

Sign Live! CC Developers Guide

Februar 2022

intarsys GmbH

Sign Live! CC Developers Guide

Version 7.1

Customizing and Developing for Sign Live! CC

intarsys GmbH
Sign Live! CC Developers Guide
Version 7.1

All rights reserved
© 2021 intarsys GmbH
www.intarsys.de

Preface

- Author and company

This book has been provided by different authors from the development staff of intarsys GmbH.

- Trademarks

Wherever possible and where the authors were aware of a trademark claim, such designations are marked as trademarks in this book.

CABAReT is a registered trademark of intarsys (Schweiz) AG.

EForm is a registered trademark of intarsys GmbH.

jPod is a trademark of intarsys GmbH.

Sun, Java and JavaScript are trademarks of Sun Microsystems

Microsoft and Windows are trademarks of Microsoft Corporation.

Adobe and Acrobat are trademarks of Adobe Systems Incorporated

- Who should read this book

This book is intended for Java developers intending to enhance the product or write a new application from scratch. You will gain a deep understanding of the how to's of the Sign Live! CC machinery.

If you simply want to add some buttons to a Sign Live! CC GUI, you may be better off reading the "Scripting Tutorial" and mimicking some of the examples. You don't need the information included here. ...but, still, you may get inspired. There are more possibilities than you might imagine at the moment.

The basic concepts and APIs available to the developer are presented, as well as complete examples that you can use as templates.

You will not need this book if you simply wish to use Sign Live! CC or work with simple scripting features. You will find information thereto in the tutorials and the online help.

■ Organization

The first part introduces all basic concepts of Sign Live! CC and its underlying platform framework “claptz”. After some insights in the big picture of the system we will handcraft a small instrument. The key components that you will need to effectively enhance an application are introduced.

The next part will go into more detail on some special topics, the most important one being the scripting integration.

The next section deals with external APIs. This is how you can use and integrate Sign Live! CC features from another application.

The last part gives some advice on how to build a claptz based system from scratch.

■ Other documentation

The Online Help includes information about every feature that can be done with the desktop application (and some more). It is installed and available with every distribution.

Operator’s Guide is the book about installation and configuration of Sign Live! CC

Now we come to the fun part, do a little programming - The Scripting Tutorial shows a fast path to customizing Sign Live! CC and some useful tips for scripting it.

The Developer’s Guide is the book about programming Sign Live! CC. This is about the architecture, the basic concepts and generic APIs.

On some special topics there is additional documentation, presenting APIs and examples for specific business tasks.

One of the most important is Security Applications Developer’s Guide, the book about external access to the security applications in the Sign Live! CC application.

■ Reviews and comments

We make constant efforts to improve our documentation and meet your requirements. Your comments are welcome and are a valuable resource for us.

Email

support@intarsys.de

Website

www.intarsys.de

■ Disclaimer

Every effort has been made to make this book as complete and accurate as possible, but no warranty is implied.

The information is provided “as is”. The authors shall be in no way liable to any person or entity with respect to any loss or damages arising from the information contained in this book, or from the use of the disks or programs that may accompany it.

Contents

Preface	5
▪ Author and company	5
▪ Trademarks	5
▪ Who should read this book	5
▪ Organization	6
▪ Other documentation	6
▪ Reviews and comments	7
▪ Disclaimer	7
Contents	9
Introduction	13
1. On the Stage	15
1.1 The big picture	15
1.1.1 Overview	15
1.1.2 Schematic Model	16
1.1.3 Life Cycle	17
1.1.4 Basic services and components	17
1.1.5 Summary	18
1.2 A simple instrument	18
1.2.1 Overview	18
1.2.2 Adding an action	18
1.2.3 The instrument declaration	18
1.2.4 Instrument example	19
1.3 Instrument structure	20
1.3.1 Overview	20
1.3.2 Packaging	20
1.3.3 instrument.xml	22
1.3.4 Extension Points	24
1.4 Functor, CodeExit & Action	24
1.4.1 Overview	24
1.4.2 Functor	25
1.4.3 CodeExit	26

