USER MANUAL

Version 0.9.0

Parsley AS2 Framework User Manual

Contents

8. Tasks – Handling asynchronous operations

9. Utilities

1. Introduction

1.1. Purpose of the framework

Parsley is an Action Script 2 application framework. It tries to fill some gaps that the core functionality built into the Flash IDE left. It tries to prevent you from writing repitive infrastructure code. It tries to help you build your application with the MVC (Model View Controller) pattern in that it keeps the configuration and initialization of your View classes separate from the rest of the application. It may prevent you from ever having to write custom XML configuration parser logic. It gives you an UI component framework that is fundamentally different from the one built into Flash in that it lets you skin them in virtually any way you want.

The framework does not have a rather "academic" origin. It was developed driven by the requirements of real-world projects and it is very likely that further improvements will be added depending on the requirements of future projects.

1.2. Feature Overview

Parsley includes an extensive XML configuration framework. You can use it to configure your UI elements like components or text fields, configure other objects like controller objects, set up dependencies and listeners, configure internationalization and logging and other aspects of your application. The core class of the configuration framework is the `ApplicationContext` which will be described in Chapter 2. The syntax and all the options of the XML configuration file itself is described in Chapter 3.

Another large part of the framework is the UI framework with its own set of components. It contains classes for navigation, history, screens and subviews, component configuration, UI event handling, tooltips, popup messages, etc. The Parsley UI elements are described in Chapters 5, 6 and 7.

Parsley also contains a bunch of useful utilities, like the `Task` framework for handling asynchronous tasks (Chapter 8), classes for event handling (Chapter 4), a small AOP framework and other utilities (Chapter 9).

2. ApplicationContext – The Parsley IoC container

2.1. Overview

The `ApplicationContext` is the heart of the framework. It is the entry point to most of the features if you use the standard Parsley XML configuration. Central ideas of this functionality were heavily influenced by the Spring Framework (www.springframework.org) which is a sophisticted IoC container for Java applications.

IoC (Inversion of Control) lets you easily decouple configuration from application logic. A traditional approach might be to write controller classes which load and parse their configuration files themselves. The disadvantage is that your controller objects have to know about the location and the structure of their configuration. If you add a new object which needs a new set of configuration options it is very likely that you develop a new set of functions to load and parse this configuration for each new aspect of your application.

The purpose of the Parsley `ApplicationContext` is to eliminate the need to develop custom XML formats and custom logic for loading and parsing the configuration. It is so generic that you can use it to configure any ActionScript 2 class. An example configuration might look like this:

```
<object id="controller" class="com.domain.package.MyController"
 singleton="false">
    <constructor-args>
        <number>23</number>
        <string>a string</string>
    </constructor-args>
    <property name="helper"><object-ref ref="myHelper"/></property>
    <property name="flag"><boolean>true</boolean></property>
</object>

<object id="myHelper" class="com.domain.package.MyHelper
 singleton="false">
```

If you obtain an object with the id "controller" from the corresponding `ApplicationContext` it has the same effect as if you constructed it programmatically:

```
var controller:MyController = new MyController(23, "a string");
controller.setHelper(new MyHelper());
controller.setFlag(true);
```

Thus the classes do not have to know anything about the way how they are constructed. It does not make any difference if you create them programmatically or obtain them through an `ApplicationContext` instance. If they are configured through XML, they do not load and parse the configuration themselves, it is the task of the IoC container, the `ApplicationContext` in this case, to configure the instances "from the outside". This is a very simple and short description for the concept behind "Inversion of Control"

In addition to this generic object tag there are several special tags to configure UI elements like Navigators, Screens, Views, Components and Text Fields. Furthermore you can configure internationalization, logging, and local preferences in your `ApplicationContexts`.

2.2. Loading and parsing configuration files

To obtain a reference to an `ApplicationContext` instance you first need to load one or more configuration files with an `ApplicationContextLoader`:

```
public function init () : Void {
    // create instance – the name will be passed to the ApplicationContext:
    var parser:ApplicationContextParser =
                                new ApplicationContextParser("myName");

    // add one or more files:
    parser.addFile("main.xml");
    parser.addFile("help.xml");

    // add listener methods:
    parser.addListener(TaskEvent.COMPLETE, this, onLoad);
    parser.addListener(TaskEvent.ERROR, this, onError);

     // optionally set the locale programmatically
    // (there are different ways to do this, see section 3.6. for details)
    parser.setLocale(new Locale("en", "UK"));

    // start the parser:
    parser.start();

}

public function onLoad (parser:ApplicationContextParser) : Void {
    var context:ApplicationContext = parser.getApplicationContext();
    // now you can use the context
}

public function onError (parser:ApplicationContextParser,
                         errorCode:String, errorDetail:String) : Void {
    trace("ERROR: " + errorCode + " – " + errorDetail);
}
```

2.3. Multiple configuration files in a single context

You can use more than one configuration file in a single `ApplicationContext`. There are two ways to add additional files. The first was demonstrated in the previous section, you just add the files to the `ApplicationContextParser`. The second options is to use the `include` tag in the configuration file described in section `3.4`. In both cases all the features, objects and UI elements you configure in those files will add up to a single context as if they were all configured in a single file. This means that you can configure objects that depend on other objects which are configured in a different file. All dependencies will resolved after all configuration

files were loaded and parsed. Thus it is sufficient to define central objects or constants only once and refer to those definitions in other files.

The main advantage of splitting your configuration across multiple files is that it makes it easier to swap a single aspect of your configuration. If you develop in a team you can, for example, extract all logging configuration to a separate file and each developer only needs to edit a single include tag to switch to his personal settings. The second advantage is that you can structure your configuration in any way you want which is especially convenient for large projects.

2.4. Using multiple context instances

Although it may help structure your application to split the configuration to several files, sometimes an application might become so big that you do not want to load all your configurations into a single context. This is especially true if you use the Parsley internationalization features and load message bundles into your `ApplicationContexts`. You may want to load the messages only when they are needed. In those cases you could split your application to modules and each of those modules uses a separate `ApplicationContext` instance. This way you can load a context instance when you load your module and destroy it when you switch to different module. This may help reduce your memory requirements.

Even if you use separate `ApplicationContexts` you can reuse the configuration for your central controller objects and components when you specify a parent context on your `ApplicationContextParser`:

```
parser.setParent(myContext);
```

This way you could load your main configuration file with settings for all the central objects when the applications starts. The modules can then refer to those central objects defined in the parent context in their local configuration files.

2.5. Methods of the ApplicationContext class

This section gives a quick overview of the methods of the `ApplicationContext` class because it is the entry point to many of the Parsley features. For all the other classes of the framework you should use the AS2Doc API documentation which contains descriptions for the public methods of the other framework classes.

```
getLocaleManager () : LocaleManager
getMessageSource () : MessageSource
```

For internationalized applications you can obtain a reference to the `LocaleManager` instance and to the `MessageSource` instance. In most cases you won't use these objects much in your application code. The `MessageSource` instance will mostly be used internally for binding internationalized messages to text fields or object properties. The `LocaleManager` might be used to switch the active locale.

```
getParent () : ApplicationContext
```

Will return the parent of this ApplicationContext or null if none was specified.

```
getMessage (aKey:String, bundle:String) : String
```

Internationalized messages can be obtained with this method. Note that the `MessageSource` instance has a getMessage method too, but the advantage of calling the method on the `ApplicationContext` instance is that the context will call the parent if the local `MessageSource` instance cannot find a message for the specified key.

```
getPreferenceManager () : PreferenceManager
```

With this method you can obtain a reference to the `PreferenceManager` which is an obejct that handles writing and reading properties from Local Shared Objects (LSO). The type and structure of the preferences can also be defined in the XML configuration file.

```
getConfig () : ApplicationContextConfig
```

In the rare cases when you need access to the object graph representing the XML configuration loaded, you can get a reference to an `ApplicationContextConfig` instance with this method. Note that this object does not represent a single configuration file but the result of merging all the included files.

```
getName () : String
```

Each `ApplicationContext` is also identified by a name, which will be returned by this method. The name will also be used internally to name the Local Shared Objects which store the preferences so that each `ApplicationContext` can have its own set of preferences.

```
getObject (id:String) : Object
getNavigator (id:String) : Navigator
```

These are the core factory methods for creating instances of any of the objects configured in XML. If your configuration file contains a node like this:

```
<object id="myObj" class="com.domain.package.SomeClass">
    <property name="url">
        <string>http://www.anywhere.com</string>
    </property>
</object>
```

you can create an instance of this object with the following method:

```
var obj:SomeClass = SomeClass(context.getObject("myObj"));
```

The instance returned will be a fully configured object. If the singleton attribute is set to true (the default), the instance will only be created once and subsequent calls will return the same instance.

```
destroy () : Void
```

The last public method of `ApplicationContext` is `destroy` which will be explained in the next section.

2.6. Destroying context instances

Sometimes loading an ApplicationContext and all corresponding internationalized message resources and constructing a bunch of instances with managed dependencies like UI element bindings or listeners or interceptors will increase the memory requirement. If you want to get rid of an `ApplicationContext` and all its loaded resources you can call its destroy method. This will remove all listeners that were added by this context, remove all interceptors, unload all message resources and preferences and delete all the internal references to objects that were constructed by this context. If you have any active navigators the moment you invoke `destroy`, the currently active screen in that navigator will be removed and all UI elements currently on the stage that are associated with this context will disappear.

In addition any destroy methods that you defined with the `destroy-method` tag will be called.

3. XML Configuration

3.1. Overview

The XML configuration file in Parsley lets you configure many central aspects of your application and your UI elements. Let's start with a quick overview of the top level tags:

constant	Use this tag to define constants that you use in other parts of the configuration
include	Includes other configuration files and merges their content with the current file
import	Imports AS2 classes so you can use its short name when configuring class names
locale-manager	Configures the LocaleManager used for this ApplicationContext
message-source	Use this tag to define which messages should be loaded into the context
preference-manager	Set the names and data types of all preferences you use
log-factory	Gives you fine-grained control over log levels
static-initializer	Used to set static properties and call static methods
object	Generic object configuration facility
navigator	The top level tag for all UI elements configured in an ApplicationContext

3.2. XML Schema

Parsley comes with a XML Schema definition for its configuration file format. You can use this Schema document as a reference or to validate the XML files in your IDE. Some IDEs like IDEA come with this capability built in, if you use Eclipse you'll need a Plugin for Schema validation.

One thing to keep in mind is that the Action Script XML class does not contain Schema validation. The files loaded with the `ApplicationContextParser` class will be validated internally. This internal validation is less strict than the Schema validation, because some structures in Schema could only be described with forcing a certain order of elements, which Parsley does not care about. So in some cases you may find out that your IDE detects validation errors in your configuration files although Parsley loads them without complaining.

On the other hand there are some configuration errors that cannot be detected with Schema validation. Some examples are class names that cannot be resolved or internal references to objects configured in other configuration files. These errors will only be detected when the files are actually loaded into the `ApplicationContext`. The context parser will keep track of all the errors in the files and if there are one or more errors the `TaskEvent.ERROR` event will be fired instead of the `TaskEvent.COMPLETE` event and a summary of all the detected errors will be in the errorDetail parameter of the error callback.

3.3. Defining constants

Sometimes you use the same chunks of text over and over in your configuration files. This is error-prone and means typing a lot. In these cases you can use the constant tag to define values that you want to reuse in other parts of the configuration. If you define a constant like this

```
<constant name="basePath">http://www.anywhere.com/base</constant>
```

you can reference this string in other atrributes or text nodes like this:

```
<object id="myObj" class="com.domain.package.MyClass">
    <property name="imageLocation">
        <string>${basePath}/images</string>
    </property>
</object>
```

3.4. Including other configuration files

If you want to include additional configuration files simply add them with the include tag:

```
<include file="other.xml"/>
```

You can include an unlimited number of additional files. The framework will always parse the include tags first no matter in which order they appear in the configuration file. Other tags will be parsed only after all included files are loaded. Note that you can also add additional files programmatically with the addFile method of `ApplicationContextParser`.

3.5. Importing classes and packages

In many tags you have to specifiy class names. To facilitate this procedure you can import single classes or even complete packages similar to the import statement in AS 2 class files. If you configure a lot of UI elements for example and if you heavily use some of the standard Parsley components it may be convenient to import the whole `parsley.comp` package like this:

```
<import class="org.spicefactory.parsley.comp.*"/>
```

After that import you can specifiy all components in that package just with their class name:

```
<navigator id="main">
    <screen name="first" linkage="screen_1">
        <component name="box" class="CheckBox"/>
    </screen>
    <screen ...>
    </screen>
</navigator>
```

3.6. Internationalization

An example configuration for the locale-manager might look like this:

```
<locale-manager persistent="true">
    <default-locale language="en" country="UK"/>
    <locale language="en" country="US"/>
    <locale language="de" country="DE"/>
    <locale language="fr" country="FR"/>
 </locale-manager>
```

With the persistent attribute you specify if the framework should remember the last active locale (it will be stored in a Local Shared Object). The `default-locale` tag is a second option to determine the first active locale after the configuration is loaded. With the remaining `locale` tags you specify all locales that are supported by your application. In most cases those are the locales for which you include message resources. The exact order of actions taken by the framework to determine the first active locale looks like this (the execution of those checks will stop as soon as one of the following checks succeeds):

1. First it checks if you specified a startup locale programmatically with `ApplicationContextParser.setLocale` and if the specified locale is among the supported locales of the loaded XML file.
2. Then it checks if the persistent attribute is set to true and there is a corresponding Local Shared Object which contains the last active locale and that locale is among the supported locales of the loaded XML file.
3. Then it looks if there is a `default-locale` tag. Note that you do not have to include that locale with the other supported locales because the locale specified with the `default-locale` tag will be automatically added to the set of supported locales.
4. Then it checks the language with `System.capabilities.language` and if that language is among the supported locales.
5. If all the above checks fail it uses the empty locale as default.

It is recommended to use ISO language codes defined by ISO-639 and ISO country codes defined by ISO-3166. The language code is a lower-case, two-letter code, the country code is an upper-case, two-letter code.

The active locale will be used to determine which message resources to load. Message resources are explained in the next section.

3.7. Localized Messages

When the configuration files are loaded and parsed and the first active locale has been determined, the framework will load all message resources specified with the `message-source` tag for the active locale. An example configuration may look like this:

```
<message-source>
    <default-message-bundle id="main" basename="main" localized="true"/>
    <message-bundle id="links" basename="links"/>
</message-source>
```

The id attribute is used to refer to the bundles in application code and configuration options. The basename is used to determine the filenames of the localized messages. In the above example the framework will look for the following files, assuming that the default locale has the language code en and the country code UK:

main_en_UK.xml

main_en.xml

main.xml

links.xml

Three files will be loaded for the default bundle: The first one contains message specific to the UK. The second file contains other english messages and the third one contains messages for that bundle which are the same for all languages. If you do not need one of those three files just use an XML file with an empty root node. Loading empty files is faster with Flash than waiting for a timeout for files which cannot be found. Note that for the second bundle only a single file will be loaded because the localized attribute was not set for that bundle.

