Improving User Experience by Infusing Web Technologies into Desktops

Jonathan Bardin Philippe Lalanda
Laboratoire d'Informatique de Grenoble &
Université de Grenoble
Grenoble, France
{firstname.lastname}@imag.fr

Clément Escoffier Alice Murphy
akquinet AG
Berlin, Germany
{firstname.lastname}@akquinet.de

Abstract

Modern applications are able to adapt their architecture dynamically in order to tackle requirements, correctives and context changes. Such dynamism is often an echo of complexity and is not well supported by traditional client and user software stacks making complex the design, implementation and maintenance of the end user interface. Meanwhile, the web has seen the emergence of user interface technologies (e.g. HTML5, CSS3, JavaScript) widely adopted by developers to create highly flexible user interfaces. However, such clients are intrinsically bound to run on a web browser which is out of the control of the application.

In this paper, we present ChameRIA, an application framework where a browser engine is reified as a component within the framework, thus allowing for better control over the rendering engine. We describe how we preserve a clear separation of concerns between the user interface and the application logic while maintaining coherence between them. We discuss how ChameRIA has been successfully used in two projects: a DRM document reader and a valve control application.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Languages,Patterns

General Terms Design, Experimentation, Reliability

Keywords Runtime software evolution, components, service, OSGi, JavaScript

1. Introduction

Nowadays, application availability is critical, not just in high-risk spheres like the financial and aerospace industries, but also in everyday applications such as operating systems and Internet services. Indeed, as already noted by Oreizy at the end of the 90s [25]:

> "Continuous availability is a critical requirement for an important class of software systems."

However, software must also adapt and evolve over its lifetime. Software engineering research has highlighted the importance of taking these evolutions into account in the software life-cycle. Although these types of changes are familiar to developers and architects, it is also increasingly necessary to accommodate them later in the life-cycle. It is therefore imperative to emphasize approaches that allow us to address the issues while minimizing the downtime of the application in other words, to be able to apply changes during execution [20][5].

In prior work, we and others have proposed service-oriented component approach to supporting application changes during execution [24][6][19][9]. Such approach allows us to designed and developed extensible applications capable of change during execution [4]. Unfortunately, this extremely dynamic resource availability is not well-supported by traditional client and user interface software stacks. Desktop user interface technologies such as Swing, SWT and QT all have severe limitations. Swing and QT pose too many difficulties in terms of threading, are not modular, and are not easily adaptable. SWT, while modular, requires manual object deallocation which is delicate for the developer and can hurt extensibility [12].

Despite the usefulness of those user interface technologies, such concerns make their cohabitation with the application logic in a same dynamic framework deeply critical. Indeed, dynamic frameworks (e.g. OSGi) are based on dynamic loading/linking language capabilities. Applications evolves at run-time thanks to the dynamic loading and unloading of modules. If the complexity associated with the dynamic loading of modules is hidden from developers thanks to the framework, the intrinsic problematic persists.

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.

Each modification within the application logic potentially impacts the user interface and reciprocally. This underlying behavior implies that each unintended behavior within the user interface modules can result in memory leaks and incoherent states in the long-run as well as the crash of the whole application. In OSGi those problematics are linked to the manipulation of distinct java ClassLoader for each module of the application, manipulation which can result in linking errors during the execution and in the apparition of dangling objects within the framework [13] (i.e. Stale Reference [1] in OSGi). This question of isolation make the usage of classic desktop user interface technologies potentially harmful for dynamic applications.

The web, meanwhile, has seen the emergence of user interface technologies such as HTML5, CSS3 and JavaScript. These have been widely adopted by developers to create highly flexible user interfaces. The recent adoption of the web-socket makes it possible to push asynchronous changes to the client. Furthermore, recent browser evolution allows for the isolation of web programs instances from each other, thus improving stability and performance [30][11]. Nevertheless, such clients are intrinsically bounded to run on a browser engine which is out of the control of the application. This restriction could lead to a lack of reliability and convenience (no file association, device access) for the user [2][18].

In this work, our goal is to take the best in terms of dynamism of both world, the OSGi platform and the browser client technologies. That is, we are able to provide an homogeneous framework for the design and runtime management of rich client applications. We thus propose ChamRIA an application framework which target the development of dynamic application and their user interface. We present a new software stack and its architecture were the user interface is fully developed in web technologies and run on an rendering engine while the application logic is fully developed in java and run on an OSGi platform, thus isolated from the user interface runtime. To better control the browser engine, we present how we have integrated the browser within the framework thus allowing to easily create and configure a browser window following a factory pattern. We discussed how using a service-oriented component programming model in both the development of the web user interface and the application logic helps to tackled down the problematic induced by the dynamism of the application and the runtime heterogeneity.

The rest of this paper is organized as follows. The next section gives a brief background on OSGI. We then present the ChameRIA approach and show how it allows for the creation of dynamic applications (from the application logic to the user interface) in section 3. In section 4, we describe the framework implementations and how to create an application on top of this framework. The section 5 describes how a ChameRIA application is able to evolve at runtime. We introduce two projects: a DRM document reader and a valve control application which have been developed on top of ChamRIA in section 6. In section 7, we consider the limitations and dissonances of our approach, along with different application domain. Finally, we discuss related work in section 8 and conclude in section 9.