Contents

1.4.4	Action	32
1.4.5	Action registry	32
1.5	Widget	33
1.5.1	Overview	33
1.5.2	Declaration	34
1.5.3	Naming convention	37
1.5.4	Namespaces	38
1.6	Lifecycle events	39
1.6.1	Overview	39
1.6.2	Instrument startup	39
1.6.3	Example	40
1.7	Preferences	40
1.7.1	Overview	40
1.7.2	The preferences framework	41
1.7.3	Stage preference usage	42
1.8	Working with processors	45
1.8.1	Overview	45
1.8.2	Processor declaration	45
1.8.3	Processor arguments	48
1.8.4	Processor preferences	50
1.8.5	Processors & Code Exit integration	51
1.8.6	Processors & Commandline integration	52
1.8.7	String expansion	52
1.8.8	The Processor API	52
1.9	Working with wizards	53
1.9.1	Overview	53
1.9.2	Wizard declaration	53
1.9.3	Wizard arguments	56
1.9.4	Wizards & Processor integration	57
1.9.5	Wizards & Code Exit integration	58
1.9.6	Wizards & Commandline integration	58
1.9.7	Standard arguments	58
1.9.8	The Wizard API	59
1.10	Working with documents	60
1.10.1	Overview	60
1.10.2	Document type declaration	60
1.10.3	Processor double dispatch pattern	62
2.	Advanced Use	65
2.1	Toolkit	65
2.1.1	Overview	65
2.1.2	Dispatch execution	65
2.1.3	Dialogs	66
2.2	Script Integration	69
2.2.1	Overview	69
2.2.2	Language Bindings	70
2.2.3	Environment and configuration	70

2.2.4	Script Center	72
2.2.5	CodeExit integration	72
2.2.6	Application globals	73
2.3	Javascript Binding	74
2.3.1	Overview	74
2.3.2	Syntax and semantics	74
2.3.3	JavaScript engine	75
2.3.4	Scripting framework binding	75
2.3.5	Reflection	75
2.3.6	CodeExit Integration	83
2.3.7	Tipps & Tricks	83
2.3.8	LiveConnect features and pitfalls	83
2.3.9	Best practices	87
2.3.10	Common Error	89
2.4	JavaScript Template Binding	90
2.4.1	Overview	91
2.4.2	Motivation	91
2.4.3	Syntax	91
2.4.4	Directives	93
2.4.5	Examples	93
3.	APIs	95
3.1	APIs	95
3.1.1	Overview	95
3.1.2	SDK	96
3.2	Generic API Service Object	96
3.2.1	Overview	96
3.2.2	Mechanics	97
3.2.3	Declaration	97
3.2.4	Modifiers	98
3.2.5	Object selector	99
3.3	Commandline	100
3.3.1	Overview	100
3.3.2	Adding a new option	101
3.4	Shared library	102
3.4.1	Overview	102
3.4.2	Client setup and Installation	102
3.4.3	Demo setup and Installation	103
3.4.4	Design	103
3.4.5	API mechanics	103
3.4.6	API reference	106
3.4.7	Server implementation	108
3.5	ActiveX	108
3.5.1	Overview	108
3.5.2	Client setup and Installation	108
3.5.3	Demo setup and Installation	109
3.5.4	Design	109

Contents

3.5.5	API mechanics	110
3.5.6	API reference	110
3.5.7	Server implementation	115
3.5.8	Problem tracking	116
3.5.9	Integration tips	118
3.6	Service Framework	119
3.6.1	Overview	119
4.	Embedding	120
4.1	Overview	120
4.2	Interfacing Stage	120
4.3	File Environment	120
4.3.1	The file environment	120
4.3.2	The base directory	120
4.3.3	The profile directory	121
4.3.4	The working directory	122
4.3.5	The temp directory	122
4.3.6	The file environment singleton	122
4.4	Launching	122
4.4.1	Launching stage	122
4.4.2	Using an interactive embedding	123
4.5	Embedding a servlet container	126
4.5.1	Running in a servlet container	126
4.5.2	Deployment decisions	126
4.5.3	Reference an existing installation	126
4.5.4	Accessing Stage	128
4.5.5	Additional servlet parameters	128
4.5.6	The configuration file	129
4.6	Controlling	131
4.6.1	Controlling Stage	131

Introduction

First and foremost Sign Live! CC is a powerful standalone application for working with documents, most notably PDF documents. It is a competitive document processing application **out of the box**. If this is what you are looking for, then do not read any further, this document is not for you. You can download and use Sign Live! CC and there is plenty of documentation for it in the online help.