The XML files containing the messages have a very simple format:

```
<message-bundle>
    <message key="tooltip.editor.newFile">Create a new file</message>
    <message key="tooltip.editor.editFile">Edit this file</message>
    <...>
</message-bundle>
```

But how do you use these message bundles? There are three options to include those messages in your application. The first one is to programmatically read them from your ApplicationContext with the getMessage method:

```
var msg:String = appContext.getMessage("key", "bundle");
```

The second argument is optional, if left out the default bundle will be used.

The other two options are to refer to those message bundles from within your configuration files, either when setting properties or when setting initial values for text fields:

```
<navigator id="main">
    <screen name="first" linkage="screen_1">
        <text path="myField">
            <message-binding key="myKey" bundle="main"/>
        </text>
    </screen>
    <screen ...>
    </screen>
</navigator>
```

Again the bundle attribute is optional if you refer to the default bundle. You can also bind localized messages to arbitrary properties of any object:

```
<object id="localizedObj" class="com.domain.i18n.ErrorMessages>
    <property name="errorCode">
        <message-binding key="app.error"/>
    </property>
</object>
```

If you configure your objects or text fields like this, every time you switch the current locale the value bound to that particular message key will be automatically updated. If you do not want to those values to be updated or if your message is not localized anyway, you should use the message tag instead of the `message-binding` tag. The `message` tag will only set the initial value without establishing a link between the object and the message source.

But how do you switch the active locale? The `LocaleManager` contains a method to perform this task:

```
appContext.getLocaleManager().setCurrentLocale(new Locale("fr", "FR"));
```

If you call this method the corresponding message source instance will load new bundle resources for the specified locale. You can add a listener to the `MessageSource` instance to be notified when the new resources are completely loaded. The `MessageSource` instance fires three events: `MessageSourceEvent.LOAD` when it starts loading new resources, `MessageSourceEvent.COMPLETE` when it successfully loaded new resources, and `MessageSourceEvent.ERROR` when it could not complete the task and rolled back to the previous locale.

There are also two attributes for the `message-source` tag. The `cacheable` attribute should be set to true if the `MessageSource` instance should keep all previously loaded messages when it switches to a new locale. The `class` attribute can be used to tell the `ApplicationContext` to use your own implementation of the `MessageSource` interface.

3.8. Preferences

This is an example configuration for the `PreferenceManager`:

```
<preference-manager>
    <preference name="volume"><number>100</number></preference>
    <preference name="filenames">
        <array>
            <string>succes.mp3</string>
            <string>error.mp3</string>
        </array>
    </preference>
</preference-manager>
```

Each preference tag defines a single preference which will be stored in a Local Shared Object. The defaults specified in the configuration file will only be used if the framework cannot find a value in the LSO for a particular preference name. The datatypes you can specify are a subset of those you can use within a property tag and are listed in section `3.17`.

In addition to the supported standard data types there are three special types that are only available inside a preference tag: `persistentObject`, `persistentArray` and `customPreference`. They are extensions of the standard `object`, `array` and `custom` tags respectively and add a `persister` attribute. Since the Flash Player is not able to store instances of custom classes in Local Shared Objects (they would be degraded to generic Objects) you'll need the help of a custom persister to read and write objects from and to a Local Shared Object. Each persister must implement the `Persister` interface of the `parsley.config` package

which contains two methods for reading and writing objects. In the case of the `persistentArray` tag, the framework will use the specified persister for each element of the array.

3.9. Logging

The `log-factory` tag lets you specify log levels and appenders for the logger instances Parsley uses internally as well as for your own logger instances. Let's show an example configuration:

```
<log-factory>
    <root level="warn">
        <appender threshold="debug">
           <object class="${parsley}.logging.appenders.TraceAppender"/>
        </appender>
    </root>
    <logger name="${parsley}.comp" level="debug"/>
</log-factory>
```

The Parsley logging framework is heavily inspired by log4j, the defacto standard for logging in Java. It has a hierarchical structure of logger instances which gives you fine grained control over which messages will be logged. The only required tag is the definition of the root logger. It contains the default level for all loggers that have no matching logger tag. In the example above all loggers will ignore messages weaker than log level warn except for all loggers created with the name of the package org.spicefactory.parsley.comp or subpackages. Those loggers will also show messages that have a log level of info or debug.

To explain the logic used internally to determine the log level for each logger let's look at an example. If you create a logger like this:

```
var log:Logger =
     LogFactory.getLogger("org.spicefactory.parsley.comp.select.List");
```

the framework will first look for a logger definition for the fully qualified class name. If it cannot be found it cuts the last part of the name and looks for a logger with a name "`org.spicefactory.parsley.comp.select`". Since we did not define such a logger it cuts the last package again and then finally finds the definition for "`org.spicefactory.parsley.comp`". So the level for that logger will be debug. Since debug is the lowest level it means that all log messages will pass. If we hadn't specified this logger the last name the framework would have looked for would have been a logger with name "`org`". If this would not exist the effective level would be the level of the root logger.

The last step is to specifiy the output for the log messages. To see anything you have to specify at least one appender, usually in the root logger. Parsley comes with a simple `TraceAppender` which simply writes to the Flash Output window. But you can write your own appenders, implementing the `Appender` interface or subclassing `AbstractAppender`. This way you can route the messages to files, to a server or anywhere you like. In contrast to the logger hierarchy you do not overwrite an appender specified at the root level. If you include an appender tag for a specific logger, this appender will be added to all the appenders specified at higher levels. So a single log statement might go to multiple destinations.

Note that an appender adds his own threshold for the log level. So even if the logger itself has an appropiate level so that the message passes, each appender specified for that logger or for a logger at a higher level will additionally check if the level of the messages is equal or above the threshold. This way you can have different thresholds for different output destinations. For example you could set the threshold for the `TraceAppender` to debug so that all messages pass, but set the level for your custom `RemoteAppender` which logs to a file on the web server to error, so only error messages will stored on the server.

3.10. Static initializers

The `static-initializer` tag enables you to specify static properties and static methods which will be executed after the configuration file has been parsed. An example might look like this:

```
<static-initializer class="com.domain.package.Environment">
    <property name="applicationLoader">
        <object class="com.domain.package.ApplicationLoader"/>
    </property>
    <method-call name="init">
        <number>0</number>
        <application-context-ref/>
    </method-call>
</static-initializer>
```

This has the same effect as the following code snippet:

```
var context:ApplicationContext = getReferenceSomehow();
Environment.setApplicationLoader(new ApplicationLoader());
Environment.init(0, context);
```

3.11. Creating objects

The object tag is a generic configuration facility. Together with the top level navigator tag it is the core of the configuration framework. This tag should help you avoid developing custom configuration parsers and logic. It should even enable you to configure existing classes that were never explicitly prepared to be configured by the Parsley framework.

The most simple example would look like this:

```
<object id="test"/>
```

You can obtain a reference to this object with an `ApplicationContext` instance:

```
var obj:Object = context.getObject("test");
```

This does not make that much sense because it is effectively the same like the following line:

```
var obj:Object = new Object();
```

So in most cases you would specify a class attribute:

```
<object id="test" class="com.domain.package.TestClass"/>
```

Again you obtain a reference with a context instance:

```
var obj:TestClass = TestClass(context.getObject("test"));
```

This is equivalent to the following code:

```
var obj:TestClass = new TestClass();
```

In addition to these simple examples Parsley gives you a lot of optional control over the creation of the objects. For example you can specify constructor arguments:

```
<object id="test" class="com.domain.package.TestClass">
    <constructor-args>

        <string>just a string</string>
        <array>
            <message key="test.message.1"/>
            <message key="test.message.2"/>
        </array>
        <null/>
    </constructor-args>
</object>
```

For an overview of all the different data types that are legal as nested tags for the constructor arguments, see section 3.17. The above configuration is equivalent to the following code:

```
var context:ApplicationContext = getReferenceSomehow();
var msg1:String = context.getMessage("test.message.1");
var msg2:String = context.getMessage("test.message.2");
var test:TestClass = new TestClass("a string", [msg1, msg2], null);
```

Alternatively instead of specifying constructor arguments you can tell Parsley to use a factory, either static factory methods or a factory instance that you configured with a different tag. Let's look at an example for the first case:

```
<object id="test">
    <static-factory-method class="com.domain.package.TestFactory"
     name="createObject">
        <string>just a string</string>
        <null/>
    </ static-factory-method>
</object>
```

Note that we omit the class attribute in the object tag. We delegate the responsibility of object creation to a static factory method and somehow do not care about the type of object that method returns. Instead we configure the class name of the factory, the method name and the arguments that should be passed to the factory method. This is equivalent to the following code:

```
var obj:Object = TestFactory.createObject("just a string", null);
```

The most sophisticated option is to use a factory instance. This has to be created and configured in any different object tag. Let's show a complete example:

```
<object id="myFactory" class="com.domain.package.MyFactory"
 singleton="true">
    <property name="maxInstances"><number>5</number></property>
</object>

<object id="myInstance" singleton="false">
    <factory-method factory="myFactory" method="createObject">
        <string>just a string</string>
        <null/>
    </factory-method>
 </object>
```

The factory will be created and configured like any other object. In this case we set a property named maxInstances. Object configuration is described in the next section. The second object configuration uses the first as a factory. The factory attribute in the factory-method tag is the id of our factory. The above example is equivalent to the following code:

```
var factory:MyFactory = new MyFactory();
factory.setMaxInstances(5);
var obj:AnyType = factory.createObject("just a string", null);
```

We introduced another feature in the above example: The use of the singleton attribute. For the factory it is set to true (which is the default so you could omit the attribute in those cases). For the other object we set it to false. This means that the factory will only be created once, but each time you call `context.getObject("myInstance")` the same factory instance will be asked again to produce a new instance. If the singleton attribute for the second object would have been set to true, the factory method of the first obejct would have only be called once. Subsequent calls to `context.getObject("myInstance")` would always return the same object that was created with the first request.

Finally Parsley gives you also control over object destruction:

```
<object id="toBeDestroyed" class="com.domain.SoonToBeDestroyed">
    <destroy-method name="dispose">
        <boolean>true</boolean>
    </destroy-method>
</object>
```

This will tell the container to call dispose(true) on the instance when the ApplicationContext gets destroyed. The details of context destruction are described in section `2.6`.

3.12. Configuring objects

The previous section concentrated on object creation. Now we will give you an overview over all the remaining configuration objects. First let's look at a complete example:

```
<object id="myJpgLoader" class="com.domain.package.JpgLoader">
    <property name="directory">
        <string>http://www.anywhere.com/images/</string>
    </property>
    <method-call name="addFile">
        <string>background.jpg</string>
    </method-call>
    <method-call name="addFile">
        <string>logo.jpg</string>
    </method-call>
    <listener event="TaskEvent.COMPLETE" method="onComplete">
        <object-ref ref="myListener"/>
    </listener>
    <listener event="TaskEvent.ERROR" method="onError">
        <object-ref ref="myListener"/>
    </listener>
    <interceptor methods="getBytesLoaded,getBytesTotal">
        <object-ref ref="myInterceptor"/>
    </interceptor>
</object>
```

This is equivalent to the following code:

```
var context:ApplicationContext = getReferenceSomehow();
var listener:Object = context.getObject("myListener");
var ic:Interceptor = Interceptor(context.getObject("myInterceptor"));
var loader:JpgLoader = new JpgLoader();
loader.setDirectory("http://www.anywhere.com/images/");
loader.addFile("background.jpg");
loader.addFile("logo.jpg");
loader.addListener(TaskEvent.COMPLETE, listener, listener.onComplete);
loader.addListener(TaskEvent.ERROR, listener, listener.onError);
AopUtil.addInterceptor(loader, ["getBytesLoaded", "getBytesTotal"], ic);
```

In this example we used the four remaining child nodes of the object node: property, method-call, listener and interceptor. Let's explain them in detail:

property

The property tag can be used to set any property on the target object. But what exactly is a property in the Parsley world? There are three options to define a property so that it can be configured with Parsley. The first is using traditional getter and setter methods, not the special built in property accessor and mutator functions. In the above example the property was named `"directory"`. The first thing Parsley will check is if there is a `setDirectory` method and use it if present. Otherwise it will use the plain property syntax: `obj.directory`. This will cover the second and third option: To specify a special Action Script setter method: `function set directory ()` or to specify a plain property: `var directory` in the target class.

The data types allowed in the property tag are listed in section 3.17.

method-call

The method-call tag is straightforward. The name attribute is mandatory for obvious reasons. But the nested tags for the arguments are optional. Of course you can call a method without arguments. The data types allowed as method arguments are almost the same as those allowed for the property tag, listed in section 3.17. The only exceptions are the message-binding and the preference-binding tag, since it does not make sense to bind to method parameters.

listener

The listener tag lets you specify the event type and target object and method. It only works with the Parsley event model which is explained in Chapter 4. It expects an addListener and removeListener method in the configured object with the same signature as the methods with the same name in the Parsley EventDispatcher class:

```
public function addListener
    (event:EventType, target, method:Function) : Void

public function removeListener
    (event:EventType, target, method:Function) : Void
```

The target object i.e. the listener must be specified in a child node. In many places where only objects can be specified and not arbitrary data types you have two options: Use the object-ref tag to point to an id of a top level obejct node in your configuration file. If you know you won't use this instance elsewhere you can also define it inline with a nested object tag. Nested object tags cannot have id attributes and cannot be directly obtained with the getObject method of the ApplicationContext instance, but in many cases this will not be necessary anyway.

interceptor

The interceptor tag lets you define the methods which you want to be intercepted in a comma-separated string and the interceptor itself. Like the listener tag the interceptor tag expects either an obejct-ref child node or an nested object tag. AOP is explained in more detail in section 9.2.

3.13. Navigators

The navigator tag is the root tag for all configuration related to UI elements. It contains nested screen tags which in turn can contain tags for nested views, components, buttons or text fields. A very simple example might look like this:

```
<navigator id="navi">
    <screen name="loginScreen" linkage="screen_1">
        <button target="myTarget" method="onClick"/>
        <text path="aField"/>
    </screen>
```

```
        <screen name="registerScreen" linkage="screen_2">
            <button target="myTarget" method="onClick"/>
            <text path="aField"/>
        </screen>
    </navigator>
```

This simple `Navigator` contains two screens which in turn both contain a button and a text field. You can obtain a reference to this `Navigator` with the following method:

```
var nav:Navigator = context.getNavigator("navi");
```

Note that the navigator instance still is an completely abstract thing. It is not associated with anything on your Flash stage. How to activate this `Navigator` will be explained in section `5.1`. This section focusses on configuration.

The navigator tag has several optional attributes:

class

In the rare cases where the built-in `Navigator` class is not sufficient you can create a subclass with added functionality.

base-path

This atrribute is useful if the `Navigator` contains several `Screen` instances which point to resources in an external SWF file. If you specify this attribute, all `Screen` instances that have their filename attribute set will prepend the base-path.

stateful

A boolean value. Specifies if the nested Screens and their components are stateful. The default is false. You can specify a value in the navigator tag and overwrite it for a particular screen tag which also contains a `stateful` attribute. State management is explained in detail in section `7.21`.

transition-cancellable

A boolean value. The default is false. Specifies if a transition in progress can be cancelled when one of the `goXX` methods of the `Navigator` are called. Transitions are explained in detail in section `5.4`.