2. OSGi

OSGi (formerly known as Open Services Gateway Initiative) is both a component-based platform and a service platform for the Java programming language. OSGi aims to facilitate the modularization of Java applications as well as their interoperability. The OSGi framework can be divided into three distinct layers:

Module layer: This layer allows for the division of a Java application into modules. An OSGi unit of modularization is called a *bundle*. Bundles address some of the weaknesses of Java's deployment model; a bundle extends the Java Archive files (JAR) with a special *manifest*. The manifest is used to specify static information about the bundle, such as the Java packages shared or required by the bundles.The module layer needs those information in order to install correctly and activate a bundle within the framework. In OSGi, bundles are the only entities for deploying Java-based applications.

Life cycle layer: The life cycle layer defines a runtime model for bundles. This model specify how bundles are started, stopped, installed, updated and uninstalled. Fundamentally, the life-cyle layer provides an API to control the bundles. This API covers the installation, starting, stopping, updating, uninstalation as well as the monitoring of bundles. This layer depends on the module layer.

Service layer: The service layer is the heart of dynamism in OSGi. It allows for the creation of component depending on services rather that other components. This enable a concise model built on interface based programming. Bundles developers are able to bind to services only using their interface specification. Services are not a component model but enable a true component model by reifing the coupling between components [17].

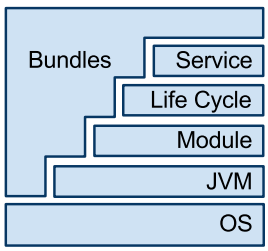

Figure 1. OSGi layers.

3. ChameRIA

3.1 Motivation

The OSGi Service layer allows for a smooth functional coupling between bundles. Unfortunately, in practice, bundles are tightly coupled via their packages. This coupling can result in complex problems such as dangling objects and linkage exceptions during the execution, especially when bundles are installed, updated or uninstalled. One common way to address this problem in OSGi is to extract interface packages in specific bundles containing only the application service interface package. In that way consumer and producer bundles are not coupled together via their interface packages since it is contained in an independent bundle. Such bundles can thus be updated independently from one and other, the dynamism being handled by the Service layer. However, such a process is not always possible, and in practice most of bundles don't even use the service layer (figure 2). Developing a rich client interface in OSGi means wrapping existing java UI libraries into one or several bundles. This operation is laborious, especially if we want to introduce the notion of service into such library in order to use the OSGi Service layer, that is to say, to be able to update the application at runtime.

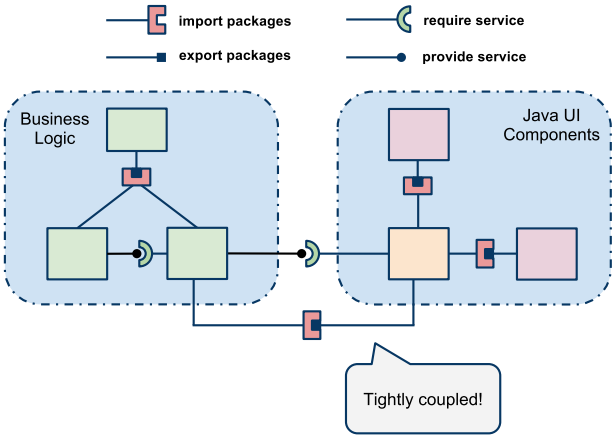

Figure 2. Rich client application.

In order to address this problem and thus to be able to create rich dynamic client applications we propose to use web technologies as a means to develop the dynamic application User Interface (UI). In contrast to classic Java UI technologies, web technologies can easily handle the notion of consuming a service. Furthermore, running UI components in a browser engine allows to fully isolate them from the application logic components, thus avoiding packages coupling between them (figure 3).

3.2 Concepts

The key concept behind ChameRIA is to merge web technologies into the OSGi world (figure 4). ChameRIA is both

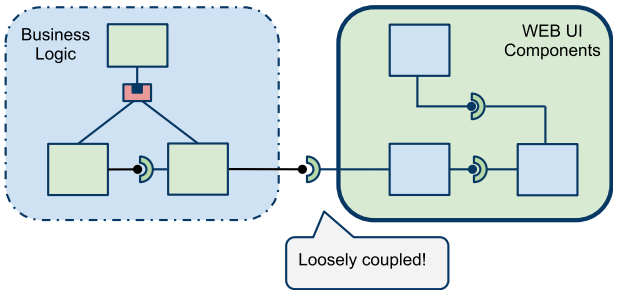

Figure 3. ChameRIA rich client application.

a service-oriented component framework and a web browser (engine supplied by WebKit [35]). The seamless integration of the web browser within the modular platform allows the application's user interface to be built as a web application, while the business part is done in Java.

Furthermore, ChameRIA follows the service-oriented component principles [6]:

1. A service is a provided functionality.
2. A service is characterized by a service contract (i.e.service specification). The contract describes some combination of the service's syntax, behavior, semantics and dependencies on other services.
3. A component implements a contract. In this way, a component provides a service.
4. The service-oriented interaction pattern is used to resolve service dependencies at runtime.
5. Compositions are described in terms of contracts.
6. Contracts are the basis for substitutability. A component can be substituted for another if it implements the same contract.