After that, Sign Live! CC is flexible - this offers two additional advantages:

- You can **integrate** Sign Live! CC using one of the many APIs provided
 - Commandline
 - Shared library
 - ActiveX
 - Native Java
 - XML RPC
 - Native HTTP
 - SOAP
- You can **extend** its features using **Instruments**. These extensions range from simply adding an additional icon in the toolbar, giving you a shortcut for a daily task to automatic backend integration, a database, interfacing archiving systems and so on.

The concept of an application built on a flexible platform is today widely known as a “rich client platform”. It is comparable to the concepts and services provided, for example, by Eclipse RCP. The rest of this book will deal with our implementation of this concept, named **claptz** (which is not necessarily a meaningful acronym).

Here is a list of our primary design goals:

Introduction

- **Lightweight**
claptz should be lightweight in all dimensions. Small jar, few dependencies, few resources. All the bells and whistles can be attached, but are not included.
- **Simple**
Insofar as a framework of this type can be called “simple”, this one should fulfill these criteria. The learning curve is not as steep as, for example, in eclipse.
- **Ad Hoc Customization**
The main goal here is to create applications that can be customized upon installation by the user. The customization has to be accomplished with tools typically available at this environment and with easily acquired knowledge. You can watch this feature in action by reading the “Scripting Tutorial”. Almost everything you can do with a Java IDE you can also do with a text editor and a basic understanding of JavaScript. This point can not be stressed enough - the integration of scripting languages into the framework at a very early point is a key concern and other goals may be subordinate to this one.
- **Performant**
The target applications are desktop tools. These tools have to be up and running fast, typically running concurrently with many others. The platform should not eat up CPU and memory resources.

We hope that these goals are reached - at least we are sure that Sign Live! CC **claptz** will remain the platform of our choice.

This book is about integrating, extending and adapting Sign Live! CC. After reading it you should be able to customize your own Sign Live! CC or even build a new application from scratch.

1. On the Stage

1.1 The big picture

1.1.1 Overview

Sign Live! CC is built upon yet another rich client framework, `claptz`. This means there is an infrastructure defining the application's general building blocks and its interfaces, without defining the application itself.

This infrastructure is built on the following concepts

Stage

The coordinating instance of the system. It will bootstrap the system according to the declarations made by the system components. These system components are named **instruments**.

Instrument

An instrument is a component participating in the final application. It can be anything from a set of documents to a complete web server. An instrument may be completely self-contained or it may be dependent on other instruments in the system. Features offered or supported by the instrument are declared using **extension points** and **extensions**. The instrument is said to be the provider of an `IExtensionPoint` or `IExtension`.

Extension

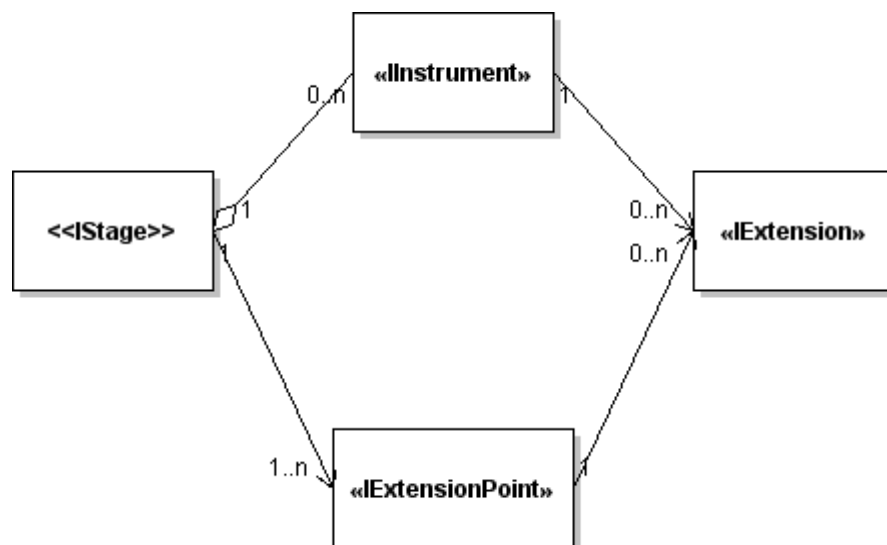
Dependencies between the building blocks (the instruments) are handled using extensions. First, an instrument may allow other instruments to change or extend its behavior. This is done by declaring an **extension point**. Any other instrument can now define an actual **extension** to this point. The semantics of these extension points are completely up to the provider (the instrument). The platform only provides the registry for the declaration.