In addition to the listed attributes the navigator tag accepts all the child nodes that the object tags accepts and three special child nodes more. Of the child nodes that you can use with the object tag, you will most likely only rarely use the `property`, `method-call` or `interceptor` tag. The next section explains the remaining tags:

listener

```
Example:
    <listener event="NavigatorEvent.ENTER_SCREEN" method="onEnterScreen">
        <object-ref ref="myNaviListener"/>
    </listener>
```

The `Navigator` class accepts two different kinds of listeners: Listeners for `NavigationEvents` or listeners for `TaskEvents` and `MovieLoaderEvents` if you want to listen to the events related to loading screens associated with external files. The syntax is the same as for the `object` tag. The `listener` tag expects a mandatory child node which must be either an `object-ref` tag or an nested `object` tag. Note that the context instance will check if the specified method exists. If it does not exist an error will be logged and the `getNavigator` method of the ApplicationContext instance will return null if one or more errors occurred during construction of the `Navigator` instance.

transition

```
Example:
    <transition>
        <object class="com.domain.package.MySpecialTransition"/>
    </transition>
```

This child node is optional. Each navigator can contain only one transition. The tag expects a mandatory child node which must be either an `object-ref` tag or an nested `object` tag. The specified object must implement the `Transition` interface of the `parsley.ui` package. Otherwise an error will be logged and the `Navigator` instance will not be constructed. Transitions are explained in detail in section `5.4`.

voter

```
Example:
    <voter>
        <object class="com.domain.package.MySpecialVoter"/>
    </voter>
```

This child node is optional. Each navigator can contain an unlimited number of NavigationVoters. The tag expects a mandatory child node which must be either an `object-ref` tag or an nested `object` tag. The specified object must implement the `NavigationVoter` interface of the `parsley.ui` package. Otherwise an error will be logged and the `Navigator` instance will not be constructed. `NavigationVoters` are explained in detail in section `5.6`.

screen

The screen tag is explained in the next section.

3.14. Screens and Views

Screen tags can be nested in navigator tags. View tags are used to define subviews within screens or other nested views. Screens and views are essentially the same. If you look at the Parsley source code, you'll notice that `Screen` is in fact a subclass of `View` which adds only a small number of methods. The subtle differences are explained in detail in section `6.1`.

An example configuration for a screen:

```
<screen name="myScreen" linkage="screen_symbol" stateful="false">
    <listener event="ViewEvent.ENTER" method="onEnterScreen">
        <object-ref ref="mainController"/>
    </listener>
    <listener event="ViewEvent.EXIT" method="onExitScreen">
        <object-ref ref="mainController"/>
    </listener>
    <view name="myNestedView">
        <button name="sumbit" target="formHandler" method="submit"/>
        <button name="cancel" target="formHandler" method="cancel"/>
        <text path="head"><message-binding key="form.head"/></text>
    </view>
    <button name="help" target="mainController" method="showHelp"/>
    <component name="box" class="ComboBox" stateful="true">
        <listener event="SelectionEvent.CHANGE" method="onChange">
            <object-ref ref="mainController"/>
        </listener>
    </component>
</screen>
```

A list of all the attributes the screen and view tags support:

name

This attribute is required. It has slightly different semantics for `screen` and `view` tags. For `screen` tags this attribute defines just an identifier. The identifier can be used with several methods of the `Navigator` class like `goName` or `getScreenByName`. For `view` tags this attribute describes the instance name for the nested view component within the screen movie clip. If you omit the linkage attribute for the view tag, the framework will assume that a nested movie clip with the specified name was added to the screen clip in the authoring environment. It will throw an error if it cannot find a clip with the given name. If you do specify a `linkage` attribute, the corresponding clip must not be present in the screen clip, in this case the `name` attribute specifies the instance name that the framework will assign to the attached clip if you call `View.createElement`.

linkage

This attribute is required for the screen tag but optional for the view tag. It should refer to the linkage identifier that you assigned to the movie clip in the Flash IDE. For screen tags the `Navigator` instance will use this linkage identifier to attach the corresponding movie clip the internal container movie clip if you navigate to this screen. For `view` tags you'll only need to

specify this attribute if you intend to add this view to the parent view or screen dynamically. If you already added the nested view clip to the parent clip in the authoring environment you should omit this attribute.

class

In the rare cases where the built-in `Screen` or `View` classes are not sufficient you can create a subclass with added functionality. For screen tags it must be a subclass of `Screen`, for view tags it should be a subclass of `View`.

stateful

A boolean value. This attribute is optional. The default is to use the `stateful` attribute of the parent element. In the case of a screen tag this is the `stateful` attribute of the `navigator` tag, in the case of a `view` tag it is the `stateful` tag of the parent element (`screen` or `view`). Note that in our example we set the `stateful` attribute to false for the `screen` tag but overwrite the value for the combobox component. This way only the combobox will remember its state. State management is explained in detail in section `7.21`.

filename (screen tag only)

A screen can be associated with a movie clip in an external SWF. In this case the `linkage` attribute will point to a movie clip in the library of the external file. If you set the `base-path` attribute of the `navigator` tag, that path will be prepended to the specified filename. Working with external files is described in detail in section `5.5`.

A list of all supported child tags:

property, method-call, listener, interceptor

Like the `navigator` tag the `screen` and `view` tags support the same child tags like the generic `object` tag. It is unlikely that you would use `property`, `method-call` or `interceptor` child tags often. But the `listener` tag can be used to bind `screen` or `view` events to controller classes. Those lifecycle events are explained in detail in section `6.4`.

binding

`Example:`

```
    <binding target="myController" property="loginScreen"/>
```

You can bind a property of any object defined with a top level `object` tag to a screen or view instance. Details about the options to define the property in the target class are explained in section `3.12`. If the screen or view instance gets destroyed the property of the target obejct will be set to undefined.

view

Example:

```
    <view name="nestedView">
        <component name="box" class="CheckBox"/>
        <text path="username"><string>enter your name here</string></text>
    </view>
```

A `view` tag can be nested in a `screen` tag or in another `view` tag. There is no limit for the depth of the structure.

component

Example:

```
    <component name="data" class="SelectionList" stateful="true">
        <binding target="myController" property="selectionList"/>
    </component>
```

The `component` tag is described in the next section.

button

Example:

```
    <button name="submit" target="formController" method="submit"/>
```

The `button` tag is described in the next section.

text

Example:

```
    <text path="container.nestedField">
        <message-binding key="editor.title"/>
    </text>
```

The `text` tag is described in section `3.16`.

3.15. Components and Buttons

The component tag let's you define a single component. All components must either be one of the standard Parsley components or a custom subclass of the core `Component` class.

Example:

```
    <component name="data" class="SelectionList" stateful="true">
        <binding target="myController" property="selectionList"/>
    </component>
```

Supported attributes:

name, linkage, class, stateful

These four attributes are similar or identical to those of the `screen` and `view` tags that are explained in section `3.14`. The `class` attribute is different though in that the attribute will not be used to create the component (the actual class of the component will be defined in the library of the Flash file). It will only be used to verify that the component is of the correct type. If you do not specify the `stateful` attribute for a component the `stateful` attribute for the containing `view` or `screen` tag will be used.

tooltip

You can define internationalized tooltips with this attribute. The value of the attribute will be interpreted as a key in the message bundle specified with the `bundle` attribute. If you omit the `bundle` attribute the default bundle will be used.

bundle

This attribute is optional. It can be used to define the bundle that should be used for the tooltip message. If you do not specify a `tooltip` attribute or if you want the default bundle to be used, this attribute can be omitted.

Supported child nodes:

property, method-call, listener, interceptor

Like the `navigator`, `screen` and `view` tags the `component` tag supports the same child tags like the generic `object` tag. Those tags are explained in section `3.12`.

binding

`Example:`

```
<binding target="myController" property="checkBox"/>
```

You can bind a property of any object defined with a top level `object` tag to a component instance. Details about the options to define the property in the target class are explained in section `3.12`. The specified property of the target object will be updated continuously. If the screen or view the component belongs to is currently not active, the property will be reset to undefined. When the screen or view is active the property will be set to the component instance after the onLoad event of the component has fired.

The `button` tag is just a convenient shortcut to define button components. Since you are likely to define quite a lot of buttons for your views and since those buttons will not need all the configuration options of the component tag in most cases, you can use the button tag instead:

```
<button name="submit" target="formController" method="submit"/>
```

The example above is just a shortcut for this rather verbose component definition:

```
<component name="submit" class="SimpleButton">
    <listener event="MouseEvent.RELEASE" method="submit">
        <object-ref ref="formController"/>
    </listener>
</component>
```

3.16. Text Fields

Example:

```
<text path="container.nestedField">
    <message-binding key="editor.title"/>
</text>
```

The `text` tag let's you configure text fields. The `path` attribute points to the instance name of the text field you specified in the Flash IDE. In contrast to the `name` attributes of the `view` and `component` tags, you must specify the full path to the text field relative to the containing screen or view if it is nested in one or more movie clips, since text fields are not able to find their corresponding parents like the Parsley components.

You can specify an optional child node if you want the field to be initialized with a value when the screen or view loads. Permitted child tags are: `string`, `preference`, `message`, `preference-binding`, `message-binding`. Note that the two binding tags create a permanent link between the text field and the specified preference or message resource. This is especially useful for internationalized applications since you can force all text fields to be updated automatically when the active locale changes.

3.17. Data types for value tags

This section describes all the tags for the many data types that can be used with tags like `property`, `array` or `method-call`.

null

Example:

```
<null/>
```

This tag represents the null value. There is really nothing more to say about it.

boolean

Example:

```
<boolean>true</boolean>
```

This tag represents a boolean value and only accepts the strings `true` or `false`.

number

Example:

```
<number>23</number>
```

This tag represents a number value. If it evaluates to NaN an error will be thrown during parsing.

string

Example:

```
<string>follow me</strin>
```

This tag can contain any string values including line breaks. If you want to use special characters like '<' you should enclose the text node in a CDATA section.

date

Example:

```
<date>2002-12-14 23:50:00</date>
```

This tag represents date values with the format YYYY-MM-DD HH:MM:SS. The time part is optional. The date string will be converted to an Action Script Date object.

number-array

Example:

```
<number-array>23,44,51,69,102</number-array>
```

The value of this tag will be converted to an array using ',' as a delimiter and converting all elements to numbers.

string-array

Example:

```
<string-array delimiter="|">Lucy|Jenny|Pam|Nikita</string-array>
```

The value of this tag will be converted to an array using the specified delimiter or ',' as the default delimiter.

array

Example:

```
    <array>
        <number>0</number>
        <string>stop making sense</string>
        <date>1999-12-31</date>
        <array>
            <string>A</string>
            <string>B</string>
        </array>
        <object class="com.domain.package.SomeWeirdClass"/>
    </array>
```

This tag can be used to define arrays with arbitrary content including nested arrays and objects. All tags described in this section are permitted for the array elements, except for the two binding tags `preference-binding` and `message-binding`.

list

Example:

```
    <list>
        <number>0</number>
        <string>stop making sense</string>
        <date>1999-12-31</date>
    </list>
```

This tag can be used to define `List` instances with arbitrary content including nested arrays, lists and objects. `List` is an interface of the `parsley.collection` package and serves as a convenient alternative to the builtin `Array` object. All tags described in this section are permitted for the list elements, except for the two binding tags `preference-binding` and `message-binding`.

object

Example:

```
    <object class="com.domain.package.Example">
        <property name="size">
            <number>1000</number>
        </property>
        <property name="helper">
            <object class="com.domain.package.Helper"/>
        </property>
        <listener event="TaskEvent.COMPLETE" method="onComplete">
            <object-ref ref="myListener"/>
        </listener>
    </object>
```

If you use the `object` tag for nested values it has the same structure like the top level `object` tag except for the `id` attribute which can not be specified for nested objects.

class

Example:

```
<class>com.domain.package.ExampleClass</class>
```

This tag converts the value to an instance of `ClassConfig` which is included in the `parsley.config.tree` package.

static-property-ref

Example:

```
<static-property-ref class="ExampleClass" property="name"/>
```

This tag evaluates to the specified static property of the specified class. The property can have an accessor method like `getName` or be defined as a simple property.

object-ref

Example:

```
<object-ref ref="formController"/>
```

This tag evaluates to an instance that you configured with a top level `object` tag.

navigator-ref

Example:

```
<navigator-ref ref="popupNavigator"/>
```

This tag evaluates to a navigator instance that you configured with a top level `navigator` tag.

app-context

Example:

```
<app-context/>
```

This tag evaluates to the instance of the `ApplicationContext` that processed this configuration. Sometimes it is convenient to pass a reference to one of the configured objects in case they need to obtain other objects or messages from the context later.

custom

Example:

```
<custom converter="myDateConverter">May 2nd, 1998</custom>
```

This tag allows you to specify a custom converter in case all the built-in converters for the value tags described in this section are not sufficient. The `converter` attribute points to the `id` of an object you defined with a top level `object` tag. This object must implement the `Converter` interface of the `parsley.config` package.

preference

Example:

```
<preference name="volume"/>
```

This tag evaluates to the value of the preference named volume. It will only be evaluated once and not be updated when the preference changes.

message

Example:

```
<message key="editor.popup.message" bundle="popups"/>
```

This tag evaluates to the message with the specified key. If you omit the optional `bundle` attribute the default bundle will be used. It will only be evaluated once and not be updated when the active locale changes. You should prefer this tag over the `message-binding` tag if the message is not internationalized.

preference-binding

Example:

```
<preference-binding name="volume"/>
```

This binding tag can only be used when nested within a `property` tag or a `text` tag. It binds that property or text field to the value of the preference with the specified name. Thus the value will be updated when the preference changes.

message-binding

Example:

```
<message-binding key="editor.popup.message" bundle="popups"/>
```

This binding tag can only be used when nested within a `property` tag or a `text` tag. It binds that property or text field to the an internationalized message. Thus the value will be updated when the active locale changes. If you omit the optional `bundle` attribute the default bundle will be used.

4. The Parsley event model

4.1. Differences to the Macromedia event model

Based on practical experience I decided to use a different event model than that used by Macromedias v2 components. I found two disadvantages with their event model:

1. The event name is a simple string. The compiler is not able to give you any kind of feedback or error message when you specify an event name that is not supported by the class you want to observe.
2. If you want to delegate the event you have to use the `Delegate.create` construct which is cumbersome when delegating is your default way of handling events. In all my projects I always used delegates to avoid scope issues and their cumbersome workarounds and to keep the code more readable.

Thus Parsley comes with its own `EventDispatcher` class. The signature for the methods you use to add or remove listeners to objects looks like this:

```
public function addListener
                     (event:EventType, target, method:Function) : Void

public function removeListener
                     (event:EventType, target, method:Function) : Void

public function removeAllListeners (target) : Void
```

All Parsley classes that support the `EventDispatcher` define their own set of methods for adding and removing listeners to be able to restrict the `EventTypes` to a certain corresponding subclass of `EventType`. The methods for the `View` class for example look like this:

```
public function addListener
                    (event:ViewEvent, target, method:Function) : Void

public function removeListener
                    (event:ViewEvent, target, method:Function) : Void

public function removeAllListeners (target) : Void
```

So you only have to look inside the `ViewEvent` source file to see which `EventType` constants are supported and what method parameters are passed into the event listener method. The event source itself will always be the first parameter passed to the listener method.

