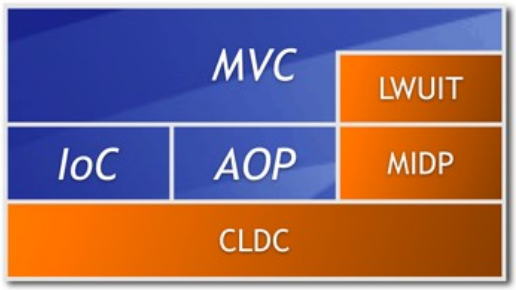
Signal Framework for Java ME

Introduction

Signal Framework is an open-source IoC, AOP and MVC framework for Java ME (J2ME) based on Spring. The framework has been designed to overcome the limitations of the CLDC API that prevent IoC containers implemented in Java SE from running on J2ME implementations. Signal Framework uses regular Spring XML configuration files, allowing developers to leverage existing tools and skill sets while coping with the limitations of J2ME.

The following diagram illustrates the architecture of the framework:



The name of the framework was inspired by the Antenna library (http://antenna.sourceforge.net/) and, obviously, Spring Framework.

The IoC container

The reflection support in the CLDC API is very limited compared to Java SE. The API only allows objects to be constructed with default (no-arg) constructors. It is not possible to pass arguments to constructors, invoke methods, access fields or create dynamic proxies.

To overcome those limitations the IoC framework reads context configuration files when an application is compiled and generates Java code responsible for instantiating a context at runtime. When a J2ME application is started, it executes the generated code instead of loading any configuration files. In effect an application context is created at runtime without relying on XML parsing or advanced reflection features. The footprint of the generated code is very small, because the IoC runtime only consists of approximately 10 classes. The generated code does *not* depend on any Spring libraries.

Even though the framework is geared towards the J2ME platform, the IoC container can be used in any Java applications that need to take advantage of Spring without relying on reflection or XML parsing (e.g. Android or GWT platforms).

Signal Framework supports the following features of the Spring IoC container:

- Spring XML configuration files
- Singleton beans
- Dependency injection through constructor arguments and properties
- Autowiring
- Lazy initialization
- Bean post-processors (`com.aurorasoftworks.signal.runtime.core.context.IBeanProcessor` - an equivalent of `org.springframework.beans.factory.config.BeanPostProcessor`)
- Lightweight AOP based on auto-generated proxy objects
- `com.aurorasoftworks.signal.runtime.core.context.IInitializingBean` - an equivalent of `org.springframework.beans.factory.InitializingBean`
- `com.aurorasoftworks.signal.runtime.core.context.IContextAware` - an equivalent of `org.springframework.beans.factory.BeanFactoryAware`

The code generator is normally invoked by a Maven plugin that requires two parameters: a name of a Spring configuration file and a name of an output Java class. The generator does support the `<import resource="…">` tag so it is possible to process multiple context configuration files with a single invocation of the plugin.

The following example demonstrates a context configuration file and a corresponding generated Java class:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
      <bean id="greeter"
      class="com.aurorasoftworks.signal.examples.context.core.Greeter">
            <constructor-arg ref="msgSource" />
      </bean>

      <bean id="msgSource"

class="com.aurorasoftworks.signal.examples.context.core.MessageSource">
            <constructor-arg>
                  <map>
```

```
                    <entry key="greeting" value="Hello!" />
                </map>
            </constructor-arg>
        </bean>
</beans>


public class ApplicationContext extends
    com.aurorasoftworks.signal.runtime.core.context.Context
{
    public ApplicationContext() throws Exception
    {
        /* Begin msgSource */
        ApplicationContext.this.registerBean("msgSource", new
com.aurorasoftworks.signal.examples.context.core.MessageSource(new
java.util.Hashtable(){{put("greeting", "Hello!"); }}));
        /* End msgSource */