For example, in a very simple case you build an application launcher for your platform. Your main instrument shows a dialog (or perhaps an icon in the system tray) with a button to launch, let's say, a browser or your favorite text editor. Later on you discover you need a fast path to your mail system. You can change your main class, recompile and redeploy... or else you rewrite your main to provide an extension point. The extension point expects a declaration for a command line to be executed. Your main instrument iterates all extensions and creates a button for each. Next time you need another application in your launcher you just add an instrument declaration providing an extension to this extension point, containing nothing but the command line to be executed. You then relaunch and off you go.

This example shows a few important things:

- Things are often easier when you implement the more general case.
- An extension may be anything; it does not have to be a piece of code.
- Using the platform services not much work is required to take advantage of the additional benefits.
- Your client will thank you for providing such a versatile system.

1.1.2 Schematic Model



This picture shows a simplified model of the platform. The IStage entity holds references to all its loaded IInstrument instances and to all IExtensionPoint instances defined by these IInstruments. The IInstrument itself holds references to all IExtensions it makes to the different IExtensionPoints. The IExtensionPoint knows all IExtensions made by the different IInstrument instances.

1.1.3 Life Cycle

The application life cycle is managed by the Sign Live! CC implementation. The basic course is as follows:

- You start Stage via a standard Java command line or the available launcher.
- This will trigger the launch method of Stage.
- The system will start to bootstrap
 - Read configuration and initialize basic services and environment
 - Install root extension point
 - Find and load instruments
 - Order instruments according to dependencies
 - Start ordered instruments, for each:
 - Register extensions in the order of their declaration
- Run an IMain implementation. This is provided by you via... correct, an extension point
- Stop system
 - Stop ordered instruments in reverse order, for each:
 - Unregister extensions in reverse order of declaration
- cleanup and exit

1.1.4 Basic services and components

Based on `de.intarsys.claptz.*` there is a collection of loosely coupled components and frameworks useful for a broad range of applications:

- Sophisticated CLI support
- Preferences and Properties
- Action and Widget (GUI) abstraction
- Processor and document abstraction
- and some more...

From these you can select and strive for any kind of application. Text editor or web server, it can be done (and has been done).

To make life a little easier, Sign Live! CC has an even more specialized layer. You can use it if it fits your needs.

- Application model
- Application processing model

All in all this will add up to about 500 KB for claptz (including debug).

1.1.5 Summary

By now you should have a fair understanding of the goals of our framework and some of its mechanics. What follows is a lot of information about

- How to use and customize the existing platform. This is part I of this book. Here you will not have direct contact with the implementation classes of the platform above - and you won't miss it.
- How to enhance the platform with your own features and use advanced techniques. This is part II of the book. Here you will access the API for claptz directly. You'll ask yourself, "why didn't I do this earlier?"
- Although Sign Live! CC is fine running standalone, you will wish to include its features in your own application. This is part III.
- And after all, why not use the technology to build an application of your own?

1.2 A simple instrument

1.2.1 Overview

In this section we will create some extensions for Sign Live! CC step-by-step. We will use a lot of features and concepts that are formally introduced in later chapters - don't panic.

1.2.2 Adding an action

The simplest way to extend Sign Live! CC is to add an action. The user presses a toolbar button and the message "hello, world" appears.