4.2. Adding the EventDispatcher to your own classes

If you want to create your own classes and want to be able to define listeners for them in your XML configuration files they must be built with the Parsley `EventDispatcher`. The following example class shows how to integrate the `EventDispatcher` into your own classes. The example class fires an `NAME_CHANGED` event every time the name property changes:

```
import org.spicefactory.parsley.events.EventDispatcher;
import com.domain.package.ExampleEvent;

class com.domain.package.ExampleClass {

    private var _name:String;
    private var _dispatcher:EventDispatcher;

    public function ExampleClass (name:String) {
        _name = name;
        // always pass "this" to the constructor of EventDispatcher so that
        // this object can be passed to the listener methods
        // as the first argument
        _dispatcher = new EventDispatcher(this);
    }

    public function addListener
                    (event:ExampleEvent, target, method:Function) : Void {
        // just delegate to the EventDispatcher:
        _dispatcher.addListener(event, target, method);
    }

    public function removeListener
                      (event:ViewEvent, target, method:Function) : Void {
        // just delegate to the EventDispatcher:
        _dispatcher.removeListener(event, target, method);
    }

    public function removeAllListeners (target) : Void {
        // just delegate to the EventDispatcher:
        _dispatcher.removeAllListeners(target);
    }

    public function setName (name:String) : Void {
        // This is the method that should trigger the event
        _name = name;
        _dispatcher.dispatchEvent(ExampleEvent.NAME_CHANGED, [_name]);

    }

}
```

All that is left is to define the event type in a separate class which subclasses `EventType`:

```
import org.spicefactory.parsley.events.EventType;

class com.domain.package.ExampleEvent extends EventType {

    // define all supported events as static constants
    // (in this case only one):
    public static var NAME_CHANGED:ExampleEvent
                                        = new ExampleEvent("NAME_CHANGED");
```

```
    // keep the constructor private to make sure
    // that only the event types created
    // as constants for this class exist:

    private function ExampleEvent (type:String) {
        // pass the type to the constructor of the super class:
        super(type);
    }

}
```

Now you are ready to use those example classes:

```
    var example:ExampleClass = new ExampleClass();
    example.addListener(ExampleEvent.NAME_CHANGED, this, onChange);
```

The signature for the `onChange` method should look like this:

```
    public function onChange (example:ExampleClass, newName:String) : Void
```

Note that two parameters are passed to that method: First the event source that will be passed to all listener methods followed by any parameters you passed to the `dispatchEvent` method of the `EventDispatcher` instance.

5. Navigation

5.1. Activating navigators

You already saw how to configure `Navigator` instances in section `3.13`. In this chapter we will demonstrate how to use `Navigator` instances in your application code. A simple configuration for a `Navigator` might look like this:

```
<navigator id="navi">

    <screen name="loginScreen" linkage="screen_1">
        <text path="aField"/>
    </screen>
    <screen name="registerScreen" linkage="screen_2">
        <text path="aField"/>
    </screen>
</navigator>
```

You can obtain an instance of this navigator with the `ApplicationContext`:

```
var nav:Navigator = context.getNavigator("navi");
```

This is effectively the same as to construct the instance programmatically:

```
var screen1:Screen = new Screen();
screen1.setName("loginScreen");
screen1.setLinkage("screen_1");
var text1:TextConfig = new TextConfig();
text1.setPath("aField");
screen1.addTextConfig(text1);

var screen2:Screen = new Screen();
screen2.setName("registerScreen");
screen2.setLinkage("screen_2");
var text2:TextConfig = new TextConfig();
text2.setPath("aField");
screen2.addTextConfig(text2);

var nav:Navigator = new Navigator();
nav.addScreen(screen1);
nav.addScreen(screen2);
```

After you obtained the navigator instance it is still an completely abstract thing. It is not associated with anything on your Flash stage. It is just a graph of configured UI elements. You create a link to your visual environment with the activate method:

```
var container:MovieClip = getTheContainerForThisNavigator();
nav.activate(container.createEmptyMovieClip("navi", 1);
```

The `Navigator` instance always expects a completely empty movie clip to be passed to its activate method. You still won't see anything. Finally you must tell the navigator to display one of the screens. The following three methods will all display the first screen which we configured with the name `"loginScreen"`:

```
nav.goFirst();
nav.goIndex(0);
nav.goName("loginScreen");
```

The navigator attaches the Screens to the empty movie clip you passed to the activate method using the linkage attribute you specified in the XML configuration. Further navigation methods are shown in the next section. If you want to remove the association from the `Navigator` instance to the container movie clip, just call the `deactivate` method:

```
nav.deactivate();
```

The nice thing is that the `Navigator` is still usable. You can activate it again with any empty movie clip. This way you could reuse the same `Navigator` instance in different parts of your application and in different places on the Flash stage.

5.2. Navigation methods

The methods the Navigator class offers are straightforward. Here is a short summary:

```
public function goName (name:String) : Boolean
```

This navigation method takes a string as a parameter, specifying the name of the screen you want to navigate to. This string value corresponds to the name attribute that you specified in the screen tag if the navigator was configured in XML or to the name that you passed to the Screen.setName method if you configured the navigator programmatically. If no screen with the specified name can be found this method will return false.

```
public function goIndex (index:Number) : Boolean
```

This method let's you navigate to the screen at the specified index. The order of the screens is either the order you configured them in XML or the order you added them with `Navigator.addScreen`. If the index is out of range this method will return false.

```
public function goScreen (scr:Screen) : Boolean
```

Navigates to the screen specified as an argument. Only succeeds if the screen instance belongs to the `Navigator` instance. You can obtain references to screen instances of a particular navigator with the `getScreenByName`, `getScreenByIndex` and `getCurrentScreen` methods.

```
public function goFirst () : Boolean
public function goLast () : Boolean
public function goNext () : Boolean
public function goPrevious () : Boolean
```

Those four methods can be used for relative navigation. They will return true if navigation succeeds and return false if, for example, you call `goFirst` although the first screen is already the currently active one. Do not confuse the `goNext` or `goPrevious` methods with the methods of the `History` class. The `goPrevious` method of the History class will return to the last active screen. The `goPrevious` method of the `Navigator` object activates the screen

that comes before the currently active screen in the order the screens were added to the Navigator.

```
public function goBlank () : Boolean
```

Removes the currently active screen from the stage. Will return false if no screen is currently displayed. You can navigate to a new screen later with any of the other navigation methods. This is different from the deactivate method, after which you cannot use the navigator any more until you reactivate it. This is also different from the destroy method after which you can no longer use this navigator object.

In addition to certain circumstances mentioned in the sections above which result in the methods returning false and preventing the navigation to proceed, all the navigation methods will return false if:

1. The Navigator was destroyed, was never activated or was deactivated.
2. There is a transition in progress and the transitionCancellable property was set to false.
3. Any of the NavigationVoters added for this Navigator returned false. NavigationVoters are explained in section 5.6.

5.3. History

Each Navigator instance creates an instance of the History class. You can obtain a reference with the getHistory method of the Navigator class and then use the navigation methods of the History class to implement functionality like back buttons:

```
var nav:Navigator = getNavigatorSomehow();
var h:History = nav.getHistory();
h.goPrevious();
```

You can clear the contents of the history with the clear method, and you can enable or disable the History.

5.4. Transitions

Section 3.13. explained how to configure transitions in XML. This section shows how to set them programmatically and how to implement your own transitions. Parsley does not come with a set of built-in transitions, you have to develop them implementing the Transition interface of the parsley.ui package.

Transitions are optional, if you omit the transition tag in the navigator tag, navigation to a new screen will occur immediately, attaching the new screen and removing the old one within a single frame. If you want to add a custom transition you can either specify it in XML, described in section 3.13., or you can set it dynamically with the setTransition method of the Navigator class. This way you can switch to a different transition any time you like.

The four methods of the Transition interface that you have to implement look as follows:

```
public function startTransition (controller:TransitionController) : Void ;
```

This is the first method that will be called when the transition starts. The controller instance passed in as an argument to all the methods of the Transition interface contains two methods that you can use to obtain references to the two screens involved in the transition: getPreviousScreen and getNextScreen. This method will be called before the next screen is initialized, thus you do not have access to the components of the next screen. In many cases you won't do anything in this method unless you need to prepare something for the new screen, e.g. loading images or XML. If you want the transition to continue just invoke the resume method in the controller instance.

```
public function prepareScreen (controller:TransitionController) : Void ;
```

This method will be called after you invoked TransitionController.resume for the first time during this transition and after the next screen was attached but before the next screen is fully initialized, e.g. the onLoad events for the components on the new screen have not fired yet. If you do something like a fade transition you should set the initial alpha property within this method to avoid flickering. Waiting for the enterScreen method which will be called after the onLoad events of the components on the new screen have fired is too late unfortunately. If your transition is complete just call resume on the controller instance a second time.

```
public function enterScreen (controller:TransitionController) : Void ;
```

This method will be called after the new screen is fully initialized, e.g. the onLoad events for all components on that screen have fired. In most cases you only need to implement this method if you need access to the components on the new screen. In most transitions you will use prepareScreen or enterScreen alternatively. If your transition is complete just call resume on the controller instance a second time.

```
public function cancelTransition (controller:TransitionController) : Void ;
```

This method gets called when the transition was cancelled. If you set the transitionCancellable property of the Navigator class to false (either programmatically or declaratively in XML) this method will never be called and you can leave the method body empty. In the case the property is set to true this method will be called if any of the goXXX methods of the Navigator class are called during a transition. The transition currently in progress will be cancelled and the new transition will start immediately. So you should use this method to reset or simply stop the running transition, e.g. setting alpha properties to 100 or stopping any animation.

Finally let's show an example for a simple fade transition:

```
import org.spicefactory.parsley.fx.TweenTask;
import org.spicefactory.parsley.task.events.TaskEvent;
import org.spicefactory.parsley.ui.Transition;
import org.spicefactory.parsley.ui.TransitionController;

class com.domain.package.FadeTransition implements Transition {
```

```
    private var _controller:TransitionController;
    private var _tween:TweenTask;
    private var _duration:Number;

    public function setDuration (duration:Number) : Void {
        // this is a property that you can set in XML
        _duration = duration;
    }

    public function startTransition
                                (controller:TransitionController) : Void {
        // do nothing here, just continue to load the next screen:
        controller.resume();
    }

    public function prepareScreen
                                (controller:TransitionController) : Void {
        // keep a reference to the controller that you need
        // in the onComplete event handler
        _controller = controller;

        // the tween function, you can also use functions of Macromedias
        // mx.transitions.easing package or Robert Penner's
        // easing equations
        var f:Function = function(t:Number, b:Number, c:Number, d:Number) {
            return c*t/d + b;
        };

        // the movie clip of the new screen should initialize with
        // an alpha set to 0, the final alpha value should be 100
        // the duration of the transition will be set in a property
        // the TweenTask class used here will be explained in section 8.3.
        var container:MovieClip
                                = controller.getNextScreen().getContainer();
        _tween = new TweenTask(f, container, "_alpha", 0, 100, _duration);
        _tween.addListener(TaskEvent.COMPLETE, this, onComplete);
        _tween.start();
    }

    private function onComplete (tween:TweenTask) : Void {
        // just call resume when the transition is complete:
        _tween = null;
        _controller.resume();
    }

    public function cancelTransition
                                (controller:TransitionController) : Void {
        // cancel the tween and reset the _alpha property to 100
        _tween.cancel();
        _tween = null;
        controller.getNextScreen().getContainer()._alpha = 100;
    }

}
```

Now you can add and configure your new transition in XML:

```
<navigator id="popup">
    <transition>
        <object class="com.domain.package.FadeTransition">
            <property name="duration">
                <number>25</number>
            </property>
        </object>
    </transition>
    <other child tags ...>
</navigator>
```

5.5. Integrating content from external SWF files

You saw how to configure screens in XML in section 3.14. Let's look at a simple example:

```
<screen name="myScreen" linkage="screen_A">
    <button name="submit" target="formController" method="submit"/>
</screen>
```

In this case the linkage attribute "screen_A" refers to a linkage indentifier of a movie clip in the library of the current SWF. If you want attach a movie clip of an external SWF just add a filename attribute:

```
<screen name="myScreen" linkage="screen_A" filename="external.SWF">
    <button name="submit" target="formController" method="submit"/>
</screen>
```

If the base-path attribute for the navigator was specified that path will be prepended to the filename. Now the linkage attribute "screen_A" points to a movie clip in the library of the external file.

Loading the file will be handled internally, you do not have to do anything. There are two things to keep in mind though:

If you added a transition to this navigator loading of the SWF and the transition will be "combined". This means that if you call controller.resume from within the startTransition method of the Transition interface before the external file was completely loaded, the next screen will not be initialized immediately. The controller will wait until loading is complete and controller.resume was called. The order of those two events does not matter.

Furthermore you may want to show something like a progress bar if you load a large file. You can add a listener to the Navigator instance with the addMovieLoaderListener method or you could do that in XML:

```
<navigator id="popup">
    <listener event="TaskEvent.COMPLETE" method="onComplete">
        <object-ref ref="progressController"/>
    </listener>
    <listener event="MovieLoaderEvent.PROGRESS" method="onProgress">
        <object-ref ref="progressController"/>
    </listener>
</navigator>
```

Note that the MovieLoader class used internally is a subclass of Task. It supports all the events for the Task class, i.e. the events defined in the TaskEvent class, plus an additional custom PROGRESS event defined in the MovieLoaderEvent class which is a subclass of the TaskEvent class. The listener method for the PROGRESS event receives the percentage of the SWF already loaded in the second parameter. A listener method might look like this:

```
public function onProgress (loader:MovieLoader, percent:Number) : Void
```

5.6. Controlling navigation with NavigationVoters

Sometimes you need more control over navigation logic than you get with adding event listeners. Under certain circumstances you may want to prevent navigation. You can use a `NavigationVoter` for this purpose. The `NavigationVoter` interface contains a single method:

```
public function vote (oldScreen:Screen, newScreen:Screen) : Boolean ;
```

Note that one of the two Screen arguments may be `null` if one of them was a blank screen. When this method returns `false` navigation will be prevented. You can add an unlimited number of `NavigationVoters` to a `Navigator` instance. If one of the voters returns false navigation will be prevented no matter what the other voters return. In addition to the voter instances that control a single `Navigator` instance you can add global voters that control all `Navigator` instances currently active. You can add global voters with a static method of the `Navigator` class:

```
Navigator.addGlobalVoter(new MySpecialVoter());
```

Although this feature might seem a bit esoteric at first glance it is very useful in many situations. If your application contains screens with large forms for example, you do not want the user to accidently click on a navigation button and loose all the content she edited. In those cases you should let one of your voters return false and open a popup message asking the user if she really wants to quit the current screen. If she clicks cancel nothing will happen. But if she clicks OK you may want to navigate to the screen the user originally clicked for. And the nice thing is you do not even have to remember which screen it was. If you call `Navigator.resume` it will navigate to the last target screen for which any voter prevented navigation. Just keep in mind that when you call `Navigator.resume` all registered voters will be invoked again so you have to set a flag somewhere in your voter to tell him to return true this time.