        /* Begin greeter */
        ApplicationContext.this.registerBean("greeter", new
com.aurorasoftworks.signal.examples.context.core.Greeter(((com.aurorasoft
works.signal.examples.context.core.MessageSource)
GreeterContext.this.getBean("msgSource"))));
        /* End greeter */
    }
}
```

The framework does not currently support Ant, but Ant tasks could be easily implemented as thin wrappers for the existing generator.

The AOP framework

In addition to making IoC tricky, the limitations of the CLDC API prevent existing AOP frameworks from running on Java ME devices. The AOP Alliance API and popular AOP libraries like AspectJ and JBoss AOP depend on `java.lang.reflect.*` types that are not present in Java ME. Moreover, AOP implementations often rely on custom class loaders and/or dynamic proxies neither of which are supported by CLDC.

The AOP implementation provided by the Signal Framework is designed to work on Java ME devices: it has a relatively small footprint, it only relies on CLDC classes and it allocates as few temporary objects at runtime as possible. Those features come at a price, however: the framework is not as feature-rich as its desktop/enterprise counterparts and it only supports interception of interface

method calls.

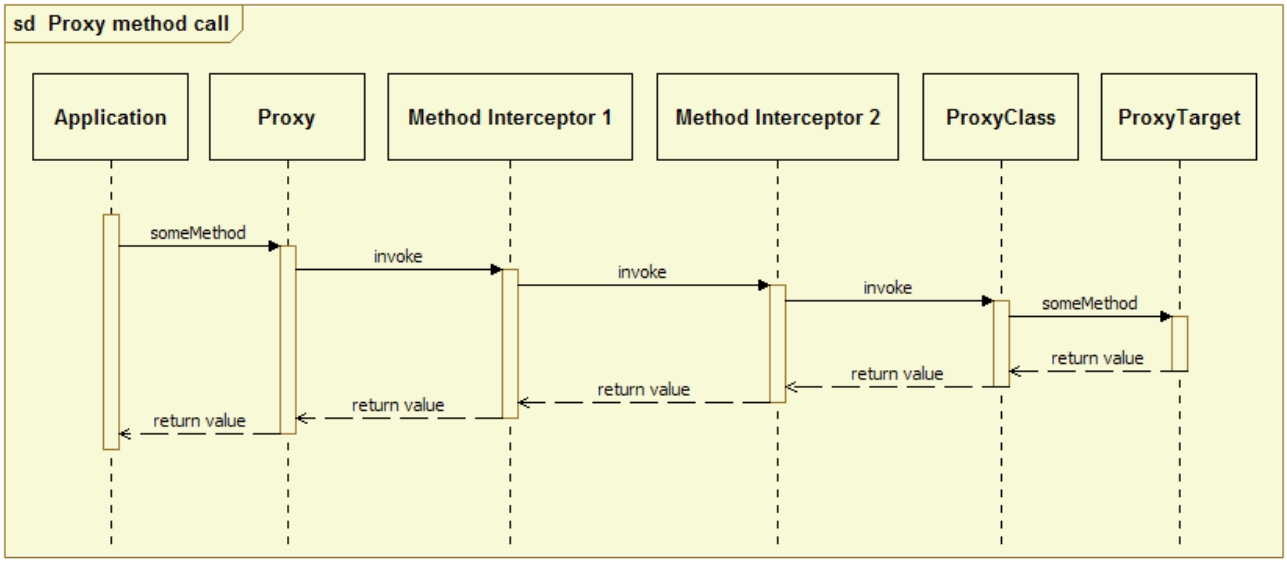
The AOP framework overcomes the limitation of Java ME by relying on code generation. When an application context is created at build time, the framework identifies bean classes that implement the `com.aurorasoftworks.signal.runtime.core.context.proxy.IProxyTarget` interface and generates proxies for them. Those proxies are in turn used to intercept calls and execute method interceptors.

This concept is similar to dynamic proxies supported by Java SE. Most classes included in the AOP framework have corresponding types in Java SE, as shown below:

Signal AOP type (com.aurorasoftworks.signal.runtime.core.*)	**Java SE type**
`context.proxy.ProxyFactory`	`java.lang.reflect.Proxy`
`context.proxy.IInvocationHandler`	`java.lang.reflect.InvocationHandler`
`context.proxy.IMethodInterceptor`	`org.aopalliance.intercept.MethodInterceptor`
`context.proxy.IProxy`	a return value of `java.lang.reflect.Proxy.newInstance`
`context.proxy.IProxyTarget`	any object
`context.proxy.IProxyClass`	`java.lang.Class`
`context.proxy.IMethodHandler`	`java.lang.reflect.Method`

Proxies are normally created by the `ProxyFactory` class. The most common way of creating a proxy is passing a list of interceptors to the `IProxyFactory#createProxy(IProxyTarget target, IMethodInterceptor [] interceptors)` method. An object returned by that method implements the same interfaces as the passed target object and can be safely cast to those interfaces.

The interception works in the following way:



Due to its simplicity the framework does have some limitations. Proxies created by the AOP framework reuse array instances when passing arguments to an invocation handler so that temporary objects do not need to be created. This however implies that that proxies need to be synchronized (this is handled by the code generator). In a multithreaded application this can lead to performance issues. The proxy code is synchronized on a proxy instance so multiple proxies of the same object can be used concurrently without blocking any threads.