While this is quite trivial, there is a lot going on behind the scenes. Other instruments are providing us with the possibility to add actions and widgets. Even more instruments are already extending those basic ones and providing us with toolbar and menubar. And, very easy to overlook, we have some instruments adding powerful scripting features.

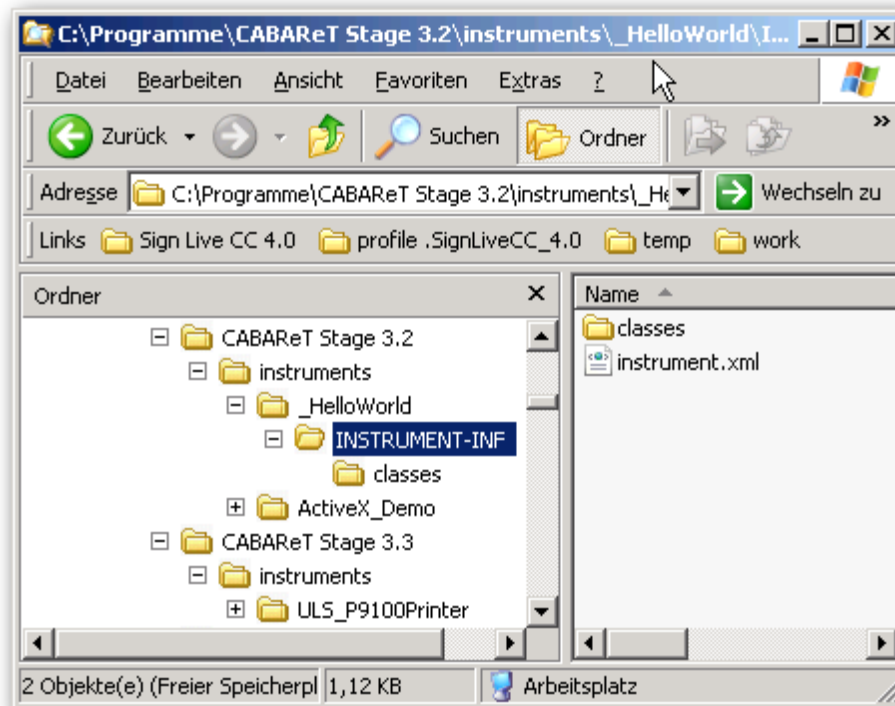
1.2.3 The instrument declaration

To make this instrument available to the platform you have to provide an instrument declaration and include this declaration in the startup process.

An instrument declaration is created as follows:

- A directory with a unique name. This is called the *base* directory of the instrument.
- A directory INSTRUMENT-INF in this base directory. Here you will find the instrument's meta information.

- A file `instrument.xml` in the directory `INSTRUMENT-INF`. This is the XML declaration for the instrument integration.
- In the `INSTRUMENT-INF` directory you can add two other directories that are used by the platform: `classes` and `libs`. The contents of these directories are added to the runtime classpath for this instrument. This allows for a very modular deployment. Everything is contained in the instrument directory.



The instrument declaration itself, contained in the `instrument.xml`, will contain some meta information about the instrument, for example, the name and version, information about the resources used or available like message bundles or (in a later release) information about the signatures for the instrument.

From a functional point of view, it is more important what the instrument contributes to the system. This is declared in its extension and extension point statements.

A more formal definition is available in a later chapter.

1.2.4 Instrument example

The following instrument code (saved in the `instrument.xml` file in the directory structure defined above) will install an action that can be executed via an entry in the Tools menu.

```
<?xml version="1.0" encoding="UTF-8"?>
<instrument
  id="my.company.HelloWorld"
  name="Hello"
  version="1.0">

  <requires>
    <prerequisite
      instrument="com.cabaret.scripting.javascript.application.pdf" />
  </requires>
  <extension point="com.cabaret.claptz.action.actions">
    <action
      id="my.company.HelloWorldAction"
      label="Hello, World">
      <effect>
        <perform type="JavaScript" source="app.alert('Hello, World')"/>
      </effect>
    </action>
  </extension>
  <extension point="com.cabaret.claptz.widget.widgets">
    <widget parent="com.cabaret.widget.menubar.tools/additions">
      <on event="select" action="my.company.HelloWorldAction"/>
    </widget>
  </extension>
</instrument>
```

You can see the instrument meta information in the instrument element, the dependency declaration and two extensions to an existing extension point.