5.7. Destroying navigators

Finally you occasionally want to get rid of a `Navigator` instance. You can invoke the `destroy` method in those cases. Any currently active screen will be removed from the Flash stage and all `Screen` instances that were added to that `Navigator` will be destroyed. Destruction includes the removal of any component states that `Screen` or `View` instances of this `Navigator` managed. The `Navigator` class cannot be used any longer after you invoked the `destroy` method.

If you call `destroy` on an `ApplicationContext` instance all the `Navigator` instances managed by that `ApplicationContext` will be destroyed in turn.

6. Screens and Views

6.1. Preparing Screens and Views in the Flash IDE

Configuration of `Screen` and `View` instances was explained in section `3.14`. This section shows you how to prepare those assets in the Flash IDE.

Preparing Screens

Screens will always be attached from the library. Each screen you use will be associated with a single movie clip in the library. You have to specify a linkage identifier for each of those clips in the IDE and use the same identifier for the `linkage` attribute in the XML configuration:

```
<screen name="myScreen" linkage="linkageIdentifierInLibrary"/>
```

The assets that you include in that symbol may be nested views, components or text fields. If you want to add those assets dynamically in your application code and not in the authoring environment, you could also leave the clip empty. Or alternatively include only certain assets in the IDE and add the remaining ones dynamically. The movie clip containing your screen assets must not be associated with an AS2 class.

Preparing nested Views

Nested views are essentially collections of other nested views, components and text fields, that you may want to reuse on different screens. You prepare them in the same way that you prepare Screens with only one difference: The clip containing your view assets must be associated with the `ViewContainer` class of the `parsley.comp` package. This class will make sure that the view registers itself with its parent view or screen in its `onLoad` event.

Preparing components

A single component is always associated with an AS2 class. Either one of the `parsley.comp` package or subpackages or a custom class that you implemented and that extends the `Component` class of the `parsley.comp` package. If you add a component to the screen or view clip in the authoring environment, you must not specify a `linkage` attribute in the `component` tag. Just make sure that the `name` attribute refers to the instance name that you assigned to that component when you placed it on the stage:

```
<component name="theInstanceNameAssignedInTheIDE" class="CheckBox"/>
```

If you intend to add that component dynamically you must specify the `linkage` attribute:

```
<component name="arbitrary" linkage="linkageIdentifierInLibrary"
           class="CheckBox"/>
```

Now the `name` attribute does not point to an instance name you specified in the IDE (you did not place the movie clip on the stage anyway). The `name` attribute will be used by the framework to assign that name as an instance name to the clip when you dynamically attach that component to a screen. Section `6.3.` shows you how to do that.

6.2. Programmatic vs. declarative configuration

Section 3.14. described how to configure `Screens` and `Views` in XML. Like almost all the features that can be configured in XML, the `Screen` and `View` instances can alternatively be configured programmatically. Let's compare both approaches with a simple example:

Configuration in XML:

```
<screen name="myScreen" linkage="linkageID">
    <component name="box" class="CheckBox"/>
</screen>
```

Programmatic configuration:

```
import org.spicefactory.parsley.ui.Screen;
import org.spicefactory.parsley.config.tree.ClassConfig;
import org.spicefactory.parsley.config.tree.ComponentConfig;
import org.spicefactory.parsley.comp.CheckBox;

[...]

var scr:Screen = new Screen();
scr.setName("myScreen");
scr.setLinkage("linkageID");
var config:ComponentConfig = new ComponentConfig();
config.setName("box");
var className:String = "org.spicefactory.parsley.comp.CheckBox";
var clazz:ClassConfig = new ClassConfig(className, CheckBox);
config.setClass(clazz);
screen.addComponent(config);
```

As you see XML configuration is pretty straightforward whereas programmatic configuration is rather cumbersome. So in most cases XML configuration is the recommended approach. Only in cases where you must defer configuration until the application is loaded because it may depend on user interaction or the state of the application, programmatic configuration is available as a second option.

6.3. Adding components and subviews dynamically

When you define a component or nested view to be added dynamically the corresponding XML configuration might look like this:

```
<navigator id="test">
    <screen name="myScreen" linkage="anEmptyClip">
        <component name="box" linkage="box_1" class="CheckBox"/>
    </screen>
</navigator>
```

In this case the movie clip with the linkage identifier "`anEmptyClip`" should indeed be empty because the screen only contains a single component that has its `linkage` attribute specified. That means that the framework will expect that the component was not added to the screen in the IDE and you intend to do that dynamically. To do that you need a reference to the screen instance:

```
var context:ApplicationContext = getContextSomehow();
var nav:Navigator = context.getNavigator("test");
var container:MovieClip = getTheContainerForThisNavigator();
nav.activate(container.createEmptyMovieClip("navi", 1);
nav.goFirst();
var scr:Screen = nav.getCurrentScreen();
```

Now you can later attach the component you configured in XML to that screen and set its position:

```
var comp:Component = scr.createElement("box");
comp._x = 25;
comp._y = 100;
```

You can also dynamically remove that component from the screen like this:

```
scr.removeElement("box");
```

If you know that you won't reuse that component later you can also destroy it:

```
scr.destroyElement("box");
```

After the element was destroyed you can no longer attach it to the screen, whereas after calling `removeElement` you can attach it again later with the `createElement` method and if the component had its `stateful` atrribute set to true it will initialize to the state it had when you removed it.

Note that you should not create and remove a component within a single frame as this might confuse the internal event handling of the framework.

Handling for nested views is essentially the same. The given examples also apply for Views.

6.4. The View lifecycle events

The `ViewEvent` class fires five different lifecycle events. They apply to `Screen` instances as well as to View instances. Remember that `Screen` is just a simple subclass of `View`. You can register listeners for those events in XML configuration or programmatically.

`ViewEvent.PREPARE`

This event will always fire when the screen or view was attached but before the `onLoad` event for nested components and views fired.

`ViewEvent.ENTER`

This event will fire when the screen or view was completely initialized, e.g. all `onLoad` events for nested views and components have fired.

`ViewEvent.UPDATE`

If you dynamically add components or subview to a screen or view, this event will fire after all `onLoad` events for all added views and components have fired.

`ViewEvent.EXIT`

Will fire before the screen or view will be removed from the stage. You have still access to components and subviews during that event.

`ViewEvent.DESTROY`

Will fire when the screen or view gets destroyed. You cannot use the `Screen` or `View` instance after that event.

An example XML configuration for a listener might look as follows:

```
<screen name="myScreen" linkage"linkageID">
    <listener event="ViewEvent.ENTER" method="onEnter">
        <object-ref ref="controller"/>
    </listener>
</screen>
```

You could also add the listener programmatically from within your controller class:

```
var scr:Screen = getScreenSomehow();
scr.addListener(ViewEvent.ENTER, this, onEnter);
```

6.5. Handling text

You saw how to configure text fields in section `3.16`. Now let's have a look at how you could dynamically modify them in application code.

To read text from a field use the `getText` method of the `Screen` or `View` class:

```
var scr:Screen = ...;
var text:String = scr.getText("myField");
```

The `myField` parameter points to the `path` attribute that you specified in XML:

```
<text path="myField"/>
```

To change the contents of the text field use the `setText` method:

```
var scr:Screen = ...;
scr.setText("myField", "Hello World");
```

To set the focus to a particular field use the `setFocus` method:

```
var scr:Screen = ...;
scr.setFocus("myField");
```

If you need full control over the text field you can obtain a reference to the `TextField` instance:

```
var scr:Screen = ...;
var tf:TextField = scr.getField("myField");
doSomethingUsefulWith(tf);
```

7. The component framework

7.1. Overview

The `parsley.comp` package contains a lot of standard UI components. They all extend the base `Component` class. You saw how to configure components in XML in section `3.15`. This chapter shows you how to prepare each of the components in the IDE and how to use them in your application code. This section gives a quick overview over the available components:

Component	The base component class.
SimpleButton	A button with rollover, pushed and disabled states.
DraggableButton	A subclass of SimpleButton with added Drag'n'Drop functionality.
ChechBox	A subclass of SimpleButton with additional selected states.
Slider	A vertical or horizontal slider component.
SnapSlider	A subclass of Slider with fixed snap positions.
Scrollbar	A scrollbar consisting of nested Slider and SimpleButton components.
TextArea	A scrollable text field with a nested field an scrollbar component.
ScrollPane	A masked movie clip with a nested scrollbar component.
Window	A draggable window component.
SelectionGroup	A group of selectable elements like radio buttons.
SelectionList	A scrollable SelectionGroup with an associated DataSource.
Menu	A component with nested SelectionList and SimpleButton components.
ComboBox	A subclass of Menu with an additional nested display element.

7.2. Key differences to the Macromedia v2 components

The design of the Parsley component framework was driven by rather unusual project requirements. For example I had to develop a kind of DataGrid component with irregularly shaped row elements. With such requirements the way you skin components with Macromedias v2 components is not sufficient. I needed full control over the size and position of each sub element in the DataGrid component. So I used a very different approach. I did not come up with a set of ready to use components with default skins which you can drag from the component palette to the stage. I just developed a set of component classes which you can assign to your movie clips in the library. This way every component has to be built by hand. You follow certain rules which frames and which subcomponents each of the components

classes expect, but the positioning of all elements is done in the IDE and not through dynamic attaching during runtime.

Of course this approach is rather cumbersome if you do not need full control over positioning but rather need a set of ready to use components with deafault skins, e.g. for prototyping your UI. So the parsley.comp package is very likely to change a lot for version 1.0. I am in the process of evaluating ways to offer both approaches within the same package.

Another difference between the Parsley components and the Macromedia v2 components is that the Parsley components are more lightweight. They do not add that many kilobytes to your SWF like the Macromedia ones. On the other hand they currently lack some of the features of the v2 components, especially focus management and dynamic resizing of components.

7.3. Preparing components in the IDE

Working with Parsley components basically involves three steps:

1. Preparing the components in the Flash IDE
2. Configuring the components in XML
3. Using the components in your application code.

Step 2 was explained in section `3.15`. Step 1 and 3 will be explained for each component in the remaining sections of this Chapter. The following sections will each describe a single component and start with a quick summary of the requirements for building the movie clip for that component, listing all required frames and child components.

To facilitate the process of preparing the components Parsley comes with a small set of JSFL actions. If you copy them to the Commands directory in your Flash installation directory they will be available in the Commands menu in the IDE. Most of the commands simply create a new layer and insert the required frame labels for a particular component. Make sure that a the timeline of the movie clip that you want to prepare is the currently active timeline and then choose one of the Commands from the menu:

Component	Just inserts a "normal" and "disabled" frame. Can be used with every component that does not require additional frames.
SimpleButton	Inserts "normal", "over", "pushed" and "disabled" frames. Can be used with SimpleButton or DraggableButton components.
SelectionElement	Inserts "normal", "over", "pushed", "selected" and "disabled" frames. Can be used with the SelectionElement component which is a child component for DisplayGroup or SelectionGroup.
CheckBox	Inserts "normal", "over", "pushed", "selected", "disabled", "over_selected", "pushed_selected" and "disabled_selected" frames. Can be used with CheckBox components.

Most other components do not expect a special set of frame labels.

In addition to the commands which simply create a new layer with required frame labels there are two more commands which help you manage components:

The `AnalyzeClasses` command gives you an overview over all the movie clips in the library that have AS 2 classes assigned. It's divided into two sections. First all the components that belong to the Parsley framework are listet. The second section lists all your custom components. The list be written to the Flash Output window.

The `AssignClasses` command helps you assign the AS 2 classes to the movie clips in the library. Especially the Parsley components have a rather long package namespace which you would not want to type too often. With the help of this command it is sufficient to append a particular suffix to the name of the movie clip in the library. When you run this command it will find all clips with one of those special suffices and automatically assign the required class. For example a movie clip with the name `submit_button` will be assigned to the `SimpleButton` class with a linkage identifier consisting of the base movie clip name without the suffix. Just keep in mind that the command will ignore all movie clips that already have either a linkage identifier or an AS 2 class assigned.

Here is a complete list of available suffices:

comp	Component
button	SimpleButton
drag	DraggableButton
chbox	CheckBox
slider	Slider
snap	SnapSlider
scrbar	Scrollbar
scrpane	ScrollPane
area	TextArea
progr	ProgressBar
window	Window
view	ViewContainer
mask	Mask
cobox	ComboBox
menu	Menu
dgroup	DisplayGroup
sgroup	SelectionGroup
dlist	DisplayList
slist	SelectionList
delem	DisplayElement
selem	SelectionElement

7.4. The Component base class

The `Component` class in the `parsley.comp` package is the base component for all Parsley components. If you develop your own components they must extend this class either directly or indirectly if you want to use them with the Parsley UI and configuration framework.

It contains some basic functionality that all Parsley components share. It uses the `onLoad` and `onUnload` events of the MovieClip class which it extends to register and unregister with the Parsley UI framework. Thus you should never overwrite those methods. How to extend this class for your own components is explained in section `7.23`.

Furthermore it contains methods for handling tooltips (section `7.20`.) and double click detection (section `7.19`.), to enable or disable the component, hooks into state management for components (section `7.21`.) and a bunch of private methods for configuring the component programmatically in your own `Component` subclasses, i.e. telling the component which child components it should expect when it loads.

Though you would usually assign a concrete subclass of `Component` to your movie clips, sometimes it may be sufficient to assign the base class, if all your component needs to support are tooltips, enabling and disabling the component or handling mouse events.

7.5. Buttons

Requirements for the `SimpleButton` component:

Expected frames: normal, over, pushed, disabled

Expected children: none

This component is pretty straightforward. The four frame labels enable you to build the component within a single timeline. You will most likely never use the component in your application code. The most convenient way to use this component is with the special `button` tag which lets you bind the `RELEASE` event to a method in one of your controller classes:

```
<button name="submit" target="formController" method="submit"/>
```

7.6. Drag and Drop

Requirements for the `DraggableButton` component:

Expected frames: normal, over, pushed, disabled

Expected children: none

`DraggableButton` is a subclass of `SimpleButton`. It adds drag'n'drop functionality. It supports the three events `START`, `MOVE` and `STOP` of the `DragEvent` class. You can register listeners for those events in XML:

```
<component name="dragMe" class="DraggableButton" stateful="true">
    <listener event="DragEvent.START" method="onStartDrag">
        <object-ref ref="dragController"/>
```

```
        </listener>
        <listener event="DragEvent.MOVE" method="onMove">
            <object-ref ref="dragController"/>
        </listener>
        <listener event="DragEvent.STOP" method="onStopDrag">
            <object-ref ref="dragController"/>
        </listener>
    </component>
```

If this component is configured with the `stateful` attribute set to true it will remember its position on the screen when you exit and reenter the screen or view that component belongs to.

