Aranea—Web Framework Construction and Integration Kit

Oleg Mürk
oleg.myrk@gmail.com

Jevgeni Kabanov
ekabanov@gmail.com

ABSTRACT

Currently there exist dozens of web controller frameworks that are incompatible, but at the same time have large portions of overlapping functionality that is implemented over and over again. Web programmers are facing limitations on code reuse, application and framework integration, extensibility, expressiveness of programming model, and productivity.

In this paper we propose a minimalistic component model Aranea aimed for 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 implementing the same features over-and-over again. Open source implementation of Aranea framework together with reusable controls, such as input forms and data lists, and a rendering engine are ready for real-life applications.

1. 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 [13] let alone commercial products or other platforms like .NET and numerous dynamic languages. Not to mention 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. The advantage of such a situation is that different approaches and ideas are tried out. Indeed, many very good ideas have been proposed during these years, many of which we will describe later in this paper. On a longer timescale the stronger (or better marketed) frameworks and approaches will survive, the weaker will diminish. However, in our opinion, such situation also has a lot of disadvantages.

1.1 Problem Description

First of all let's consider the problems of the web framework ecosystem from the view point of application development. Framework user population is very fragmented as a result of having many incompatible frameworks with similar programming models. Each company or even project is using a different web framework, which requires learning a different skill set. As a result, it is hard to find qualified work force for the selected web framework. For the same reason it's even harder to reuse previously developed application code.

Moreover, sometimes it is useful to write different parts of the same application using different approaches, which might be impossible, because the required frameworks are incompatible. Portal solutions that should facilitate integrating disparate applications provide very limited ways for components to communicate with each other. Finally, there is a lot of poorly designed frameworks that limit expressiveness, productivity, and quality.

System programmers face additional challenges. Creators of reusable components have to target one of the frameworks, consequently their market shrinks. Framework designers implement overlapping features over and over again, each new feature must be added to each framework separately. Many neat ideas cannot be used together because they have been implemented in different frameworks.

We see the main cause of these problems to be the early stage of evolution of web frameworks—no clear winner has emerged yet. Some variation of web frameworks is inevitable because of different application domains have different and conflicting demands: complexity of user interface, application performance, developer productivity, which leads to different programming models and approaches. Also, different programming platforms with different language features and philosophies affect how web frameworks are designed. Finally, there are many social, economical, and political reasons of the problems, but these are out of scope of this paper.

We think that web framework market would win a lot if there were two or three popular platforms with orthogonal philosophies that would consolidate proponents of their approach. Application programmers would not have to learn new web framework at the beginning of each project. Writing reusable components and application integration would be easier and more rewarding. Framework designers could try out new ideas much easier by writing extensions to the framework, while maintaining a large potential user-base.

This is not to say that smaller frameworks would not exist—completely new approaches always have to be tried out. Some of these frameworks would survive and become big players, others would diminish. We predict that web framework ecosystem will converge to this situation anyway, we just think that it needs some hints to converge faster.

1.2 Contributions

In this paper we will describe a component framework that we named *Aranea*. Aranea is written in Java and allows assembling server-side controller web frameworks out of reusable components and patterns. Aranea applications are pure Java and can be written without any static configuration files. In Section 2 we describe our approach and motivation. We find that one of the strengths of this framework is its conceptual integrity—it has very few core concepts that are applied uniformly throughout the framework. The number of core interfaces is small, as is the number of methods in the interfaces. Components are easy to reuse, extend, and test, because all external dependencies are injected into them. The details of the Aranea core abstractions are explained in Section 3.

In different configurations of Aranea components we can mimic principles of most existing server-side web controller frameworks and combine most of patterns in use, but we can also use these configurations together. Possible configurations are described in Section 4. We concentrate on implementation of server-side controllers, but we also intend to support programming model where most of UI is implemented on the client-side and server-side contains only coarse-grained stateful components corresponding roughly to activated use-cases.

Of particular interest is configuration supporting programming model that allows expressing rich user interface as dynamic hierarchical composition of components and maintaining call stacks of nested processes (we call them *flows* further) at arbitrary place in the hierarchy. As an example of rich user interface at extreme, consider Figure 1: multiple interacting windows per user session, each window contains a stack of flows, flows can call nested flows that after completing return values, flows can display additional UI (side-menu, context information) even when a nested flow is executing, flows can contain tabbed areas, wizards, with input forms, lists, other controls, and even other flows.