By following these rules, ChameRIA enables strong support for dynamic availability of services. Indeed, substitutability is encouraged since the compositions are described in terms of specification rather than a specific implementation. Another important principle is the service-oriented interaction pattern which enables the resolution of all service dependencies at runtime.

In conformance with these principles, the WebKit browser component is reified within the framework as a component (figure 4). This component allows the browser view to be created following the factory pattern [28]. It is fundamental to note that while this service provides the functionalities to manipulate the appearance and the behavior of the browser itself, it does not actually construct the web pages. Indeed, the main goal is to use the afore-mentioned web technologies (such as HTML5, CSS3 and JavaScript) in order to develop the user interface.

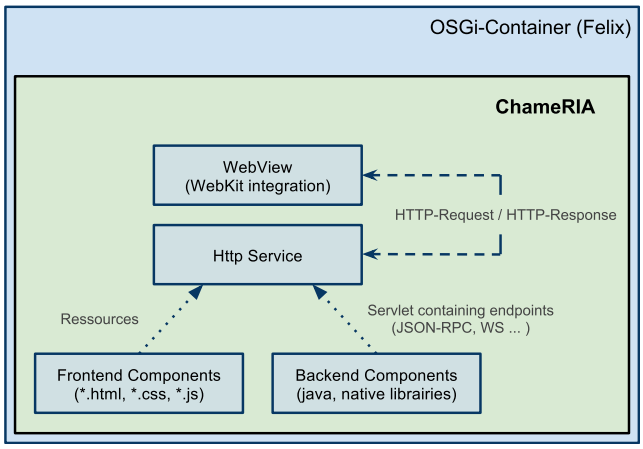

Figure 4. ChameRIA architecture overview.

While the integration of the browser engine within the framework is essential, a clear separation between the Application logic and the interface (web client) allows for a simpler and more efficient development model. Such separation of concerns is achieved by following the client-server style [3] (figure 5). Indeed, by moving all of the user interface functionality into the client, the application logic and the user interface can evolve independently.

This separation can be defined with three simple principles:

1. All communication between the server and client is contract-based and done through the use of virtual objects (e.g. JSON-RPC proxies, REST virtual resources).
2. No part of the web client should be evoked, generated or templated from the server-side. This rules out in-line conditional HTML in JSP, ASP, PHP etc.
3. The server will only implement the business logic.

In order to respect these principles, communications between the application logic (server) and the web user interface (client) are assured by service remote proxies (figure 5). Basically, each service of the server which is required to be invoked from the client is automatically exported through an endpoint. The client is notified and a proxy providing the service contract within the client is dynamically created and made available. Thanks to the service-oriented pattern which enables the resolution of all service dependencies at runtime, the coherence between the web user interface and the application logic is maintained. Indeed, in this way, the proxy service availability within the client is mapped on the service availability within the server. This runtime adaptation can be summed up in five steps:

1. The framework is notified of the availability of a service within the server.
2. If the service must be available within the client an endpoint is created.
3. The client is notified of the endpoint availability through the publication of the contract.
4. A proxy is created which provides the service contract and delegates to the endpoint.
5. If the service is no longer available, the client is notified, the proxy is destroyed along with the endpoint. Thus, the coherence between the server and the client is maintained.

4. Drilling Down

4.1 Browser Integration

The ChameRIA browser is based on the WebKit project. The integration of the browser consist of two modules: the ChameRIA launcher and the *WebViewFactory* component.

First, the Chameria launcher is a wrapper around the webkit gui and the osgi framework. It allows to start the browser engine and the osgi framework as a monolithic application. Secondly, the *WebViewFactory*; each instance of this component receives a configuration, create a browser windows and provides a *BrowserService* through which we can change several property of the web view. Those properties include the window: URL, size, full-screen mode, context menu and the menu bar.

A simple configuration of a *WebView* is the following:

```
#url to load
url = http://localhost:8080/web/index.html

#menu bar
menu.bar = true

#size
resizable = true
fullscreen = false
scrollbar.horizontal=ScrollBarAsNeeded
scrollbar.vertical=ScrollBarAsNeeded

#features
download=false
print=false

# no context menu
# to change before releasing
context.menu=true
inspector=true

#disable local storage
storage=false

application.name=chameria

width= 1200
```

WebView instance configuration.

4.2 The Communication Machine

Both, the application logic and the web user interface are designed and implemented on top of a service-oriented component framework. This particularity allows for a smooth functional cohesion between server and client components. However, both framework are running on two separate run-time machine. Additionally, the application logic is implemented