When primitive objects are passed to or returned from a proxy method they need to be wrapped with object types like `java.lang.Integer` before being passed. This is the only scenario that requires allocation of temporary objects. Apart from wrapping primitives the AOP framework does not need to allocate any temporary objects and can be used safely regardless of the quality of a garbage collector.

The MVC framework

The MVC framework is based on IoC and AOP capabilities described in the previous sections. The framework supports both MIDP and LWUIT APIs and it is easy to add support for other view technologies if needed.

The framework does not impose any restrictions on the domain model or views, as long as either LWUIT or MIDP is used. Instead, it is designed to make implementation of controllers easy. The most important features implemented in the controller layer are lazy initialization of controllers (and corresponding views, if any) and declarative navigation rules defined in an IoC configuration file.

A controller is a simple bean defined in an IoC application context. All

controllers need to implement the `com.aurorasoftworks.signal.runtime.ui.mvc.ICotroller` interface, or subtypes thereof. A controller manages part of an application workflow, which could be a single view or a complex wizard consisting of multiple steps.

The most common type of a controller is a view controller. View controllers for MIDP and LWUIT views need to implement `com.aurorasoftworks.signal.runtime.ui.mvc.midp.IViewController` and `com.aurorasoftworks.signal.runtime.ui.mvc.lwuit.IViewController`, respectively. The framework automatically forwards commands (`javax.microedition.lcdui.Command` and `com.sun.lwuit.Command`) to a controller of the view that dispatched them. Multiple view controllers can point to the same view.

The second type of a controller is a flow controller that orchestrates a reusable process, typically a wizard consisting of multiple screens. Upon completion a flow returns a result to its caller, much like a method invocation. Flow controllers need to implement the `com.aurorasoftworks.signal.runtime.ui.mvc.IFlowController` interface. Flows can be chained together, i.e. a flow can start other flows, including instances of the same flow class.

Dependencies between controllers are defined as bean dependencies, so that controllers can "fire" events by invoking interface methods. This results in loose coupling of controllers, declarative definition of navigation rules and type safety. Interface method calls are intercepted by the framework to transparently perform required processing like displaying a correct view, registering command handlers and starting or stopping a flow.

In most cases it is desirable to have controllers and views lazily initialized. If so, an application context needs to be configured to do so:

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd"
        default-lazy-init="true">
        <!-- ... -->
</beans>
```