Further, the framework facilitates both event-based and sequential programming (using continuations). The programming model is quite similar to the one used in Smalltalk web framework Seaside [25], but has completely different implementation and is more general in terms of where sequential programming can be used. This topic is discussed in Section 8.1 as one of extensions.

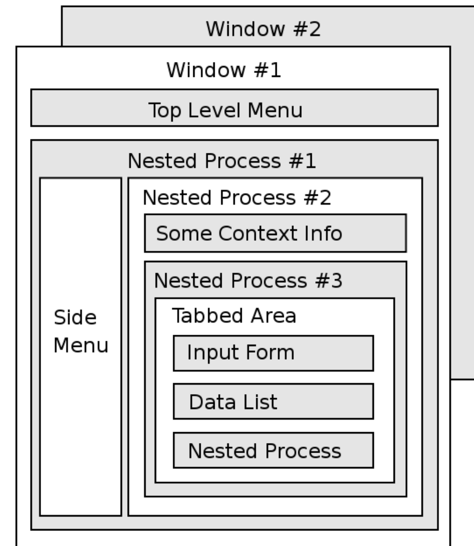

Figure 1: A sketch of a rich user interface.

All web frameworks have to handle such aspects as configuration, security, error handling, and concurrency. We explain how Aranea handles these issues in Section 5.

As a side-product, a reusable component library was created featuring input forms and validation, data lists, tabbed dialogs and wizards, tree control, etc. Another side product was a rendering framework based on Java Server Pages [11], which allows describing the layout of user interface without redundancy of W3C XHTML [22] and CSS [5]. We will describe these additions shortly in Section 6.

We see, as one of the most important differentiating factors of Aranea, its ability to serve as a vehicle for integration of existing frameworks, thanks to its orthogonal design. In Section 7 we show how Aranea can be used in one of the following roles:

Host Aranea hosts components of other controller framework.

Guest Aranea components used within other controller framework.

Protocol Components of two frameworks communicate with each other through Aranea interfaces.

The components mentioned can be either reusable controls or whole application specific use-case implementations.

Finally, we see Aranea as a research platform. It it very easy to try out a new feature without having to write a whole web framework. A framework is assembled out of independent reusable components so essentially everything can be reconfigured, extended, or replaced. If Aranea becomes popular,

writing a new component for Aranea would also mean a large potential user base.

Naturally, there are still numerous framework extensions to be made and further directions to be pursued. These are described in Section 8. Last, but not least, Aranea is based on great ideas originating from prior work of many people. When possible, we reference the original source of idea at the time of introducing it. In Section 9 we compare Aranea with existing work.

2. BACKGROUND

As we mentioned in the introduction, our aim is to support most of programming models and patterns available in existing controller frameworks. We present here, unavoidably incomplete and subjective, list of profound ideas used in contemporary web controller frameworks.

The first important alternative is using stateless or reentrant components for high performance and low memory footprint, available in such frameworks as Struts [3] and WebWork [23].

Another important approach is using hierarchical composition of stateful non-reentrant components with event-based programming model, available in such frameworks as JSF [10], ASP.NET [4], Seaside [25], Wicket [24], Tapestry [7]. This model is often used for developing rich UI, but generally poses higher demands on server's CPU and memory.

The next abstraction useful especially for developing rich UI is nested processes, often referred to as *modal* processes, present for instance in such web frameworks as WASH [30], Cocoon [2], Spring Web Flow [21], and RIFE [17]. They are often referred to by the name of implementation mechanism—*continuations*. The original idea comes from Scheme [29], [26].

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 [25], 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, coined Ajax [1]. 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.

We would also like to support different forms of metaprogramming such as domain specific language for describing UI as a state machine in Spring Web Flow [21] and domain-driven design as implemented in Ruby on Rails [19] or RIFE/Crud [18]. This often requires framework to allow dynamic composition and configuration of components at run-time.

2.1 Pitfalls

We have also studied implementations of many frameworks and would like to avoid pitfalls that we have identified.

Some frameworks designs suffer such problems as poor decomposition—core interfaces with 50-100 methods, and lack of conceptual integrity—many core concepts with limitations on how to combine them. One example of the former is when external interfaces (framework contract) and internal interfaces (for custom logic) are mixed together. An example of the latter would be creating a sophisticated component model to compose pages but providing only trivial navigation model between them.

Another common trouble is static decomposition of application—the structure of application is defined at compile time and new component instances cannot be configured at run-time. In some cases components can be assembled at run-time, but only before they are executed, which is limiting with very dynamic UI.

Historically, there has been very little encapsulation of component state—unnecessary shared state is kept for instance in session data scope. This also makes harder to decide when data should be released because it is unclear who else is using it.