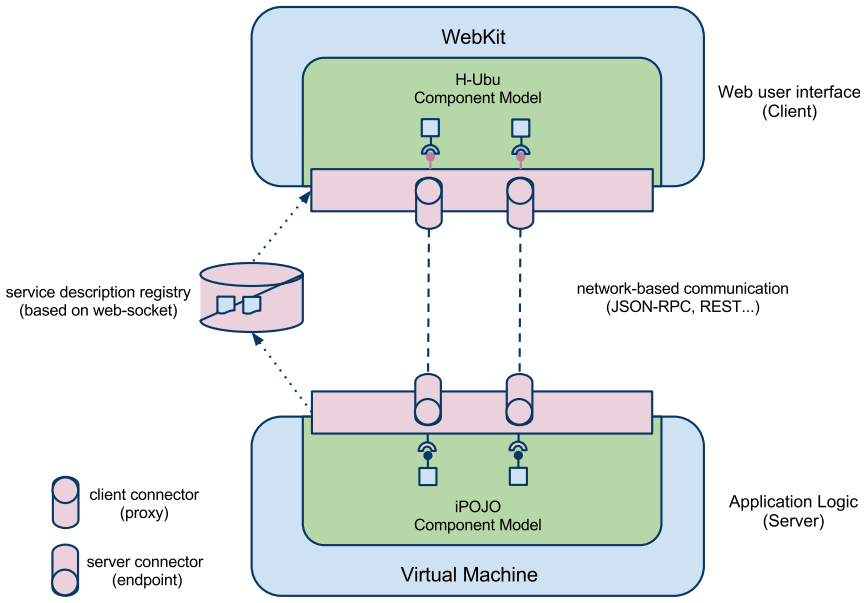

Figure 5. ChameRIA detailed architecture.

in Java while the logic part of the web user interface is implemented in JavaScript.

We addressed those problems via the automatic creation of virtual objects (i.e. proxies). In order to make available a service from the server within the client we define a filter. Each service matching this filter are tracked during the execution. When such service is available on the server, an endpoint is created. Once the endpoint has been successfully created, the contract is published through a network registry. Eventually, the client is notified of the endpoint availability; then a proxy which provides the same contract and delegates to the endpoint is instantiated within the client.

This process is hidden from the applications developers. Programmers are able to access to a remote service in the same way that a local one. The distributed nature of a service is transparent. This notion of transparency has been deeply encouraged in loosely coupled distributed systems [33] [15]. However, such a centralized view of the system must not be the only tool available for programmers. Actually, problems of centralized systems are subsumed by those inherent in distributed systems; it would be illusory and counterproductive to handle a distributed systems exclusively as a centralized system. Especially for aspects such as performance and fault tolerance [36] [8].

We avoid 6 of the 8 fallacies of distributed computing by using remote communication between two virtual machines which are actually running on the same physical machine (i.e. the desktop). This singularity means that we can legitimately assume that: the network is reliable, latency is zero, the network is secure, bandwidth is infinite, transport cost is zero and the network is homogeneous. The fact that the topology does not change is questionable but can be modelled by service availability. The last common assumption is that there is only one administrator. In our approach, both the web user interface and the application logic can be administrated separately.

In order to ease the administration of the services which are distributed, we have introduce introspection mechanisms and a declarative way to configure the distributed aspect of the application. A specific API enables to know at runtime if a service have an endpoint or if it is provided through a proxy, as well as the endpoint metadata such as the underlying communication protocol and the endpoint address.

The following configuration file specifies that an endpoint must be created for each instance providing an *HelloService* service, the endpoint must be invokable using JSON-RPC protocol.

```
{
 "machine" : {
  "id" : "server",
  "host" : "localhost",

  "connection" : [
   "out" : {
    "service_filter" : "(objectClass=acme.HelloService)",
    "protocol" : [ "jsonrpc" ]
   }
  ]
 }
}
```

Communication configuration.

The configuration example shows that the application logic machine (i.e. OSGi & iPOJO, discussed in the next section) provides HelloServices service through the JSON-

RPC protocol. The configuration is interpreted at runtime, that is to say that changing the configuration during the execution comes to modify this configuration file. Basically, both runtimes are modelled as meta-component sharing the same network registry. We discuss in the section 6 of how we can use our communication framework in order to construct network based applications and how it is possible to address the remaining fallacies of distributed computing, as well as limitations of this approach.

This program show how to use the *HelloService* which runs on the server from the client.

```
..
var jsonrpc = new JSONRpcClient("/JSON-RPC");
var greeting = jsonrpc.helloService.hello("Dave");
$('#greeting').text(greeting);
..
```

HelloService client proxy.

4.3 Building the Application Logic

ChameRIA has been implemented on top of iPOJO. iPOJO is a service-oriented component runtime which eases the development of applications on top of OSGi [1].One main concern with OSGi is that the Service model leaves service dependency management as a manual task for component developers. iPOJO uses inversion of control in order to wrap your plain old Java object within a container. The container handles all service-oriented interactions (e.g. service publication, service object creation, service requirement and selection).

Developing an iPOJO component is as straightforward as creating a Java class. POJO classes are manipulated (via byte-code injection) to become iPOJO component. This bytecode injection process is done at compilation time and allows to easily create a bundle containing an iPOJO component and its specification.Thus, bundles are used as a deployment unit and, thanks to the OSGi module layer, as a mean to resolve packages dependencies. Once a bundle has been deployed and started, iPOJO handle the service dependency management, linking component together thanks to the component specification which has been generated at compilation time and which contains its provided and required services. Additionally, iPOJO provides extension mechanisms known as *handlers* which allows a container to support several non-functional aspects such as security, persistence, scheduling and so on; each handler managing one non-functional property.