7.7. CheckBox

Requirements for the `CheckBox` component:

Expected frames: normal, over, pushed, disabled, selected, over_selected, pushed_selected, disabled_selected

Expected children: none

`CheckBox` is a subclass of `SimpleButton`. It adds four additional states. There are two ways to bind the component to a controller instance. First you could listen to `SelectionEvent.CHANGE`:

```
<component name="box" class="CheckBox" stateful="true">
    <listener event="SelectionEvent.CHANGE" method="onChange">
        <object-ref ref="formController"/>
    </listener>
</component>
```

But often you do not have to handle each selection event for each checkbox separately, you only need to be able to find out the current state of the checkbox when you submit the form for example. In those cases it makes more sense to bind a reference to the checkbox instance to your form controller like this:

```
<component name="box" class="CheckBox" stateful="true">
    <binding target="formController" property="checkbox"/>
</component>
```

If you bind a property like this it will be set to a reference of the `CheckBox` component during the `onLoad` event of the component and it will be reset to undefined in the `onUnload` event of the component. There are different ways to define properties (see section `3.12`.) One option is to add a setter method like this to your controller class:

```
public function setCheckbox (box:CheckBox) : Void {
    // keep a reference in a private var:
    _box = box;
}
```

When the submit button for that form is bound to the `submit` method in that controller class, you have access to the checkbox from within that method:

```
public function submit (button:SimpleButton) : Void {
    // read and process the current state of the checkbox:
    var checked:Boolean = _box.isSelected();
    if (checked)  {
        ...
    }
}
```

There are even more options to obtain a reference to the component, the remaining ones are described in section `7.22`.

7.8. Slider and SnapSlider

Requirements for the `Slider` and `SnapSlider` components:

Expected frames:	normal, disabled	
Expected children:	name	type
	dragger	DraggableButton
	bar	Component

This component can be used for vertical or horizontal sliders. `SnapSlider` is a subclass of `Slider` that adds the option to define fixed snap points for the slider.

It expects two child components. The dimensions of the component with the instance name `bar` are used to determine the range that the draggable button can be moved. When the component loads, the top or left border of the draggable button will be aligned with the top or left border of the bar component, depending on whether it is a vertical or horizontal slider. You should keep that in mind when you prepare the components in the IDE.

The `Slider` supports the three events `START`, `MOVE` and `STOP` of the `SliderEvent` class. You can register listeners for those events in XML:

```
<component name="slider" class="Slider" stateful="true">
    <listener event=" SliderEvent.START" method="onStart">
        <object-ref ref="myController"/>
    </listener>
    <listener event=" SliderEvent.MOVE" method="onMove">
        <object-ref ref="myController"/>
    </listener>
    <listener event=" SliderEvent.STOP" method="onStop">
        <object-ref ref="myController"/>
    </listener>
</component>
```

If this component is configured with the `stateful` attribute set to true it will remember its position when you exit and reenter the screen or view the slider belongs to.

The SnapSlider component adds the setSnapPoints and snapTo methods to handle fixed positions for the slider. Of course you can also set the snapPoints property in XML:

```
<component name="snapSlider" class="SnapSlider" stateful="true">
    <property name="snapPoints"><number>10</number></property>
</component>
```

7.9. Scrollbar

Requirements for the Scrollbar component:

Expected frames:	normal, disabled	
Expected children:	name	type
	up	SimpleButton
	down	SimpleButton
	slider	Slider

You can omit some of the expected children. Since Flash designers tend to try to reinvent scroll functionality for each project they sometimes leave out either the slider or the up and down buttons. The Parsley Scrollbar will work in any of those reduced modes without modification or configuration.

It supports a lot of events you could listen to: START, STOP, DRAG, DOWN and UP, defined in the ScrollEvent class. You will rarely use those events directly. In most cases you will use one of the Parsley components which expect a scrollbar as a child component like TextArea, ScrollPane and DisplayList. Those components handle all scroll events themselves.

7.10. TextArea

Requirements for the TextArea component:

Expected frames:	normal, disabled	
Expected children:	name	type
	field	TextField
	scrollbar	Scrollbar

The TextArea component has several methods to read and modify the text. The methods setHtml, setText and appendText can be used to modify the text. The methods getText and getHtml can be used to read text.

Usually you'll want to bind the TextArea instance to a property of a controller instance. Binding components is explained in section 7.22.

7.11. ScrollPane

Note: This component is kind of deprecated.

Requirements for the `ScrollPane` component:

Expected frames:	normal, disabled	
Expected children:	name	type
	scrollbar	Scrollbar
	content	MovieClip
	mask	MovieClip on mask layer

This component is pretty straightforward. The functionality has not been changed or improved since Parsley 0.6, thus it is very likely that this component will be completely refactored for version 1.0. It's possible that it will be removed completely and the functionality instead be integrated into the `View` or Screen classes in the `parsley.ui` package. A disadvantage of the component in its current state is that the content of the pane is a simple `MovieClip`. If this `MovieClip` instance contains other Parsley components they would not know where and how to register with the framework.

7.12. Window

Note: This component is kind of deprecated.

Requirement for the `Window` component:

Expected frames:	normal, disabled	
Expected children:	name	type
	titleBar	SimpleButton
	closeButton	SimpleButton

This component can be used as a draggable window with arbitrary content. The functionality has not been changed or improved since Parsley 0.6, thus it is very likely that this component will be completely refactored for version 1.0. It's possible that it will be removed completely and the functionality instead be integrated into the `View` or `Screen` classes in the `parsley.ui` package. A disadvantage of the component in its current state is that the content of the window is not handled explicitly. If this `Window` instance contains other Parsley components they would not know where and how to register with the framework.

7.13. Popup Messages

In many applications you need a lot of simple popup messages with a small set of buttons like OK and Cancel with customised button actions for each message type. It would be very cumbersome to develop this kind of Screens like you would do normally: Defining each of the messages and each of the button actions in XML. To facilitate this procedure Parsley comes with a special package `parsley.comp.popup` that provides some shortcuts for this kind of

functionality. Using Parsley Popup Messages involves 4 steps: XML Configuration, Preparing message screens in the IDE, Registration and the actual creation of a popup message:

Configuration

The following example shows a configuration snippet for a popup window with three types of messages:

```
<navigator id="popupMessages">
    <screen name="ok" linkage="popup_ok" class="PopupMessageScreen" />
    <screen name="ok_cancel" linkage="popup_ok_cancel"
                                        class="PopupMessageScreen" />
    <screen name="yes_no_cancel" linkage="popup_yes_no_cancel"
                                        class="PopupMessageScreen" />
</navigator>

<navigator id="popup">
    <screen name="popupScreen" linkage="popupScreen">
        <component name="window" class="PopupWindow">
            <property name="navigator">
                <navigator-ref ref="popupMessages"/>
            </property>
        </component>
    </screen>
</navigator>
```

As you can see in this example you need to define two navigators. The first one is the navigator containing the actual message screens. In this case there are three different screens each containing a different set of buttons which is reflected in the name of the screen. Note that in contrast to ordinary screen configuration there are two differences: You specify a special class as the Screen implementation which handles registering of the buttons and the field containing the message. Thus the second difference is that you do not have to configure the individual buttons and text fields for each screen like you normally do. The second navigator is the one containing the popup window which has to be of type `PopupWindow` and which in turn contains the nested navigator with the actual messages. This way you can combine different window types with different types of messages.

Preparing message popups in the IDE

You can prepare the screen containing the window the same way as you prepare normal screens. Just make sure that you assign the `PopupWindow` class to your Window component. The screens containing the messages require special instance names: The text field for the message needs to have the name `message` assigned, the button names must be `button_0`, `button_1` and so on.

Registration

After loading the ApplicationContext containing the configured popup messages you need to activate the navigators and register the screen containing the window component with the static PopupMessage class:

```
var nav:Navigator = context.getNavigator("popup");
nav.activate(any_mc.createEmptyMovieClip("myPopup", 1));
PopupMessage.register(nav.getScreenByName("popupScreen"));
```

The second statement is just the usual way to activate a navigator which was explained in section 5.1. So the interesting part is the third statement which makes the popup window available for the `PopupMessage.show` method which is explained in the next section.

Displaying Popup Messages

Now everything is set up to actually show the messages when needed. Example code might look like this:

```
var commands:Array = new Array();
commands.push(new Command(this, onOK, ["anArgument", 0]));
commands.push(new Command(this, onCancel, []);
PopupMessage.show("popupScreen", "ok_cancel", commands, "msg.timeout");
```

First you create an array of commands in the order your buttons are numbered on the screen. Note that you do not need to explicitly close the window, this will be handled automatically. Thus when you have a Cancel button for example that does nothing but close the window you can simply add null to the array:

```
var commands:Array = new Array();
commands.push(new Command(this, onOK, ["anArgument", 0]));
commands.push(null);
PopupMessage.show("popupScreen", "ok_cancel", commands, "msg.timeout");
```

The final step is to invoke the static show method of the `PopupMessage` class. The first argument is the name of the screen containing the window that you registered with the `PopupMessage` class. The second argument is the name of the screen containing the actual message and buttons. Make sure not to confuse these two parameters. The third parameter is the array of commands to be assigned to the buttons. The fourth parameter is the key for a message in a message bundle. In the example the default message bundle would be used, but you can pass a different bundle name as the optional fifth argument.

7.14. DisplayGroup and SelectionGroup

Requirements for the `DisplayGroup` component:

Expected frames:	normal, disabled	
Expected children:	name	type
	element_0	DisplayElement
	to element_N	

Requirements for the `SelectionGroup` component:

Expected frames:	normal, disabled	
Expected children:	name	type
	element_0	SelectionElement
	to element_N	

This component can be used to build interface elements like radio button groups. It contains a collection of nested DisplayElements and comes in two flavours: SelectionGroup allows the nested elements to be alternatively selected whereas DisplayGroup does not support Selection and is rarely used as a standalone component. It is most likely used nested within a DisplayList component which will be described in section `7.15`.

This component expects an unlimited number of nested `DisplayElement` or `SelectionElement` components respectively. The first one should have an instance name of `element_0`, the remaining elements should increment the number suffix. This way the parent component will automatically detect the number of elements. The element components themselves are described in section `7.16`.

The `SelectionGroup` component supports the `SelectionEvent.CHANGE` event. You can register controller instances as listeners for that event in XML:

```
<component name="group" class="SelectionGroup" stateful="true">
    <listener event="SelectionEvent.CHANGE" method="onChange">
        <object-ref ref="formController"/>
    </listener>
</component>
```

Whenever the user selects a different element this event fires. Alternatively you can set the selection programmatically:

```
var group:SelectionGroup = getReferenceSomehow();
group.setSelectedIndex(3);
```

The component will pass a boolean parameter to the object listening to the `CHANGE` event indicating if the selection changed due to an UI event (i.e. the user clicked on one of the elements) or if it was changed programmatically. The signature of a listener method might look as follows:

```
public function onChange (group:SelectionGroup, uiEvent:Boolean) : Void
```

Finally you can use the `getSelectedIndex` or `getSelectedModel` methods to get information about the currently selected element.

7.15. DisplayList and SelectionList

Requirements of the `DisplayList` component:

Expected frames:	normal, disabled	
Expected children:	name	type
	displayGroup	DisplayGroup
	scrollbar	Scrollbar

Requirements of the `SelectionList` component:

Expected frames:	normal, disabled	
Expected children:	name	type
	selectionGroup	SelectionGroup
	scrollbar	Scrollbar

These components contain nested DisplayGroup or SelectionGroup components. They add scrolling functionality and the `DataSource` interface for additional flexibility.

The DataSource implementation that you can configure for this component helps you to decouple the UI element from the data that should be displayed. It is explained in section `7.17.` alongside example configuration snippets for configuring `DisplayLists` and `SelectionLists`.

Which of these components you should use depends on whether the individual elements should be selectable or not. If you choose the SelectionList component keep in mind that the element components handle all mouse events. That means that you cannot have an editable field or other subcomponents that handle mouse events within the `SelectionElement` components. If you need this kind of functionality, for example to build an editable datagrid component, you should choose `DisplayList` and do without the ability to select individual rows as a whole.

7.16. DisplayElement and SelectionElement

Requirements for the nested `DisplayElement` component:

Expected frames:	normal, disabled
Expected children:	none

Requirements for the nested `SelectionElement` component:

Expected frames:	normal, over, pushed, disabled, selected
Expected children:	none

The DisplayElement or SelectionElement handle the state of the element and mouse events, e.g. they might change the background color if the element is selected or pushed. To actually display the data of the model object associated with that element (if any) you have to use a DataRenderer described in section 7.17.

When you build the DisplayElement or SelectionElement components in the Flash IDE you need to add all the required frames for handling the state of the element as well as all the UI elements needed to display the data, usually spanning all frames. A DataRenderer implementation will then use these UI elements to actually display the associated data, usually on top of the background elements that indicate the state of the element.

7.17. DataRenderer and DataSource

These two interfaces keep the Parsley data components flexible, as they allow you to reuse all the functionality for selections, organizing model objects or scrolling built into the DisplayList or SelectionList components, but add custom functionality for obtaining and displaying data.

DataRenderer

DataRenderer implementations are responsible for rendering the data for their associated model objects in a DisplayElement or SelectionElement component. It only contains a single method:

```
public function draw (element:DisplayElement) : Void ;
```

The DisplayElement instance passed to that method gives you access to all the UI elements that you might need for displaying the model data. This method will be called whenever the component needs to be redrawn. You can extract the actual model object with the getModel method of the DisplayElement class and you can get references for nested components with the getChild method of the base Component class. That should give you everything you need to create the view for the model object. Just keep in mind that you have to develop stateless DataRenderer implementations as a single instance will be used for all DisplayElements or SelectionElements within a single DisplayGroup/List or SelectionGroup/List.

When the display logic for that element is simple, e.g. it just displays properties of the associated model objects in text fields, you could use PropertyBindingRenderer, the only concrete DataRenderer implementation that comes with Parsley. This implementation expects fields with instance names corresponding to property names of the associated model object with the prefix bind_. That means if the DisplayElement that is passed to the draw method contains a field with an instance name of bind_username, it expects that the associated model has either a getUsername method or a username property and displays that property in the field, converting it to a String if necessary. There is one special reserved name, bind_toString, which will not read a single property, but instead call toString on the model object itself and display that value. This is especially useful if you use simple Strings as model objects.

Sometimes you want to reuse that logic for several fields in your element component but need to add custom functionality for rendering non-textual representations of properties. In this case you can subclass PropertyBindingRenderer and add that logic to the draw method.

The following example shows a simple subclass that attaches a flag symbol corresponding to a country property of the model object in addition to delegating to the super class for rendering the remaining properties in simple text fields:

```
import org.spicefactory.parsley.comp.data.impl.PropertyBindingRenderer;

class com.domain.package.MyRenderer extends PropertyBindingRenderer {

    private var _flag:MovieClip;

    public function draw (element:DisplayElement) : Void {
        // first call the super class to draw all fields which have
        // instance names corresponding to model properties:
        super.draw(element);

        // now get a reference to the associated model object
        // and cast it to the expected type:
        var model:Person = Person(element.getModel());

        // if it is null (may happen for empty elements)
        // remove any previously attached flag symbol:
        if (model == null && _flag != null) {
            _flag.removeMovieClip();
            _flag = null;
            return;
        }

        // now attach a flag symbol corresponding to the
        // country property of that Person object
        var country:String = model.getCountry();
        _flag = this.attachMovie("flag", "flag", 1);
    }

}
```

DataSource

`DataSource` implementations handle adding, removing and sorting of model objects and dispatching the corresponding events. In many cases `ListDataSource`, the only concrete implementation that comes with Parsley, will be sufficient. It wraps a normal `List` instance of the `parsley.collection` package and intercepts all method calls that will alter the list contents. If you create that `DataSource` programmatically, example code might look like this:

```
var dList:DisplayList = getReferenceSomehow();
var data:List = new ArrayList();
data.add(new Person("Pia", 23, "Manchester"));
data.add(new Person("Shirley", 31, "Birmingham"));
dList.setDataSource(new ListDataSource(dList));
```

If you want to modify the list later, just use it as usual. The wrapping `DataSource` implementation will detect changes and pass them to the `DisplayList` or `SelectionList` components.