Commonly component life-cycle management is implicit—components are instantiated based on information present in the web request. This makes implementing access controls more laborious and error-prone, because the developer cannot make any assumptions about when the component will be executed and what values will be contained in the (shared) data scope.

Finally there are imposed limitations on how components can interact with each other. Often the only way is either shared data scope or redirecting.

Such problems are well described in the introduction of article on Seaside [25] and are compared to the *goto* programming style popular in 70-s and earlier.

2.2 Our Approach

Based on these observations we decided to design Aranea using object-oriented principles like encapsulation and polymorphism. Components are plain Java objects and we minimize usage of static configuration files. This allows to have statically typed component system that is very dynamic and can be reconfigured at the run-time. We use dependency injection where possible to make components as reusable and testable as possible.

The main approach is hierarchical composition of components with event-driven programming model. The structure of component composition is dynamic and can be changed at the run-time. State is represented as Java fields of component instances. Dependency injection gets an interesting twist in case of hierarchical components—the way child components are configured depends on which parent they belong to. Also it is often very convenient to inherit dependencies of a parent to a child component. Our solution to this problem is based on something that we call *environment*, described in the next section.

We use by default explicit state transition management—the structure of component hierarchy does not depend on

parameters of web request and changes only when explicit statement to do that is executed. Such approach also makes authorization much easier, as will be described in Section 5. Components can communicate among each other as normal Java objects. We support communication both through shared data scopes and method invocation. Execution model is such that each web request is processed on one Java thread, which makes system considerably easier to debug.

Nested processes are supported by introducing a component that is a call stack containing special components—flows. Call stack components can be present at any level of component hierarchy. One could say that call stack is just a form of component composition. Flows have event-based interface—when starting a subflow, event handler is registered that will be notified when subflow returns. Blocking semantics of flow invocation can be implemented in a more general way as taking continuation and registering it as an event handler of the event until which we need to block (e.g. subflow returns).

3. CORE 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.

3.1 Components

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

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

A component in itself is a very simple entity that

- Has a life-cycle which 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.
- Can propagate `Message`s to its children.

Note that 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 can then be combined to create a full identifier that allows tracing the component starting from the hierarchy root. Note also that the hierarchy is in no way static and can be modified at any time by any parent.

However, 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 in the following interface:

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

It is basically a simple yet powerful discovery mechanism allowing the children to discover services (named *contexts*) provided by their parents without actually knowing, which parent has provided it. Looking up a context is done by calling the environment `getEntry()` 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 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). Although `Message` seems simple, it is actually a powerful modification of the *Visitor* pattern. The idea is that a component `propagate(m)` method will just call message `m.send(...)` method 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);
```

3.2 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. We call this component `Service` and it is captured in 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. This leads to an understanding of a service as a building block of the framework itself.

In fact a typical pattern that services realize is that of a *Filter*—a service that has a single child and that will block or modify some requests as well as provide additional functionality by adding a context to its child environment. Although it is similar Servlet [8] filters, the difference is that we can do it in any place in the hierarchy, thus providing different services. Some typical examples include file upload, localization, synchronization, and multi-submit protection filter services.

Another useful pattern that can be realized by services is that of a *Router*. A router has several children, but routes any given request to only one of them using some kind of association, typically a request parameter. A router can have a predefined number of children configured on deployment, it can allow to add/remove children explicitly at any time, or it can create children dynamically on-demand. Common examples of routers include dynamic router that enables adding new services on need and is used for top-level services and popup services, and session router that allocates a service per each user session and routes request to it.

3.3 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 process();
  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. `process()` is also called on all the widgets in the hierarchy allowing them to prepare for rendering whether or not the widget has received an event.
4. `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 after a `process()` call before an `update()` call.

Although widgets also inherit an `action()` method from the services, it may not be called during the widget request-response cycle. The only time it is allowed is after a `process()` call, but before an `update()` call. It may be used to interact with a single widget, e.g. for the purposes of making an asynchronous request through Ajax [1]. Standard widget implementation allows setting event listeners that enable further discrimination between `action()/event()` calls to the same widget. As services widgets may be used as framework building blocks and can realize the *Filter* or *Router* patterns.

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.

3.4 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 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 8, in which case the `Handler` interface is be redundant.

3.5 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 namespaces (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.

4. FRAMEWORK 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 [8] 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 else, but a hierarchy (often looking like a chain) of components fulfilling independent tasks that are arranged together, we cannot just provide a single way of assembling it. Instead we show how to assemble frameworks that can host a flat namespace of reentrant controllers (á la Struts [3] actions), a flat namespace of non-reentrant stateful controllers (á la JSF [10] components) and nested stateful flows (á la Spring Web Flow [21]). Finally we also consider how to merge all these approaches in one assembly.

4.1 Reentrant Controllers

The first model is easy to implement by arranging the framework in a chain by containment (similar to pattern *Chain-of-Responsibility*), which starting from the root would look as follows:

1. Servlet [8] 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).

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 [23] does) or route request to existing reentrant actions (like Struts [3] 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.

4.2 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 [8] 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()/process()/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.

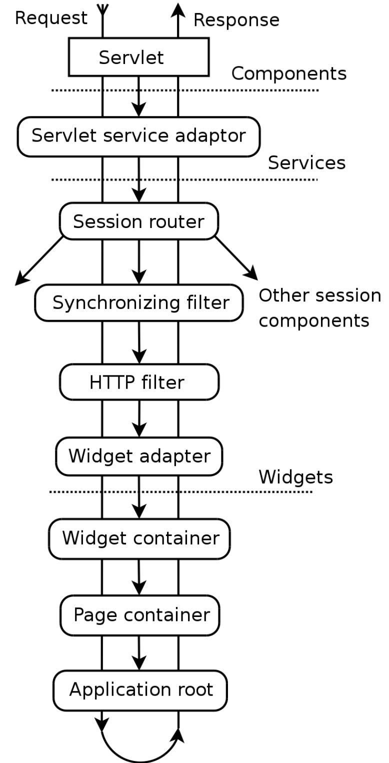

Figure 2: Framework assembly for hosting pages

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 [24], JSF [10], or Tapestry [7] 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.

4.3 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 [8] 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 4.2. 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 [21], 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.

4.4 Combining the Models

It is also relatively easy to combine these models, modifying the model shown on figure 2 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.

5. FRAMEWORK ASPECTS

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

5.1 Configuration

The first aspect that we want to examine is *configuration*. We have repeated throughout the paper 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. We use in our configuration examples Spring [20] 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;
     }
  }
);
```