That's all there is to it. If you are eager to test this and experiment, have a look at the “Scripting Tutorial”. There you will be guided further, still using this example, and find some source code to cut and paste.

1.3 Instrument structure

1.3.1 Overview

As you have already seen, an instrument is a functional component, defined in a directory structure. The directory contains resources (data and executable binaries) along with a declaration of how to include this instrument in the system.

1.3.2 Packaging

An instrument consists of different elements

- executables
- resources
- any data you like

and

- the declaration

These parts are organized in a well known way (perhaps you already noticed that we like to borrow the best of all worlds). Everything belonging to an instrument is put in its directory. Give it a meaningful

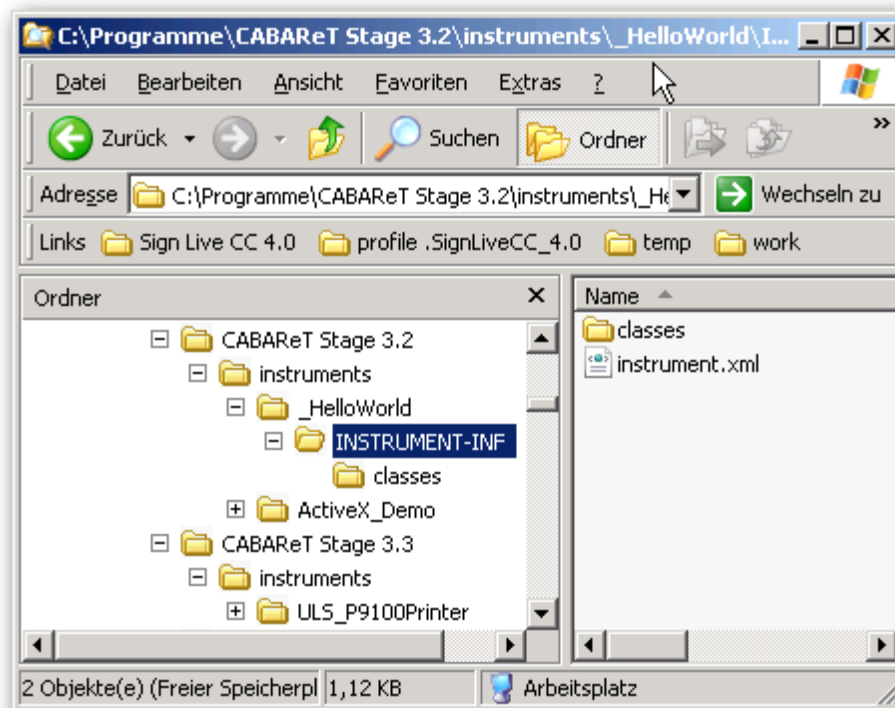
name, but the name will not be used by the platform. Within this directory, include a directory `INSTRUMENT-INF` (and spell it exactly like this, otherwise you will get a surprise when running on another platform). This is where all the meta stuff goes. These are things that should not be available to the “normal” user of your instrument, but only to the platform itself.

The first thing in this directory is the declaration file itself, `instrument.xml`. Again, spell it exactly like this! This one contains the declaration for the platform, what your instrument contains and what it wants to do. The file’s content and semantics is explained in a later chapter.

Every instrument has its own classloader, designed to access its resources (and the ones from the instruments it declares as its prerequisites). This classloader has two built-in access paths: Everything under the directory `classes` and all `.jar` files in the directory `lib`.

If you have executables or resources that should be available to the platform’s (your instruments) classloader, put them in these directories.

The deployable result should look like this



In future releases there may be additional deployment mechanics, for example, packaged instruments.

