thiemann. An embedded domain-specific language for type-safe server-side web-scripting. Available at http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASH/, visited on 12th May 2007.

[23] Ajax. Wikipedia encyclopedia article available at http://en.wikipedia.org/wiki/AJAX, visited on 12th May 2007.

[24] Apache Cocoon project. Available at http://cocoon.apache.org/, visited on 12th May 2007.

[25] ASP.NET. Available at http://asp.net/, visited on 12th May 2007..

[26] Cascading Style Sheets. Available at http://www.w3.org/Style/CSS/, visited on 12th May 2007.

[27] Jakarta Tapestry. Available at http://jakarta.apache.org/tapestry/, visited on 12th May 2007.

[28] Java Servlet 2.4 Specification (JSR-000154). Available at http://www.jcp.org/aboutJava/communityprocess/final/jsr154/index.html, visited on 12th May 2007.

[29] The Javaflow component, Jakarta Commons project. Available at http://jakarta.apache.org/commons/sandbox/javaflow/index.html, visited on 12th May 2007.

[30] JavaServer Faces technology. Available at http://java.sun.com/javaee/javaserverfaces/, visited on 12th May 2007.

[31] JavaServer Pages Technology. Available at http://java.sun.com/products/jsp/, visited on 12th May 2007.

[32] OASIS Web Services for Remote Portlets. Available at www.oasis-open.org/committees/wsrp/, visited on 12th May 2007.

[33] Open source web frameworks in Java. Available at http://java-source.net/open-source/web-frameworks, visited on 12th May 2007.

[34] Ruby on Rails. Available at http://www.rubyonrails.org/, visited on 12th May 2007.

[35] Spring Web Flow. Available at http://opensource.atlassian.com/confluence/spring/display/WEBFLOW/, visited on 12th May 2007.

[36] The Extensible HyperText Markup Language. Available at http://www.w3.org/TR/xhtml1/, visited on 12th May 2007.

[37] WebWork, OpenSymphony project. Available at http://struts.apache.org/, visited on 12th May 2007.

[38] Wicket. Available at http://wicket.sourceforge.net/, visited on 12th May 2007.