5.2 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).

5.3 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 behaviour 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 in place 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.

5.4 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.

6. FRAMEWORK 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.

6.1 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 chidren 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 3.

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.

6.2 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 backends, 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.

6.3 Presentation Framework

Finally Aranea also contains a JavaServer Pages [11] custom tag library that not only allows to access services and widgets, but also facilitates expressing user interface with less redundancy than W3C XHTML [22] and W3C CSS [5].

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.

7. INTEGRATION SCENARIOS

In this section we describe our vision of how web controller frameworks could be integrated with Aranea or between each other. In practice, we have so far integrated Aranea only with one internal framework with stateful Portlet-like [15] components, where Aranea components were hosted within the latter framework, but we are considering integrating with such frameworks as Wicket [24], JSF [10], Tapestry [7], Spring WebFlow [21], Struts [3], and WebWork [23].

Integration is notorious for being hard to create generalizations about. Each integration scenario has its own set of specialized problems and we find that this article is not the right place to write about them. For this reason we keep this section intentionally very abstract and high-level and try to describe general principles of web controller framework integration without drowning in implementation details.

In the following we assume that depending on their nature it is possible to model components of frameworks we want to integrate as one of:

- *service-like*—reentrant and/or stateless component[1],
- *widget-like*—non-reentrant stateful component.

Note that both notions consist of two contracts: interface of component and contract of the container of the component.

In our abstraction we have essentially the following integration scenarios:

- service-service,
- service-widget,
- widget-service,
- widget-widget.

Here, for instance, "service-widget" should be read as: "*service*-like component of framework X containing *widget*-like component of framework Y". In homogeneous (i.e. service-service and widget-widget) integration scenarios one has to find a mapping between service (resp. widget) interface methods invocations of two frameworks. Although we don't find this mapping trivial, there is little we can say without considering specialized details of particular frameworks. However, our experience shows that, thanks to minimalistic and orthogonal interfaces and extensibility of Aranea, the task becomes more tractable than with other monolithic frameworks. We now concentrate on heterogeneous cases of server-widget and widget-service integration. They can also occur within Aranea framework itself, but are more typical when disparate frameworks using different programming models are integrated.