Alternatively, especially for simple data models, you could configure `DataSources` in XML:

```
<object id="comboBoxData" class="ListDataSource">
    <constructor-args>
        <list>
            <string>England</string>
            <string>France</string>
            <string>Germany</string>
            <string>Spain</string>
        </list>
    </constructor-args>
</object>

<navigator id="main">
    <screen name="chooseCountry" linkage="chooseCountry_screen">
        <component name="combo" class="ComboBox">
            <property name="dataSource">
                <object-ref ref="comboBoxData"/>
            </property>
        </component>
    </screen>
</navigator>
```

First you configure the `ListDataSource` instance like any other object, specifying the model as a constructor argument. Then you could use that `DataSource` for any component like the `ComboBox` in the example above.

For more sophisticated behaviour you may want to create your own `DataSource` implementations, for example for loading data from a server with Flash Remoting.

7.18. Menu and ComboBox

Requirements for the `Menu` and `ComboBox` components:

Expected frames:	normal, disabled	
Expected children:	name	type
	list	SelectionList
	openButton	SimpleButton
	displayElement	DisplayElement or SelectionElement (ComboBox only)

The `Menu` component contains a nested `SelectionList` component and a button which will show the nested list when clicked. The `ComboBox` component is a subclass of `Menu` and adds a `DisplayElement` child component which renders the model of the selected element.

The `getList` method returns a reference to the nested `SelectionList` component which you can manipulate in the same way like a standalone `SelectionList` component. See sections `7.14.` through `7.17.` for details about the `SelectionList` component.

The default implementation just switches the `_visible` property of the nested `SelectionList` component when the `openButton` is clicked. If you want to animate it (e.g. fading in or

moving from behind a mask) you have to subclass the `Menu` or `ComboBox` class and overwrite `openList` and `closeList`.

7.19. Mouse Events

The `Component` base class contains methods to add listeners for mouse events. This is the recommended approach for all Parsley components. Traditionally you would set the event listener method on the component like this:

```
component.onRelease = function () { ... }
```

But this approach has several disadvantages:

1. You cannot add another listener elsewhere in your code. Setting the `onRelease` function will always overwrite the previous one.
2. There are no builtin events for double clicks.
3. You cannot use this approach when you want to configure listeners in Parsley's XML configuration.

So instead you would add a listener for the `RELEASE` event like this:

```
component.addMouseListener(MouseEvent.RELEASE, this, onRelease);
```

Or, if you want to configure the listener in XML it might look as follows:

```
<component name="aName" class="Component">
    <listener event="MouseEvent.RELEASE" method="onRelease">
        <object-ref ref="myController"/>
    </listener>
</component>
```

In addition to the standard events like `PRESS`, `RELEASE` or `ROLL_OVER` the MouseEvent class supports two special events: `DOUBLE_CLICK` and `SINGLE_CLICK`. There is a subtle difference between a `SINGLE_CLICK` and a `RELEASE` event: The `RELEASE` event fires every time the mouse button is released, i.e. it fires twice for every `DOUBLE_CLICK` event. The `SINGLE_CLICK` event only fires if the mouse button was released and the time span for a `DOUBLE_CLICK` event elapsed without the mouse button being pressed a second time.

This is the same behaviour that you can observe when you click on a file name in the Windows Explorer to rename it. It does not activate the file name for editing immediately, it waits for several milliseconds to see if your intention was a double-click instead.

7.20. Tooltips

To add internationalized tooltips to your components just add the corresponding attribute to the component or button tag in the XML configuration file:

```
<component name="box" class="CheckBox" tooltip="editor.checkbox"/>

<button name="submit" target="formController" method="submit"
        tooltip="form.submit"/>
```

The specified `tooltip` attribute refers to a key for a message in the default bundle for this `ApplicationContext`. If you want to use a message from a different bundle you can add a `bundle` attribute to the component or button tag.

To configure the visual appearance of the tooltips you can create a `TooltipConfig` object either programmatically or declaratively in XML:

```
<object id="tooltipConfig" class="${parsley}.comp.util.TooltipConfig">
    <property name="backgroundColor">
        <number>0xFF0000</number>
    </property>
    <property name="backgroundAlpha">
        <number>70</number>
    </property>
    <property name="borderColor">
        <number>0x000000</number>
    </property>
    <property name="borderWidth">
        <number>1</number>
    </property>
    <property name="padding">
        <number>2</number>
    </property>
    <property name="font">
        <string>Verdana</string>
    </property>
    <property name="fontSize">
        <number>10</number>
    </property>
    <property name="delay">
        <number>500</number>
    </property>
</object>
```

Finally you have to set the static `defaultConfig` property of the tooltip class to the config instance you created:

```
<static-initializer class="${parsley}.comp.Tooltip>
    <property name="defaultConfig">
        <object-ref ref="tooltipConfig"/>
    </property>
</static-initializer>
```

7.21. State Management

You already saw the `stateful` attribute for `navigator`, `screen`, `view` and `component` tags in several examples. This sections delves into the details of state management. Let's begin with a quick overview of the configuration options. A simple example configuration might look as follows:

```
<navigator id="navi" stateful="false">
    <screen name="first" linkage="screen_1" stateful="true">
        <component name="box" class="CheckBox"/>
    </screen>
    <screen name="second" linkage"screen_2">
        <component name="combo" class="ComboBox" stateful="true"/>
        <component name="area" class="TextArea">
    </screen>
</navigator>
```

The top level `navigator` tag has a `stateful` attribute set to false. That means that all components on all screens and nested views of that navigator instance do not remember their state unless those screens, views or component overwrite that attribute. In the example above the first screen tag overwrites the `stateful` attribute so that the component named box will remember its state. Note that in this case you do not need to set the attribute in the component tag itself, it will be inherited from the screen tag.

The second screen does not specify a `stateful` attribute, thus it inherits the false value from the navigator tag. The component named combo overwrites that value and will remember its state. The component named area does not specify a `stateful` attribute and inherits the false value from the top level `navigator` tag.

But what exactly does that mean, "the components remember their state"? This is handled by each component individually. For a `DraggableButton` component it means that it remembers its position on the stage. For a `CheckBox` component it means that it will remember whether it was selected or not. For a `SelectionList` component it means that it will remember the list of associated model objects, the scroll position and which element was the last selected.

If you write your own components and want to participate in the Parsley state management you ususally just have to overwrite two simple methods in your component class. This is explained in section `7.23.`

7.22. Binding components to controller classes

If you configure `navigator`, `screen`, `view`, `component`, `button` and `text` elements in XML, this view layer will be separate from the controller and model parts of your application unless you bind several of those elements to controller classes. There are basically three options to connect the controller and view part of your application:

1. Adding listeners to navigator, screen, view or component instances
2. Binding references to screen, view or component instances to properties of a controller instance
3. Obtaining a reference to an UI element programmatically

Let's demonstrate all three approaches:

Adding listeners to navigator, screen, view or component instances

```
<navigator id="navi">
    <listener event="NavigatorEvent.ENTER_SCREEN"
                                                method="onEnterScreen">
        <object-ref ref="myController"/>
     </listener>
     <screen name="first" linkage="screen_1">
         <component name="box" class="CheckBox">
             <listener event="SelectionEvent.CHANGE" method="onChange"
                 <object-ref ref="myController"/>
             </listener>
         </component>
     </screen>
     <screen name="second" linkage="screen_2"/>
</navigator>
```

In the example above the controller instance with the configuration id `myController` will listen to the `ENTER_SCREEN` event of the navigator and to the `CHANGE` event of the checkbox on the first screen.

Binding references to screen, view or component instances to properties of a controller instance

```
<component name="box" class="CheckBox">
    <binding target="myController" property="checkbox"/>
</component>
```

If you bind a property like this it will be set to a reference of the `CheckBox` component during the `onLoad` event of the component and it will be reset to undefined in the `onUnload` event of the component. There are different ways to define properties (see section `3.12.`) One option is to add a setter method like this to your controller class:

```
public function setCheckbox (box:CheckBox) : Void {
    // keep a reference in a private var:
    _box = box;
}
```

Obtaining a reference to an UI element programmatically

If you implement controller logic for a screen with a lot of components you may prefer to obtain references to components programmatically instead of cluttering the controller class with property definitions for every single component. In this case it is convenient to bind the enclosing screen instead:

```
<screen name="myScreen" linkage="s_1">
    <binding target="formController" property="screen"/>
    <component .../>
    <component .../>
    <component .../>
    <button name="submit" target="formController" method="submit"/>
</screen>
```

The example above shows a snippet of a configuration for a screen with a large form. None of the individual form components will be bound to the controller class. Instead the screen instance containing the form will be bound. In this case all you need to define in your controller class is a property for the screen instance and a method named `submit`:

```
class com.domain.package.MyFormController {

    private var _screen:Screen;

    public function setScreen (scr:Screen) : Void {
        // This method will be called automatically due to the binding
        // you defined in XML
        _screen = scr;
    }

    public function submit (button:SimpleButton) : Void {
        // Will be called when the submit button was pressed
        // Just obtain references for all components you need to
        // process before submitting the form data:
        var box:CheckBox = CheckBox(_screen.getElement("box"));
        if (box.isSelected()) {
            doSomething();
        }

        var combo:ComboBox = ComboBox(_screen.getElement("combo"));
        var p:Person = Person(combo.getList().getSelectedModel());
        doSomethingWith(p);

        // etc...
    }

}
```

Note that the parameter passed to the `getElement` method of the screen class corresponds to the `name` attribute you specified for the component in XML.

7.23. Developing and integrating your own components

If you need additional functionality not covered by the Parsley components you can develop your own set of components and integrate them nicely with the rest of the UI framework. Implementing a custom component for Parsley usually involves the following steps:

1. Subclassing `Component` or any other component class in the `parsley.comp` package
2. Telling the component which child elements to expect
3. Overwriting the init method to initialize the component and its children
4. Implementing custom component functionality
5. Preparing the component to be integrated into Parsley's state management

Lets examine all these steps in detail:

Subclassing Component or any other component class in the parsley.comp package

This decision depends on the type of component you want to implement. If, for example, all you want to do is animate the opening and closing of a `Menu` component, you would subclass `Menu` in the `parsley.comp.select` package and just overwrite the `openList` and `closeList` methods. If your component is not related to existing functionality in any way, you can simply subclass the top level `Component` class.

Telling the component which child elements to expect

To do this you overwrite the `prepare` method which gets called during the `onLoad` event. Keep in mind that you should never overwrite the `onLoad` method directly because this would very likely break the integration of that component into the UI framework. Unfortunately Action Script 2 does not allow to declare methods as final.

Let's look at how the Parsley `Scrollbar` component tells the framework which children it expects:

```
private function prepare () : Void {
    expectChild("up", SimpleButton, false);
    expectChild("down", SimpleButton, false);
    expectChild("slider", Slider, false);
}
```

The `expectChild` methods has three parameters: The first one is the instance name of the component. Note that the child does not have to be a "first level" child. It can be nested within other movie clips as long as those movie clips are not Parsley components (i.e. do not extend the base Component class directly or indirectly). The children will "find their way" to the enclosing parent component. The second parameter is the expected class. If there is a component with the specified name but it is not of the correct type, an error will be logged and the component will be disabled. The final paramter is a boolean indicating if the child is required or optional. In the example above all children are optional. Although the `Scrollbar` component will throw an error if none of the children was added, because this would obviously not make sense.

Overwriting the init method to initialize the component and its children

Another private method that gets called for every component is the init method. In contrast to the `prepare` method this method gets called after all `onLoad` events for all expected children of that component have fired. So you can be sure that all child elements are fully initialized when this method is invoked. This may sound trivial but in fact it is a very convenient feature because the Flash Player is kind of weird in ordering the `onLoad` events for movie clips. If some of the movie clips are components with Inspectable properties for example, those components will load before other components. So there basically is no ordering of events you could rely on.

Implementing custom component functionality

Well, I cannot help you with that, sorry… ;-)

Preparing the component to be integrated into Parsley's state management

This is the final optional step. You already saw a lot of example XML configurations which contained the stateful attribute for the `navigator`, `screen`, `view` and `component` tags. If you want your component to participate in the Parsley state management you just have to overwrite two methods of the base `Component` class: `saveMemento` and `restoreMemento`. You should never overwrite the `saveState` or `restoreState` methods of the `Component` base class because those methods contain additional logic for managing the states of child components.

The Parsley `Scrollbar` component manages its internal state like this:

```
private function saveMemento (memento:ComponentMemento) : Void {
    super.saveMemento(memento);
    memento.setAttribute("scrollframes", _scrollframes);
}

private function restoreMemento (memento:ComponentMemento) : Void {
    super.restoreMemento(memento);
    _scrollframes = memento.getAttribute("scrollframes");
}
```

This is pretty straightforward. First you should always call the corresponding method of the superclass so it can add its own attributes. Then you simply add arbitrary attributes to the `ComponentMemento` in the `saveMemento` method, the values can be of any data type. In the `restoreMemento` method you read them from the memento instance and initialize the component using those attributes.

In case you wonder how the `Scrollbar` component remembers the scroll position: This will be handled by the nested `Slider` component! If you have nested components you should never attempt to manage their state in the parent component. The framework will do that for you. It will call the state management methods for all nested components recursivly.

8. Tasks – Handling asynchronous operations

8.1. Overview

The Task framework was designed to help you write asynchronous operations. In the Flash environment you often have to develop functionality that cannot finish immediately and in most of these cases you have to set up callbacks to be notified when the operation is complete. This applies to animations as well as to loading images or XML files. Unfortunately there is no consistent way to handle those tasks since every piece of asynchronous operation built into Flash comes with its own API. There is the `XML` class, the `LoadVars` class and since Flash Player 7 the `MovieClipLoader` class. There is the `XMLSocket` class and Flash Remoting. Furthermore you use `onEnterFrame` functions and `setInterval`. The Task framework in Parsley is an umbrella for this kind of functionality and adds a set of standard events to handle asynchronous tasks.

The concrete task implementations in Parsley can be devided into two categories: The first one is loading content. Parsley comes with a `XMLLoader` and a `MovieLoader` class which both extend the base `Task` class. Furthermore the core `ApplicationContextParser` class is also a subclass of `Task` and supports most of its events. The second category is animation. The `parsley.fx` package includes the `TimelineTask` and `TweenTask` classes which also extend the base `Task` class. In addition two those concrete `Task` implementations the framework includes two more classes for handling collections of tasks which should run simultanously or consecutively: `TaskGroup` and `TaskChain`.