The following class realizes a simple iPOJO component which implements the Hello contract.

```
@Component(name="acme.hello.component")
@Instanciate(name="helloService") //default instance
@Provides //Provide HelloService
public class MyComponent implements HelloService{

  @Requires(optional=true) //require a Log Service
  private LogService logger;

  public String hello(String name){
    return "Hello "+name+" !";
  }

  @Validate //on validation callback
  private void start(){
    logger.log(INFO,"HelloService started.");
  }

  @InValidate //on invalidation callback
  private void stop(){
    logger.log(INFO,"HelloService stopped");
  }
}
```

A simple iPOJO component

Annotations can be replaced by a special file called *metadata.xml*, both are parsed during compilation time in order to manipulate the class and to add the iPOJO component specification to the bundle manifest. The component specification contains its service requirements (*@Requires*), capabilities(*@Provides*) and life-cycle callback (*@Validate*,*@InValidate*).

4.4 Building the Web User Interface

Recently, JavaScript has become mainstream and is used by the majority of web-app stacks. A number of frameworks and libraries such as DOJO and JQuery have been proposed in order to simplify coding. However, such libraries do not help with structuring the code. With the growth of complexity of our user interface, we identified a stringent requirement for a component model structuring our JavaScript code.

We defined a component model named H-Ubu, supporting:

1. component definition : the code is organized into components
2. component injections : components can be injected in other components
3. contract-based interactions : components implement and use contracts (i.e. interfaces)
4. synchronous and asynchronous component communication : components can interact using method calls and events
5. architecture description : the application is composed by registering components and binding them
6. component configuration : components can be configured
7. test : components are testable !
8. component separations: components are developed in separate files, the application code is more scalable.

To start with an example, the following programme shows how to compose an application with H-Ubu:

```
<script src="http://../jquery-latest.js"></script>
<script src="hubu.js"></script>
<script src="backendComponent.js"></script>
<script src="frontendComponent.js"></script>

<!-- The contracts -->
```

```
<script src="UserServiceContract.js"></script>

<script type="text/javascript">

  $(document).ready(function(){
    // Component registrations
    hub
    .registerComponent(backendComponent(), {
      component_name: 'user'
    })
    .registerComponent(frontendComponent(), {
      loginId : '#login',
      logoutId : '#logout',
      statusId : '#status',
      component_name: 'frontend'
    })
    // Declare a binding
    .bind({
        component: 'user',
        to:'frontend',
        into: 'bind', // method called on 'to'
        contract: UserServiceContract // The interface
    })
    // Start the app
    .start();
  });

</script>
```

A simple composition in H-Ubu

The first thing is to import the H-Ubu library, and other dependencies (e.g JQuery), then the component, and finally the contract. In our example, we assume that the contract is implemented by the back-end component. When the page is loaded, we register our components on a special H-Ubu object : hub. This is the *nerve* of the application. As you can see, components receive configurations. The front-end component also receives the HTML *ids* to manipulate the page indirectly. The bind method allows injection of the back-end component into the front-end component. Notice that the binding specifies the contract to use for this interaction. Finally, we start the hub, i.e. the application.

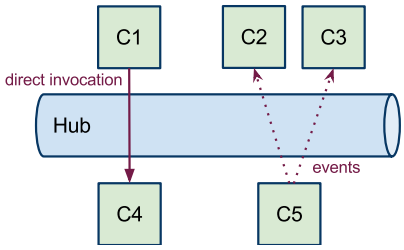

Figure 6. H-Ubu concepts.

In order to improve the modularity we proposed that each component must be define in one JavaScript file. Our web user interface can thus be realized as an assemblage of components. It also becomes easier to follow an MVC pattern [29]. *Controllers* component can easily be bind with *Models* component in order to manipulate the *Views* (HTML DOM). In our approach, Model components are the proxies of some applications logic services. An other aspect of H-Ubu component is the notion of scope. Components supports private and public members; the contracts making obvious the public member of there code. Once a component has been bind to another component, it can invoke the injected component only through its contract. Actually, H-Ubu injects a proxy in order to ensure that only the contract methods are used. Component also support testing. Each component can be tested separately but it is also possible to perform integration test by assembling the component which are required to be tested.

4.5 Web Code Integration

The *HttpService* which is part of the OSGi specification allows bundles in the OSGi framework to register resources to be accessed via HTTP. Practically, the web resources (JavaScript, HTML, PNG, CSS files and so on) are included in the bundle resource. Then the developer must implement a *BundleActivator* as a mean to get the *HttpService* in order to register the resources on the embedded web server providing the *HttpService*. This approach caused us two complication during the development of our applications. The first one was that the graphic artist and web developer in charge of creating the web user interfaces had no experienced with OSGi. Ideally, they should not need deep knowledge of OSGi, particularly not the graphic designer. The second one is that it is deeply annoying to have to repack and then update the bundle containing the web code in order to test the result between each modification.

These two inconvenience led us to the creation of a specific component called: *WebExposer*. This simple component use the *HttpService* in order to publish the content of a local folder. Each resource request on the HTTP server are delegated to the file system. Thanks to that component, web developers are able to test their applications without having to handle OSGi and just have to refresh their browser in order to access to the last version.