In service-widget scenario, generally, each web request is processed by some service and then the response is rendered by possibly a different service, whereas both services can be

[1] Note that Servlets [8] are, for instance, service-like components.

reentrant and/or stateless. As a result, such services cannot host themselves the widgets whose life-time spans multiple requests handled by different services. Consequently, widget instances should be maintained in stateful widget container service(s) with longer life-span. At each request such services would call update() and event() methods of the contained widgets. Widgets would be instantiated by services processing the request and rendered using render() method by services generating the response. Generally, the life-time of a widget container service would not match the required life-times of contained widgets. For this reason some other mechanisms should be used, for instance:

- Each service processing a request should explicitly decide which widgets are to be kept further, all the rest are to be destroyed (within current session).
- Associating with each widget and web request a hierarchical namespace and keeping only those widgets whose namespace is a prefix of web request namespace (within current session).

As the services are generally reentrant, it is important to exclude concurrent access to the widgets belonging to the same session. The simplest solution is to synchronize on session at each web request that accesses widgets.

In widget-service scenario, services should be contained in service container widgets in the position within widget hierarchy most suitable for rendering the service. On widget `update()`, the data entitled for the contained service should be memorized. On widget `render()` the memorized data should be passed to the `action()` method of contained service to render the response. If the service responds with redirection, which means that the request should not be rerun, the service should be replaced with the service to which the redirection points. After that and on all subsequent renderings the `action()` method of the new service should be called with redirection parameters. Most often, requests that lead to redirection are very easy to identify (e.g. HTTP POST vs GET [6]). In this case such requests should be routed as `event()` instead of `update()` invocations and executed there at once instead of waiting for `render()` invocation.

Coming back to not-so-abstract reality, when integrating frameworks the following issues should be handled with care:

- How data is represented in the web request and how output of multiple components coexists in the generated web response.
- Namespaces (e.g. field identifiers in web request) of contained components should not mix with the namespace of container components, which in general means appending to the names a prefix representing location of contained component within the container.
- State management, especially session state history management (browser's back, forward, refresh, and new window navigation commands) and keeping part of the state on the client, should match between integrated components. We explore this topic further in Subsection 8.2.
- A related issue to consider is view integration. Many web frameworks support web components that are tightly integrated with some variant of templating. Consequently it is important that these templating technologies could be intermixed easily.

Incompatibilities in these aspects lead to a lot of mundane protocol conversion code, or even force modifying integrated components and/or frameworks.

Generalized solutions to these issues could be standardized as *Aranea Protocol*. As compared to such protocol, current Aranea Java interfaces are relatively loose—i.e. functionality can be considerably cutsomized by using Message protocol and extending core interfaces (InputData, OutputData, Component) with new subintefaces.

Altogether, we envision the following integration scenarios with respect to Aranea:

Guest Aranea components (resp. services or widgets) are hosted within components of framework X that comply to the Aranea component (resp. service or widget) container contract.

Host Aranea components host components (resp. service-like or widget-like) of framework X through an adapter component that wraps framework X components into Aranea component (resp. service or widget) interface.

Protocol Framework X components provide Aranea component (resp. service or widget) container contract that hosts framework Y components wrapped into Aranea component (resp. service or widget) interface using an adapter component.

8. EXTENSIONS AND FUTURE WORK

In this section we discuss important functionality that is not yet implemented in Aranea. In some cases we have very clear idea how to do it, in other cases our understanding is more vague.

8.1 Blocking Calls and Continuations

Consider the following very simple scenario:

1. When user clicks a button, we start a new subflow.
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 clumsy Java equivalent 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 solution is that operating system threads are expensive, so 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 fast. Besides, threads cannot be serialized and migrated to other cluster nodes. A more conceptual problem is that 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 [29] and [26] *continuations* were proposed to solve the blocking problem in web applications, described above. Continuation can be thought of as a lightweight snapshot of thread's 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 it 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 this problem *partial continuations* [27] can be used. Essentially, the difference is that the snapshot of 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 will be the stack frame of event handler invocation that may contain blocking statements. So in case of our previous example the boundary will be invocation of method `onClick()`:

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

When we need to wait for an event, the following should be executed:

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

Similar approach can be also applied to services though mimicking such frameworks as Cocoon [2] and RIFE [17]. 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 Smalltalk web framework Seaside [25] 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 exists experimental library [9] that allows suspending 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);
```

Aranea currently does not have implementation of this approach, however, it should be relatively easy to do that. Event handlers containing blocking statements should be instrumented with additional logic denoting continuation boundaries. We could use AspectJ [28] to do that.

Altogether we view blocking calls as a rather easily implementable 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 very powerful idea because different parts of application logic can be expressed using the most suitable tool.

8.2 State Management

In this subsection we consider multiple aspects of application state management:

- Optimizing memory server-side memory consumption.
- Keeping session state on the client-side.
- Supporting (or sensibly ignoring) browser's back, forward, refresh, and new window navigation commands.
- Providing semantic back, refresh, and new window functionality in the application.

The fact that a part of session state can be saved into generated web response and later restored when corresponding web request arrives allows decreasing demands on server-side memory consumption. This approach is very natural for frameworks based on reentrant or stateless service-like components. Most often, application's navigation history is represented only within browser's request history.

However such approach either allows very simple UI structure or requires lots of manual copying of state information between server-side representation and web request/response. Stateful non-reentrant widgets provide more productive development model of rich UI. Below we show how client-side state and navigation history can be supported in applications that also have widget-like components.

8.2.1 Optimizing Memory Consumption

In high performance applications low memory consumption of session state representation is essential. The interface of `Component` has methods `disable()` and `enable()` that should release all unnecessary resources. The semantics is such that disabled component can be only destroyed and does not have to respond to other events. Disabling components could be used for flows that wait for subflows to return and with inactive sessions.

What could component do when disabled? For instance input form and data list component are quite memory consuming in default Aranea implementation. Input forms are essentially hierarchical tree of components corresponding to input fields. Normally, these components also memorize last values entered by the user. In order to conserve space, memorized values could be dropped when components are disabled. Even more, the structure of input form is usually determined by very few state variables, so the whole input form could be released and later reconstructed from these variables. Of course, this also generalizes to any components, not just input forms. Data lists usually cache data rows of the visible page and could release this cache when disabled.

An alternative way of server-side state conservation is to store it on the client-side within the web response that later becomes web request. This behavior relies on browser's request history and is described in the next section.

8.2.2 Client-Side State

Widgets can support saving state on the client-side using the following interface:

```
interface Deflatable extends Component {
  void deflate(OutputData)
  void inflate(InputData)
  Essence getEssence(OutputData)
}