1.3.3 instrument.xml

1.3.3.1 Example

Here you see an excerpt from the claptz common declaration. All instrument elements are used.

```
<?xml version="1.0" encoding="UTF-8"?>
<instrument
id="com.cabaret.claptz.common"
name="CABAReT Claptz Common"
vendor="CABAReT Solutions AG"
version="3.3">

<requires>
  <prerequisite instrument="com.cabaret.claptz.stage"/>
</requires>
<extension point="com.cabaret.claptz.extension.lifecycle">
  <install
    class="com.cabaret.claptz.common.logging.InstallLogVM"/>
  <install
    class="com.cabaret.claptz.common.logging.InstallLogInfo"/>
</extension>

</instrument>
```

INSTRUMENT

Element	Content		
instrument	Declaration for a plugin feature within claptz		
		Attribute	Content
		id	A unique id for the instrument within the platform. It is a good idea to follow the Java package naming conventions.
	?	bundle	An optional resource file path, interpreted by the class loader. <code>bundle</code> is suffixed with “_<lang>.properties” and the corresponding <code>de.intarsys.tools.message.MessageBundle</code> is associated with the instrument. You can access the resource’s content within the <code>instrument.xml</code> using the string replacement utilities, here <code>\${bundle.<resourcename>}</code> , for any attribute value.
	?	name	An optional name for the instrument.

	?	vendor	An optional vendor name for the instrument.
	?	version	An optional version string for the instrument.
	?	requires	Declaration of other instruments that are used by this one.
	?	extension	Declaration of functional extensions to the platform

REQUIRES

Element	Content		
requires	Declaration of the dependencies of this instrument		
.		Attribute	Content
	-	-	
.		Element	Content
	?	prerequisite	Declaration of a prerequisite.

PREREQUISITE

Element	Content		
prerequisite	Declaration of a prerequisite for this instrument		
		Attribute	Content

	1	instrument	The id of another instrument required for this one
	?	optional	Flag if this prerequisite is optional.

EXTENSION

Element	Content		
extension	Declaration of an extension to an extension point of another provider		
		Attribute	Content
	1	point	The id of the extension point to be extended.
		Element	Content
	1	<any>	extension accepts any child element, subject to the semantics defined by the provider of the extension point. An extension to com.cabaret.claptz.action.actions for example accepts an element action.

1.3.4 Extension Points

Well, there is no point in writing a well formed instrument declaration if you have nothing to extend. This is where it gets interesting. In this manual you will find some aspects of the most important extension points covered.

1.4 Functor, CodeExit & Action

1.4.1 Overview

For this chapter we need to define a concept named “Functor”. A “Functor” is an object representing a behavior, an iconified method. You can compare the representation in the reflection API of Java for a method, which is one possible implementation for a functor.

With `de.intarsys.tools.functor.IFunctor` and the other classes of this package we have another definition for a functor. It is an executable, stateless (!) object that will be triggered using an `de.intarsys.tools.functor.IFunctorCall`. This is the definition used throughout the Sign Live! CC code.

In Sign Live! CC we rely heavily on functors. These objects are very well suited for externally configuring many aspects. They can be used to compute a result that is used somewhere in the application, or, as in this example, they can be used purely for their side effect of doing something - the outcome of an action triggered by a toolbar icon is most often irrelevant. They are fine grained pieces of code, pluggable in any situation that can be reduced to “do this” or “compute that” question, so they are used by many extension points.

Functors are used in three different forms: In the basic form, you can provide an implementation of `de.intarsys.tools.functor.IFunctor` to an extension point.

Next, you can use a “CodeExit”. This is a higher level of abstraction that allows for different kinds of implementation languages and concepts. It is a “generic” functor.

Last in the list is the `de.intarsys.tools.action.IAction`, a kind of decorated functor, suitable for (but not restricted to) use in GUI context.

1.4.2 Functor

1.4.2.1 Implementing a functor

As you already learned, a functor is an object that stands for a single behavior. Here is a simple example, taken from the “HelloWorld” demo.

```
package com.cabaret.demo.app.helloworld;
import java.util.logging.Level;
import java.util.logging.Logger;

import de.intarsys.tools.functor.FunctorInvocationException;
import de.intarsys.tools.functor.IFunctor;
import de.intarsys.tools.functor.IFunctorCall;
import de.intarsys.tools.logging.LogTools;

public class HelloWorldInstaller implements IFunctor {
    private static Logger Log = LogTools.getLogger(HelloWorld.class);
    public Object perform(IFunctorCall call) throws
        FunctorInvocationException {
        Log.log(Level.WARNING, "installing hello world");
        return null;
    }
}
```

The behavior will simply make a log output. No result.