Once the web user interface has been fully developed and tested by the web developers and graphic artists, it can be packaged in several bundles (e.g. one bundle by H-Ubu components plus one containing styles and images). Thus allowing to manage the Web User Interface (WUI) components as OSGi bundles during the execution.

5. Coping With Runtime Change

Because we use bundles as deployment units, most runtime change are triggered by adding, removing or updating application bundles. Actually, the OSGi module layer (see 2) provides an API through which we can add, remove or update bundles at runtime. Thanks to these facilities we are able to add new functionalities, remove obsolete ones or update parts of the application during the execution without impacting the availability of the whole application. For example, updating a bundle containing a WUI component will not impact the business logic of the application or even other parts of the WUI. However, this is fully true if and only if bundles

are loosely coupled. Indeed, if two bundles are tightly coupled together, updating one of them will require restarting the other; this restart is triggered by refreshing the bundles packages. The process of uninstalling and updating bundles along with the need to refresh packages when bundles are tightly coupled is detailed in [31].

5.1 Changing ChameRIA Bundles at Runtime

A ChameRIA application is made of four different types of bundles:

- **BC** bundles. These bundles contain the iPOJO components (i.e. the business logic components described in section 4.3).
- **SC** bundles. These bundles act as service contracts containers, in OSGI a service contract is a Java interface class.
- **WC** bundles. These bundles contain the H-Ubu components (i.e. the web user interface components described in section 4.4).
- **WV** bundles. These bundles contain web view resources (i.e. HTML, CSS, Images). These resources are manipulated by the H-Ubu components.

Figure 7 shows how these types of bundles are coupled together. BC bundles can be loosely coupled together through service dependencies. Similarly, WC bundles can also be coupled among themselves, but distinctly, they can also be coupled to BC bundles. Tight coupling occurs with bundles providing the service specification (SC). Indeed the bundles which contain a service implementation (i.e. BC & WC) need to be updated along with the service specification if it changes. Finally the WC bundles can also depend on the content of Web view bundles (e.g HTML DOM objects reference).

In regards to the level of coupling, the worst runtime change is a modification to service specifications. Such modifications can impact both the application logic and the web user interface through respectively BC and WC bundles. With the exception of this particular case, both the application logic and the web user interface can evolve independently from the rest of the application.

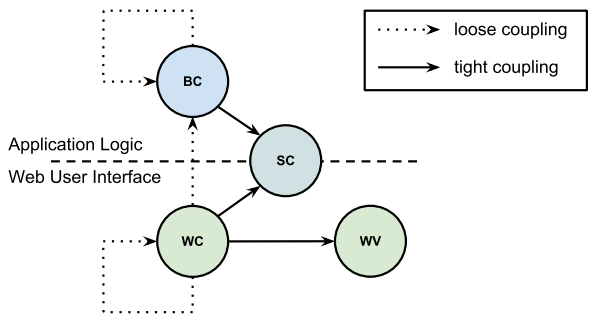

Figure 7. Coupling among ChameRIA application bundles.

6. Experience

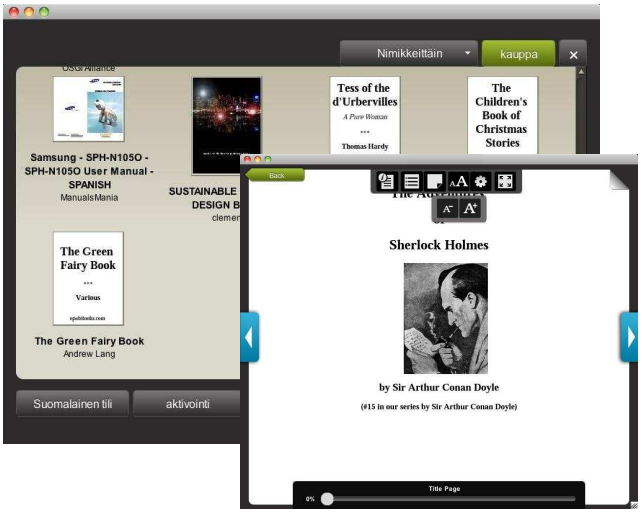

Figure 8. DRM document reader user interface.

ChameRIA has been successfully used in several projects. Two of them are presented in this paper. The first was the design and implementation of a Digital Rights Management(DRM) document reader targeting net-books and tablet PCs (figure 8). The complexity of the user interface in addition to the constraints raised by the use of a DRM native library and the need for file association was addressed by ChameRIA. Indeed, control over the WebKit within the application logic allowed us to handle file association and DRM management. Furthermore, the complexity of the user interface was tackled by the use of web interface technologies which were separated from the application logic.

The second project was a Valve control application (figure 9). In this application, the back-end was required to interact with the serial bus in order to control the valve. The front-end was essentially an administration interface. One requirement was the generation and display of graphs.