interface Essence {
  Component inflate(IntputData)
}
```

It would follow the following logic:

- Method `deflate()` writes out state to be kept on the client-side into `OutputData` and releases internal representation of the state.
- Method `inflate()` reads in information arriving from the client-side from the `InputData` and restores internal representation of the state.
- Methods `deflate()/inflate()` could also recursively call `deflate()/inflate()` on child components.
- When a reference to a child component can be released, `getEssence()` can be called to get an `Essence`, which allows similarly inflating the component, but does not guarantee that object's identity remains the same as before deflating. Component essence, which is presumably smaller than the component itself, should be kept on the server-side.

Before update (resp. after rendering) phase a special Message would traverse widget tree and if the widget is instance of Deflatable call `inflate()` (resp. `deflate()`) method. The data written out by deflate() methods would be stored in the web requst. In order to guarantee integrity, their digest would be kept on the server-side. If confidentiality is needed, it could be also encrypted, although it makes more sense to keep confidential data on the server-side instead.

8.2.3 Navigation History

Working with browser's navigation commands is trivial if there is no navigation state kept on the server-side which often happens with service-based applications. Otherwise the problem essentially boils down to thinking of session as an oriented graph, where nodes are particular states reached so far and arcs show transitions between the states as a result of web request. Its important to remember that the state does not contain all components comprising the framework, but only those instances that are bound to particular session (or even particular window within session). Also in general the state representation will consist of two parts: server-side

part and client-side part encoded in web response and subsequently in corresponding web request. In general this graph is a tree rooted in the initial state, where multiple branches occur because of using browser's navigation commands. If we add expiration of old states the graph becomes an oriented forest. If each response contains the ID of the state that generated it (i.e. the state after generation) then multiple things can be done when corresponding request arrives:

- All states are maintained, split between server-side and client-side representation. When request arrives, session state with corresponding ID is easily reconstructed based on information stored on the server-side and information present in the request. After that the request is executed in this state. The resulting server-side part of the state is serialized (thus facilitating deep-copy) in order to allow processing multiple times requests referring to this state. Such approach integrates best with browser's navigation commands but poses increased demands on server-side memory consumption.
- When request arrives that should not be run multiple times (e.g. money transfer), it should be processed as normal, but instead of rendering, a redirection should be issued that causes rendering of the state resulting from the original request. Rendering is idempotent and can be run multiple times.
- Older states can be removed based on some expiration policy to conserve memory or when application explicitly demands removing all previous states. If user goes back too many times and reaches deleted state, he is warned and told to move forward.
- Only one last state is kept on the server-side. When request with an older ID arrives it is ignored and the last state is rendered, possibly with a warning that back-button, refresh, and new window commands are ignored. This is actually the policy currently implemented in Aranea.
- By comparing the last generated state ID and the ID in the request application can infer that user has issued back or forward commands. This can be translated into a semantic event within the application that could be interpreted, for instance, as pressing cancel button on the flows. Similar tricks can be done to identify when user issues new window command, although this whole approach belongs to the domain of hacking (i.e. unreliable and unaesthetic).

Note that the last two approaches exclude keeping part of the server-side state on the client within web response. Instead we could keep the client-side part of the state in cookies [16] of which only the last value is memorized by the browser.

Maintaining multiple session states on the server-side creates problems of memory consumption. This can be relieved by using copy-on-write of objects shared between multiple state versions. Seaside [25] claims to have a possible solution for copy-on-write in Smalltalk that requires registering all objects that have to be versioned. When processing web request, the value of versioned objects is restored to the needed state, while their identity remains the same, consequently all object references remain intact. Extraneous references to versioned objects are weak in order to support garbage collection. It remains an open question for us whether such versioned, garbage collected, expiring data structure could be implemented in Java in an efficient and unobtrusive way, but we think that solution would benefit a lot from using AspectJ [28] in order to weave versioning logic into objects.

Finally, we find that supporting browser's navigation commands in rich user interface applications leads to clumsy user interface, increased demands on server-side memory and CPU and/or more complicated programming model. Also, Ajax [1] technology integrates badly with this approach because all Ajax-based state transitions are not recorded in browser's navigation history. Instead, similar capability should be built into application logic in form of commands like cancel and undo.

8.3 Integration and Portals

It is important to note that so far we have described only applications that are configured before deployment and work within one Java virtual machine (or homogeneous cluster). There are portal applications that would benefit from dynamic reconfiguration and using widgets or flows deployed to another environment. The latter could happen for multiple reasons such as using a different platform (like .NET), co-location of web application with database, or just administrative reasons.

One possible approach is to integrate with Portlet [15] specification together with remote integration protocol WSRP [12]. Unfortunately portlets cannot be composed into hierarchies and have many limitations on how they can communicate with each other. There is also no notion of nested process in portlets. Finally, portal implementations that we are aware of allow reconfiguring portals only by redeployment.

It should be easy to assemble out of Aranea components a portal application that would contain multiple pre-packaged applications, communicating with each other, but the configuration would have to be read on deployment. One further direction is to integrate Aranea with some component framework allowing dynamic reconfiguration, such as OSGi [14]. This creates interesting problems of interference between parts of application being in different class-loaders, serialization, reentrancy, and cluster replication. Also, dynamic changes would have to be broadcasted to all initiated user sessions to reconfigure application.