The most important bean in an MVC application is a dispatcher. The dispatcher is a framework object that intercepts method calls and performs needed processing like displaying a correct view or registering a command listener:

```
<bean id="dispatcher"
        class="com.aurorasoftworks.signal.runtime.ui.mvc.midp.Dispatcher"
```

```
/>
```

There are two implementations of the dispatcher concept: `com.aurorasoftworks.signal.runtime.ui.mvc.midp.Dispatcher` and `com.aurorasoftworks.signal.runtime.ui.mvc.lwuit.Dispatcher`, used in MIDP and LWUIT applications, respectively.

A definition of a dispatcher is followed by controller and view definitions, as shown below. Dependencies between controllers are defined as bean dependencies.

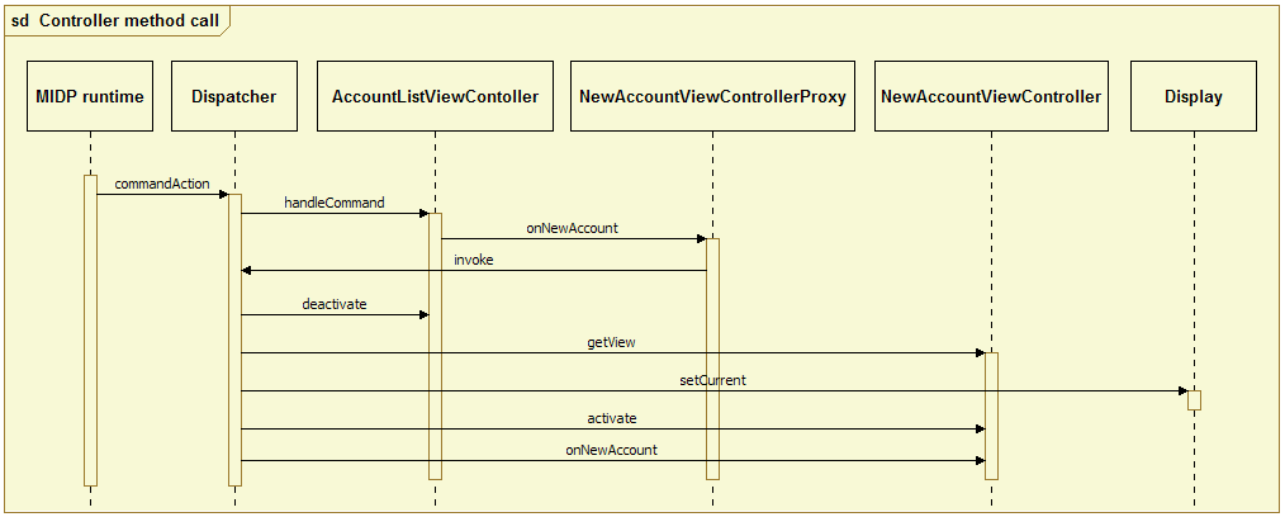
```
<bean id="accountListViewCtl"
class="com.aurorasoftworks.signal.examples.ui.mvc.midp.AccountListViewCon
troller">
        <constructor-arg ref="accountListView" />
        <constructor-arg ref="accountService" />
        <property name="newAccountEvent" ref="newAccountCtl" />
        <property name="editAccountEvent" ref="editAccountCtl" />
</bean>
```

In the example above a controller named `accountListViewCtl` supports two types of events: `newAccountEvent` and `editAccountEvent` that are wired to two other controllers as bean properties. Those events are simply references to interfaces shown below. The controller fires events by invoking interface methods which are in turn intercepted by the dispatcher.

```
public interface INewAccountEvent
{
        void onNewAccount();
}

public interface IEditAccountEvent
{
        void onEditAccount(IAccount account);
}
```

The following diagram illustrates the sequence of calls that are executed to handle user input in the scenario described above:



A command selected by a user is sent to the dispatcher (`CommandListener.commandAction`), which in turn forwards it to the active controller (`ICommandHandler.handleCommand`). `AccountListViewController` reacts to the command by firing the `INewAccountEvent.onNewAccount` event that gets intercepted by the dispatcher. The dispatcher deactivates `AccountListViewController`, changes current view to the one associated with `NewAccountViewController` and activates it. The transition from one controller to another is correctly reflected in the state of the application: `NewAccountViewController` is the active controller and its view is displayed in the screen.

The source code distribution of the Signal Framework contains a sample GUI application implemented in both MIDP and LWUIT technologies that demonstrates the usage of MVC concepts.

Summary

The framework has been designed to balance two conflicting requirements: reuse as much of the Spring code as possible and make it easy to add support for other IoC containers in the future, if needed. The functionality described in this article was implemented with a reasonably small effort: approximately 10 000 LOC, excluding sample applications included in the source code distribution.

Some tradeoffs had to be made because of the limitations of the Java ME platform; most notably the framework does not support annotations and instead requires application classes to implement framework interfaces in some cases. For the same reasons, generics are not used in the framework API.

After several beta releases the framework is now reasonable mature and

contains all major features that have been planned. A production release should be available by January 2010.

Additional information on the framework can be found at the following location: http://www.aurorasoftworks.com/products/signalframework

Marek Wiącek, SCEA