7. Discussion

The main constraint introduced by the ChameRIA approach is the requirement to follow a service-oriented component model both in the back-end and front-end parts of the application. This constraint is fundamental to our approach. It forces developers to consider the dynamic nature of services (availability change at runtime) and hence to construct applications which are intrinsically resilient to change in the execution environment. Moreover, following this model does not impact the choice of the underlying language/runtime. We have shown with H-Ubu that the implementation

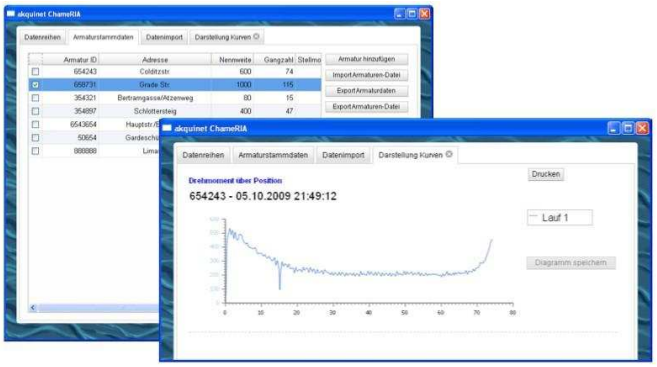

Figure 9. Valve control user interface.

of such a model was relatively straightforward (H-Ubu is about 1000 line of code), not to mention similar component models such as Scala components and COM+.

We have presented in section 4.2 the underlying communication framework of ChameRIA. This framework allows for transparent communication between the browser and the OSGi platform through the dynamic generation of endpoints and proxies. This framework can also be used in order to construct loosely coupled distributed applications. It is straightforward to run the web part of the application on a remote browser rather than the ChameRIA one. Furthermore it is also possible to use the framework as a means to communicate with several OSGi gateways or with remote applications through WebServices or following the REST architectural style.

Nevertheless, despite the fact that the service-oriented component model allows us to reflect change in the network through service availability, this is not sufficient as a means to conceive reliable distributed applications. One work around is to implement *distribution aware* components between business components and the distributed services. Those components handling distribution problematic such as latency or serialization but keeping advantage of being decoupled from the communication protocols and the network topology. Indeed, those concerns are handled by the communication framework. The main constraint imposed by the communication framework is to have at least one common network registry for the machines that need to communicate together.

8. Related Works

Dynamic software update: Several research projects from different domains address the challenges of runtime updates (e.g. [26] [21]). Dynamic software updating (DSU) techniques [14] focus on compiler based approaches to support runtime upgrades. The Ginseng project [23] is a DSU implementation for C that allows updating software during the execution, the update consists in applying a patch. It generates most of the patch and allows to apply it on-line thanks to runtime support which ensures type-safety and up to date data. The Jvolve project [32] is a DSU-enhanced Java VM, it supports method body updates along with class updates (adding, removing or changing the type of fields and methods). In contrast to these approaches, our system has been designed in order to upgrade individual components in a component-based system. This allows us to independently upgrade each component of the application and fits in with architecture-based approaches such as [16] [34] and [19] [24]. Hence, our work is closer to a runtime software adaptation framework as envisioned in [25].

Browser integration: Browser integration: Similarly to our approach, the SWT Browser widget [7] binds a native browser engine (e.g WebKit) allowing to add HTML viewing capabilities to an SWT application. Hence, developers are able to control the browser through the eclipse runtime. The eclipse runtime [10] is also built in top of the OSGi framework. The main distinction with our approach is that the eclipse runtime does not take advantage of the service layer as a means to support runtime change. This results in modular but tightly coupled applications which do not easily support runtime changes because each individual change can potentially bring down the whole application.

Service-oriented front-end architecture: Service-Oriented Front-End Architecture: The Thin Server Architecture (TSA) Working Group proposed an architectural style where all presentation layer logic is moved from the server to JavaScript logic on the client (i.e Web Browser) [27]. Thus allowing server-side components to be concerned only with business logic. One example of a framework following the TSA principles is the JavaScriptMVC [22] framework. Despite the fact that TSA has been designed for network based client server applications, the principles behind it seem to fit very well with the ChameRIA approach. We believe that the application logic model of ChameRIA along with its communication machine would allow to develop, fairly easily, server-side components as described in TSA.

9. Conclusion

The emergence of applications that have to cope with new requirements, changing context and corrective changes during execution raises new challenges. User interface developers of such applications have to deal with both traditional user interface technologies and modular frameworks (e.g OSGi) in order to build user interfaces. This raises two important problems: the first is that the use of traditional user interface technologies which have not been designed to deal with dynamic business logic is an important source of difficulties for reliability; the second problem is the requirement for user interface developer to be able to manipulate the underlying runtime framework technologies.

In this paper we have shown that following a service-oriented component model while using web technologies

makes possible the conception and execution of dynamic desktop applications where both the application logic and the user interface can evolve at runtime. The main advantages of our approach is that the user interface developers are not required to have knowledge about the business logic framework and thus can construct the whole user interface by manipulating only web technologies. We then present the ChameRIA framework which is an implementation of our approach along with two service oriented component frameworks which ease the conception and implementation of both the business logic and the web user interface. We presents two industrial projects were ChameRIA have been successfully used.

Availability

The ChameRIA source code is available for download at `https://github.com/akquinet/ChameRIA`. The H-Ubu component model for JavaScript is hosted on `http://akquinet.github.com/hubu`.