Another related direction is to develop a remote integration protocol that would allow creating a widget that would be a proxy to a widget located in another environment. One important issue would be minimizing the number of roundtrips.

8.4 Fat Client

Lately, more and more web applications started using asynchronous requests to update parts of the page without resub-

mitting and refreshing the whole page. Some applications even implement most of UI logic on the client-side and use web server essentially as a container for the business layer. The enabling technology is called Ajax [1] and is essentially a small API that allows sending web requests to the server. We think that this trend will continue and in future most application will use this approach to a varying extent. We see the following scenarios for using Ajax technology.

The first option is when UI logic is still implemented on the server-side, but in order to make web pages more responsive sometimes ad-hoc asynchronous requests are used to update page structure without refreshing the whole page. In this case server-side controller framework must support occasional asynchronous requests to stateful widgets which can be accomplished in Aranea using either messages or the fact that widgets extend services and consequently have `action(input,output)` method. Within widget, some kind of simple event handling logic could be implemented. The most of the burden lies on the presentation framework, but this is out of scope of this paper.

An interesting alternative would be to have a new protocol that would extend HTTP and XHTML. It would allow representing the hierarchical structure and state of UI as an XML, where with each tag one could associate custom client-side component. The main difference from HTTP and XHTML would be that only changes from previous UI structure would be sent from server to the client and from client to server when user fires an event. This could be used to optimize request processing, communication, and rendering performance. Although there is nothing asynchronious in this approach, ad-hoc asynchronous requests described in the previous paragraph can be viewed as a special case of this model. If such protocol existed Aranea could be extended to work with it in a straightforward manner.

Another option is when all UI implemented on the client side within browser and server-side controller acts essentially as a business layer. Although business layer is often stateless, we find that Aranea could be used to create a coarse-grained server-side representation of UI state, essentially representing activated use-cases, modeled most naturally as flows. Client-side UI would we able to only execute commands making sense in the context of current server-side UI state. Such approach is very convenient for enforcing complex stateful authorization rules and data validation would have to be performed on the server-side in any case.

9. RELATED WORK

As it was mentioned before, Aranea draws its ideas from multiple frameworks such as Struts [3], WebWork [23], JavaServer Faces [10], ASP.NET [4], Wicket [24], Tapestry [7], WASH [30], Cocoon [2], Seaside [25], Spring Web Flow [21], and RIFE [17]. When possible we have referenced the original source of idea at the moment of introducing it.

Although we were not aware of Seaside [25] when developing this framework, we have to acknowledge that rich UI programming interface of widgets and flows is almost identical with programming interface of Seaside, but the design of Seaside differs a lot and it is not intended as a component model for web framework construction and integration. Also, from private communication we became aware that Seaside does not currently support blocking for any event other than completion of a flow, but this is not a conceptual, but rather implementation issue. Finally, Aranea is written in Java, Seaside in Smalltalk.

10. ACKNOWLEDGEMENTS

Development of Aranea implementation has been supported by Webmedia, Ltd. We are grateful to Maksim Boiko, who prototyped the implementation in his bachelor thesis and Konstantin Tretyakov, who provided valuable input as well as help with developing the presentation module. This work was partially supported by Estonian Science Foundation grant No. 6713.

11. SUMMARY

In this paper we have motivated and described a component model for assembling web controller frameworks. We see it as a platform for framework development, integration, and research.

There exists an open source implementation of Aranea framework available at http://araneaframework.org/. It is bundled together with reusable controls, such as input forms and data lists, and advanced JSP-based rendering engine. This framework has been used in real projects and we find it ready for production use.

We urge readers of this paper to visit address and give it a try. We realize that current component model will probably have to be adapted for new domains and approaches. We urge readers to validate current framework design, especially core interfaces against Your needs and ideas. Your proposals are very welcome.

12. REFERENCES

[1] Ajax. Wikipedia encyclopedia article available at http://en.wikipedia.org/wiki/AJAX.

[2] Apache Cocoon project. Available at http://cocoon.apache.org/.

[3] Apache Struts project. Available at http://struts.apache.org/.

[4] ASP.NET. Available at http://asp.net/.

[5] Cascading Style Sheets. Available at http://www.w3.org/Style/CSS/.

[6] Hypertext Transfer Protocol. Available at http://www.w3.org/Protocols/.

[7] Jakarta Tapestry. Available at http://jakarta.apache.org/tapestry/.

[8] Java Servlet 2.4 Specification (JSR-000154). Available at http://www.jcp.org/aboutJava/communityprocess/final/jsr154/index.html.

[9] The Javaflow component, Jakarta Commons project. Available at http://jakarta.apache.org/commons/sandbox/javaflow/index.html.

[10] JavaServer Faces technology. Available at http://java.sun.com/javaee/javaserverfaces/.

[11] JavaServer Pages Technology. Available at http://java.sun.com/products/jsp/.

[12] OASIS Web Services for Remote Portlets. Available at www.oasis-open.org/committees/wsrp/.

[13] Open source web frameworks in Java. Available at http://java-source.net/open-source/web-frameworks.

[14] OSGi Service Platform. Available at http://www.osgi.org/.

[15] Portlet Specification (JSR-000168). Available at http://www.jcp.org/aboutJava/communityprocess/final/jsr168/.

[16] RFC 2109 - HTTP State Management Mechanism. Available at http://www.faqs.org/rfcs/rfc2109.html.

[17] RIFE. Available at http://rifers.org/.

[18] RIFE/Crud. Available at http://rifers.org/wiki/display/rifecrud/.

[19] Ruby on Rails. Available at http://www.rubyonrails.org/.

[20] Spring. Available at http://springframework.org.

[21] Spring Web Flow. Available at http://opensource.atlassian.com/confluence/spring/display/WEBFLOW/.

[22] The Extensible HyperText Markup Language. Available at http://www.w3.org/TR/xhtml1/.

[23] WebWork, OpenSymphony project. Available at http://struts.apache.org/.

[24] Wicket. Available at http://wicket.sourceforge.net/.

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

[26] 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.

[27] R. Hieb, K. Dybvig, and C. W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110, 1994.

[28] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001. Project web site: http://www.eclipse.org/aspectj/.

[29] 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.

[30] 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/.