These classes will be described in the following sections.

8.2. Loading content with MovieLoader and XMLLoader

Using `MovieLoader` and `XMLLoader` is pretty straightforward:

```
// example code for loading a SWF and a XML file:

// preparing the MovieLoader:
var mLoader:MovieLoader = new MovieLoader();
var container:MovieClip = getContainerClipSomehow();
mLoader.addFile(container, "help.swf");
mLoader.addListener(TaskEvent.COMPLETE, this, onLoadMovie);
mLoader.addListener(TaskEvent.ERROR, this, onMovieError);

// preparing the XMLLoader:
var xLoader:XMLLoader = new XMLLoader("help.xml");
xLoader.addListener(TaskEvent.COMPLETE, this, onLoadXML);
xLoader.addListener(TaskEvent.ERROR, this, onXMLError);

// starting both loaders:
mLoader.start();
xLoader.start();
```

Note that both loader classes use the same set of events. The `TaskEvent` class contains a set of events that all `Task` implementations should support: `START`, `SUSPEND`, `RESUME`, `COMPLETE`, `CANCEL` and `ERROR`.

The `MovieLoader` class supports one more event in the `MovieLoaderEvent` class: `PROGRESS`. You'll need this event if you want to display a progress bar while the file loads.

8.3. Controlling animation with TimelineTask and TweenTask

TimelineTask

The `TimelineTask` class let's you kind of observe a timeline animation. Let's show a simple example:

```
var animatedClip:MovieClip = getClipSomehow();
var tt:TimelineTask = new TimelineTask(animatedClip, false);
tt.addListener(TaskEvent.COMPLETE, this, onAnimationComplete);
tt.start();
```

This piece of code simply tells the `TimelineTask` to start the movie clip you passed to the constructor and throw a `COMPLETE` event when the animation is finished. But how does the `TimelineTask` class determine when the animation is over? The `COMPLETE` event is fired if the last frame of that clip was reached or if the `stop` method of that clip was called (The framework can detect those calls with method interception, the AOP framework is explained in section `9.2.`). The nice thing is that this even works for loaded movie clips which you have to integrate into your application and for which you do not have the Flash source files so you cannot peek inside and look where the designer added stop calls to the timeline.

A final option to define when the animation is complete is to call the `setLastFrame` method:

```
tt.setLastFrame(20);
```

This would tell the `TimelineTask` instance to play until that frame is reached and then stop the movie clip and fire the `COMPLETE` event.

TweenTask

The `TweenTask` can be used to integrate animations with easeIn and easeOut functions. In the following example code the `_x` property of a movie clip will be modified to move the clip from position 0 to position 120 during the next 50 frames:

```
var func:Function = getTweenFunctionSomehow();
var clip:MovieClip = getTheClipWhichWillBeAnimated();
var tt:TweenTask = new TweenTask(func, clip, "_x", 0, 120, 50);
tt.addListener(TaskEvent.COMPLETE, this, onTweenComplete);
tt.start();
```

Note that Parsley does not include the actual tween functions. I didn't feel the urge to reinvent the wheel here, so I'll just point you to the resources, where you can find existing sets of tween functions:

1. Flash MX 2004 comes with a set of transitions which include a bunch of easing functions. You can find them in the package `mx.transitions.easing`.
2. Robert Penner also wrote a set of easing function. You can download them at www.robert-penner.com/easing/.

If you want to use one of the Macromedia easing functions for example, you could create a `TweenTask` instance like this:

```
import mx.transitions.easing.Elastic;

[...]

var tt:TweenTask
               = new TweenTask(Elastic.easeIn, clip, "_x", 0, 120, 50);
```

8.4. Developing your own tasks

There are basically two options for writing your own tasks: Subclassing the base `Task` class or using the `DelegateTask` class. For larger classes the first approach is recommended, for simple tasks with a few lines of code the second approach is sufficient. Let's show them both.

Extending the base Task class

If you extend the `Task` class in most cases you will only have to overwrite the private run method which is empty in the base class. Never overwrite the start class, your implementation would not work correctly this way. Unfortunately ActionScript 2 does not allow to declare methods as final. The private run method will be called when the start method for that `Task` is called. You should start whatever asynchronous operation you want to implement in the run method and then call one of the methods in the superclass when the operation either completed successfully or when an error occurred.

Let's show a simple example task implementation that just plays a sound:

```
import org.spicefactory.parsley.task.Task;

class com.domain.package.SoundTask extends Task {

    private var _linkageId;

    public function SoundTask (linkageId:String) : {
        _linkageId = linkageId;
        // our task implementation is neither cancellable nor suspendable
        setCancellable(false);
        setSuspendable(false);
    }
```

```
    private function run () : Void {
        var s:Sound = new Sound();
        s.attachSound(_linkageId);
        var scope:SoundTask = this;
        s.onSoundComplete = function () {
            scope.complete();
        };
        s.start();
    }
}
```

This task simply attaches and starts the sound with the `linkageId` passed to the constructor. When the sound finished playing the complete method of the base `Task` class is called. This `Task` implementation will fire two of the events the task framework supports: `START` and `COMPLETE`. If you want to stop the task execution because of an error you can call the error method of the base `Task` class and pass two string arguments as `errorCode` and `errorDetail`:

```
 error("sound.error", "could not load the sound with filename " + filena-
me");
```

Note that we set the example task to be neither cancellable nor suspendable to keep the example simple. Of course you can cancel and suspend playing sounds. If you wanted to integrate this functionality into your task implementation you would have to overwrite the cancel, suspend and resume methods of the base `Task` class. Let's show an example implementation for cancellation for our `SoundTask` class:

```
    public function cancel () : Boolean {
        // always "ask" the superclass if we are in a legal state
        // to be cancelled:
        if (!super.cancel()) return false;

        _sound.stop();
        // assuming _sound is a private property holding the sound instance

        return true;
    }
```

Using DelegateTask

The `DelegateTask` class simply delegates from the run method to the method you passed to the constructor. This way you could integrate simple Tasks into other classes without the need to create a separate `Task` subclass:

```
    public function startSound (linkageId) : Void {
        _linkageId = linkageId; // keep reference in a private property
        var delegate:DelegateTask = new DelegateTask(this, onRun);
        delegate.start();
    }
```

```
private function onRun (delegate:DelegateTask) : Void {
    var s:Sound = new Sound();
    s.attachSound(_linkageId);
    s.onSoundComplete = function () {
        delegate.complete();
    };
    s.start();
}
```

If you still think that Tasks are a rather esoteric feature, have a look at the following section where we start to do something really useful with Tasks.

8.5. Using TaskChain and TaskGroup

So far you only saw that the task framework helps you to write asynchronous operations in a consistent way. Now in this final section about tasks I'll show you how you could take advantage of that consistency to automate more complex opertations. There are two task implementations in the Task package which let you group a set of tasks to be executed simultanously or consecutively: `TaskGroup` and `TaskChain`. If you take the two tasks to load a SWF file and a XML file from the example in section `8.2.` you could use the `TaskChain` class to execute them consecutively:

```
var mLoader:MovieLoader = new MovieLoader();
var xLoader:XMLLoader = new XMLLoader("help.xml");

// ... prepare the two loader instances (add listeners etc.)
// see section 8.2.

var chain:TaskChain = new TaskChain();
chain.addTask(mLoader);
chain.addTask(xLoader);
chain.addListener(TaskEvent.COMPLETE, this, onComplete);
chain.addListener(TaskEvent.ERROR, this, onError);
chain.start();
```

Since the `TaskChain` and `TaskGroup` implementations are subclasses of the `Task` base class themselves they support the same set of events. The `COMPLETE` event is fired when the last task added to the chain throwed its `COMPLETE` event. The `ERROR` event is fired when any of the added tasks fired an `ERROR` event. If the first task (the `MovieLoader`) fires an `ERROR` event for example, the chain instance will also fire that event and the second task will never be started. The nice thing is that you do not have to start the individual tasks, the chain takes care of that. Especially when initializing an application and loading assets and XML configuration you often end up with a bunch of callbacks and the need to keep track of which next operation to start when a particular callback is invoked. It's much easier to use a `TaskChain` for that kind of functionality.

The `TaskGroup` is for executing tasks simultanously. It may be useful if you have complex animations with many movie clips and sounds involved and you want that group of operations to be cancellable or suspendable as a whole. When you invoke the `suspend` method in a `TaskGroup` instance for example, all the tasks added to that group will be suspended.

A last thing to note about TaskChain and TaskGroup is that since both are tasks themselves you can even nest chains and groups within other chains and groups.

9. Utilities

9.1. The Timer class

If you control animations or other asynchronous operations you often end up using `onEnterFrame` events and `setInterval` a lot. I often found this to be a rather cumbersome and inconsistent way for handling operations that have to execute in intervals. The `Timer` class warps both types of operations. Let's show a simple example:

```
public function startInterval () : Void {
    var t:Timer = new Timer(this, onInterval, ["a string", 100]);
    t.schedule(1000, true);
}

private function onInterval (param1:String, param2:Number) : Void {
    // do something useful here
}
```

The `schedule` method expects two arguments. The first one specifies the number of milliseconds that should pass before the method you passed to the constructor of the `Timer` class is executed. The second argument is a boolean specifying if execution of the method should be repeated. If you pass false the method will only execute once which is very convenient because you do not have to remember fancy interval identifiers just to stop execution after the first invocation.

If you want execution of the method to be coupled to the framerate, you can use the `scheduleAtFrameRate` method instead:

```
var t:Timer = ...;
t.scheduleAtFrameRate(true);
```

Again the boolean parameter specifies if execution of the method should be repeated. Before you use the `scheduleAtFrameRate` method for the first time in your application, you must initialize the `Timer` class with an empty movie clip, calling the static `initialize` method:

```
Timer.initialize(anyMC.createEmptyMovieClip("timerController", 1));
```

If you want to stop execution just invoke the `cancel` method on the timer instance.

9.2. AOP – writing method interceptors

Parsley comes with a very small AOP framework. AOP is short for Aspect Oriented Programming and if you never heard about it the following section may contain too much unfamiliar terminology. If you want to read a very short introduction to AOP have a look at the documentation for the Spring framework (www.springframework.org).

Parsley only supports a very small subset of AOP. It only supports method interception (not property interception), only a single advice type (around advice) and does not enable you to define pointcuts.

Using interceptors is a convenient way to observe objects that do not permit to add listeners. Furthermore it can even be used to add control over the execution of methods since you can manipulate the arguments passed to the method and the value returned from that method. You could also prevent the invocation of the target method completely.

Let's show a simple example where we intercept the `setVolume` method of a `Sound` object and make sure that the value passed to that method is never lower than 50.

```
import org.spicefactory.parsley.aop.*;

class com.domain.package.SoundInterceptor implements Interceptor {

    public function invoke (mi:MethodInvocation) {
        // first check the first argument passed to the setVolume method:
        var volume:Number = mi.getArgument(0);

        // if it's lower than 50 overwrite the value:
        if (volume < 50) mi.setArgument(0, 50);

        // now invoke the target method:
        return mi.proceed();
    }

}
```

Finally all you have to do is instantiate your interceptor and tell the framework which methods of which instances should be intercepted:

```
var s:Sound = getSoundInstanceSomehow();
var ic:Interceptor = new SoundInterceptor();
AopUtil.addInterceptor(s, ["setVolume"], ic);
```

Note that the second argument is an array, you could specify more than one method to be intercepted.

Alternatively you can configure interceptors in XML:

```
<interceptor methods="setVolume">
    <object-ref ref="mySoundInterceptor"/>
</interceptor>
```

You can add more than one interceptor for each method. The framework creates a chain of interceptors. In the example above the proceed method of the `MethodInvocation` instance either calls the next interceptor in the chain or the target method if this is the last interceptor in the chain.

9.3. Logging

You already saw how to configure logging in section 3.9. In this section you'll see how to integrate logging into your own classes and how to write a custom Appender.

Integrating Parsley logging into your own classes

This is pretty easy, so let's start with an example:

```
import org.spicefactory.parsley.logging.LogFactory;

import org.spicefactory.parsley.logging.Logger;

class com.domain.package.Example {

    // declare a private static var to hold the logger instance:
    private static var _logger:Logger;

    public function Example () {
        // if this is the first Example instance to be created
        // create the logger for all Example classes:
        if (_logger == undefined) {
            _logger = LogFactory.getLogger("com.domain.package.Example");
        }

    }

    public function doSomething (arg:Number) : Void {
        // now you can use the logger instance:
        if (arg < 0) {
             _logger.error("I don't like this argument: " + arg);
             return;
        }
        _logger.info("This is a really nice argument: " + arg);
        doSomethingUsefulWith(arg);
    }

}
```

Just remember to declare the logger instance as static so that all instances can share the same logger instance. In most cases it is recommended to use the fully qualified class name as an identifier for the logger instance. But you could use any arbitrary string.

Implementing your own Appender

The only appender implentation that Parsley comes with is a simple `TraceAppender` that logs to the Flash Output window. If you want to route log messages to other destinations you can implement the `Appender` interface or subclass the AbstractAppender class. Let's show an example implementation that logs to a `XMLSocket` object:

```
import org.spicefactory.parsley.logging.appenders.AbstractAppender;

import org.spicefactory.parsley.logging.LogLevel;

class com.domain.package.SocketAppender extends AbstractAppender {

    private var _socket:XMLSocket;

    public function SocketAppender (host:String, port:Number) {
        // create the socket instance:
        _socket = new XMLSocket();
        _socket.connect(host, port);
    }

    public function append
            (loggerName:String, level:LogLevel, logMessage:String) : Void {
        // ask the superclass if we are below the configured threshold:
        if (isBelowThreshold(level)) return;

        // now construct the complete log string:
        var log:String = level.toString() + " / " + loggerName
                                                    + ": " + logMessage;

        // finally send the message:
        _socket.send(log);
    }

}
```

To keep the example simple I left out any error checking if the socket connection is successfully established.

Finally let's show how to integrate that appender into your logging configuration:

```
    <constant name="parsley">org.spicefactory.parsley</constant>

    <object id="socketAppender" class="com.domain.package.SocketAppender">
        <constructor-args>
            <string>www.myIncredibleLoggingServer.com</string>
            <number>1024</number>
        </constructor-args>
     </object>

     <log-factory>
         <root level="warn">
             <appender threshold="debug">
               <object class="${parsley}.logging.appenders.TraceAppender"/>
            </appender>
        </root>
        <logger name="${parsley}.comp" level="debug"/>
        <logger name="com.domain" level="debug">
            <appender threshold="warn">
                <object-ref ref="socketAppender"/>
            </appender>
        </logger>
    </log-factory>
```

In the example above the root logger only has a single appender, the builtin `TraceAppender`. The logger with the name "`com.domain`" adds a second appender, your special `SocketAppender` which you configured with a standard object tag specifying the URL and port as constructor arguments. So all classes in the `com.domain` package and subpackages log to the `TraceAppender` and the `SocketAppender`. But since the `SocketAppender` has a threshold of `warn` only warnings and errors will be logged to the socket, lower log levels will only be logged to the `TraceAppender`.

Keep in mind that each logger as well as each appender has its own threshold. For a log message to reach the output destination it must be accepted by both.