Acknowledgments

This work builds upon the OW2 Chameleon project, accessible at `http://chameleon.ow2.org`. We would like to specifically acknowledge our shepherd Allen Wirfs-Brock along with Walter Rudametkin, Issac Noe Garcia and Kiev Gama for helpful comments on this paper.

References

[1] O. Alliance. Osgi service platform core specification release 4, May 2007. `http://www.osgi.org/Specifications/HomePage`.

[2] C. Anderson and M. Wolff. The web is dead. long live the internet. Wired, August 2010. `http://www.wired.com/magazine/2010/08/ff_webrip`.

[3] G. R. Andrews. Paradigms for process interaction in distributed programs. *ACM Comput. Surv.*, 23:49–90, March 1991. ISSN 0360-0300.

[4] J. Bardin, P. Lalanda, and C. Escoffier. Towards an automatic integration of heterogeneous services and devices. *Asia-Pacific Conference on Services Computing*, 0:171–178, 2010.

[5] L. Baresi and C. Ghezzi. The disappearing boundary between development-time and run-time. In *FoSER'10*, pages 17–22, 2010.

[6] H. Cervantes and R. Hall. A Framework for Constructing Adaptive Component-based Applications: Concepts and Experiences. In *CBSE*, pages 130–137. Springer, May 2004. ISBN 3-540-21998-6.

[7] C. Cornu. Viewing html pages with swt browser widget, August 2004. `http://www.eclipse.org/articles/Article-SWT-browser-widget/browser.html`.

[8] P. Deutsch and J. Gosling. The eight fallacies of distributed computing. James Gosling: on the Java Road, 1997. `http://blogs.oracle.com/jag/resource/Fallacies.html`.

[9] C. Escoffier and R. Hall. Dynamically Adaptable Applications with iPOJO Service Components. In *Software Composition*, pages 113–128. Springer, March 2007. ISBN 978-3-540-77350-4.

[10] E. Foundation. Component oriented development and assembly (coda) with equinox, March 2008. `http://www.eclipse.org/eclipsert/whitepaper/20080310_equinox.pdf`.

[11] M. Foundation. Electrolysis. Mozilla Wiki, April 2011. `https://wiki.mozilla.org/Electrolysis`.

[12] W. Foundation. Standard widget toolkit, 2011. `http://en.wikipedia.org/wiki/Standard_Widget_Toolkit`.

[13] K. Gama and D. Donsez. A survey on approaches for addressing dependability attributes in the osgi service platform. *SIGSOFT Softw. Eng. Notes*, 35:1–8, May 2010. ISSN 0163-5948.

[14] M. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. *SIGPLAN Not.*, 36:13–23, May 2001. ISSN 0362-1340.

[15] ISO/IEC. Information technology - open distributed processing - reference model: Overview, December 1998. International Standard ISO/IEC 10746-1.

[16] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Trans. Softw. Eng.*, 11:424–436, April 1985. ISSN 0098-5589.

[17] P. Kriens. Scala components vs osgi. OSGi Alliance Blog, July 2010. `http://www.osgi.org/blog/2010/07/scala-components-vs-osgi.html`.

[18] G. Lawton. New ways to build rich internet applications. *Computer*, 41:10–12, August 2008. ISSN 0018-9162.

[19] M. Léger, T. Ledoux, and T. Coupaye. Reliable dynamic reconfigurations in the fractal component model. In *Proceedings of the 6th international workshop on Adaptive and reflective middleware.*, ARM '07, pages 3:1–3:6, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-931-9.

[20] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. Academic Press, 1985. ISBN 0-12-442440-6.

[21] K. Makris and R. A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 31–31, Berkeley, CA, USA, 2009. USENIX Association.

[22] J. B. Meyer and B. Moschel. Javascriptmvc 3.0, 2010. `http://javascriptmvc.com/`.

[23] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. *SIGPLAN Not.*, 41:72–83, June 2006. ISSN 0362-1340.

[24] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA*, 2005.

[25] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8368-6.

[26] A. Orso, A. Rao, and M. Harrold. A technique for dynamic updating of java software. *Software Maintenance, IEEE International Conference on*, 0:0649, 2002.

[27] G. Prasad, R. Taneja, and T. Vikrant. Life above the service tier, October 2007.

[28] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley Longman, 1st edition, 1995. ISBN 0201422948.

[29] T. Reenskaug and A. Goldberg. Models-views-controllers. Xerox PARC report, December 1979. `http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html`.

[30] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 219–232, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-482-9.

[31] J. S. Rellermeyer, M. Duller, and G. Alonso. Consistently applying updates to compositions of distributed osgi modules. In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, HotSWUp '08, pages 9:1–9:5, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-304-4.

[32] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a vm-centric approach. *SIGPLAN Not.*, 44: 1–12, June 2009. ISSN 0362-1340.

[33] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 0132392275.

[34] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., and J. E. Robbins. A component and message-based architectural style for gui software. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 295–304, New York, NY, USA, 1995. ACM. ISBN 0-89791-708-1.

[35] W. Team. The webkit open source project, 2011. `https://www.webkit.org`.

[36] J. Waldo, W. Geoff, A. Wollrath, and S. C. Kendall. A note on distributed computing. Sun Microsystems, Inc, 1994.