Compile this class, put it in your instrument and use it, for example, as an installer:

```
<extension point="de.intarsys.claptz.lifecycle">
<install class="com.cabaret.demo.app.helloworld.HelloWorldInstaller"/>
</extension>
```

Thats it. Your functor implementations may be more complex, but the basic course is the same.

1.4.2.2 The invocation context

Upon invocation, your functor implementation is called with a `de.intarsys.tools.functor.IFunctorCall` object.

The invocation context is encapsulated to support more complex invocation scenarios, where even more information may be attached to a call, for example, transactional information or a call stack. This is not currently used in the Sign Live! CC environment.

This object carries by default two pieces of information:

- The receiver

As you know, the functor is a behavior. In an object oriented world such behavior is most often attached to an object context, the receiver. Here you get the (optional) receiver object for which the call is performed. The receiver itself depends on the context of the functor definition. A functor attached to a Sign Live! CC widget, for example, will always have the widget instance as its receiver, a functor attached to the object model extension will get the extended instance.

- The arguments

The arguments supplied by the client code.

1.4.3 CodeExit

1.4.3.1 Overview

The **CodeExit** framework provides a concrete implementation of a functor that supports different implementation languages for the functor behavior. You can express your behavior in plain Java (falling back to the `IFunctor`), a scripting language expression, a commandline call or any language and protocol you like. A **CodeExit** is a “meta” functor. It consists of the “source code” (for example the `IFunctor` class name or a piece of JavaScript code) and the “type”, a tag that designates the kind of handler to use for performing the source.

Examples:

```

<perform
  type="Functor"
  source="com.foo.MyFunctor"/>

<perform
  type="JavaScript"
  source="app.alert('hello');"/>

```

These are all **CodeExit** instances, ready to be plugged into Sign Live! CC. Wherever Sign Live! CC needs a functor to plugin or customize behavior, a **CodeExit** definition can be used.

You can define new handlers of your own using the extension point *com.cabaret.claptz.codeexit.codeexithandlers* and provide a class implementing *de.intarsys.tools.codeexit.ICodeExitHandler*.

PERFORM

Element	Content		
perform	Declaration of a CodeExit		
		Attribute	Content
	1	type	A CodeExit type, matching one of the registered handlers.
	1	source	The CodeExit source. This is a generic string whose content depends on the CodeExit type.
.		Element	Content
	?	declarations	Declarations to be applied to the IFunctorCall
	?	handler	Special configuration information for private use by the CodeExit handler implementation.

DECLARATIONS

Element	Content
---------	---------

declarations			
		Attribute	Content
		Element	Content
	?	arg	

ARG

Element	Content		
arg			
		Attribute	Content
	?	name	The name of the argument
	?	value	The default value for the argument if not specified by the client (== null).
	?	modifiers	Optional modifier tags. Modifiers are separated by “;”.
		Element	Content
	?	arg	

1.4.3.2 CodeExit types

Here is an overview of the predefined CodeExit handlers.

1.4.3.2.1 Functor

The “source” attribute is the class name of a `de.intarsys.tools.functor.IFunctor`. Performing the CodeExit will perform the functor.

Example:

```
<perform
  type="Functor"
  source="com.foo.MyFunctor"/>
```

Perform “com.foo.MyFunctor”, which must implement “de.intarsys.tools.functor.IFunctor”. The class must be visible for the CodeExit’s class loader. The class loader for the CodeExit depends on the context of its declaration. Within an instrument, the classloader for this instrument will be used.

1.4.3.2.2 Static

The source is the class name followed by the name of a public static method. The CodeExit will select and execute this method. The class must be visible to the CodeExit’s class loader. The class loader for the CodeExit depends on the context of its declaration. Within an instrument, the classloader for this instrument will be used.

The following implementation signatures are currently supported:

- com.Foo.bar() Static method without arguments
- com.Foo.bar(IFunctorCall call) Static with the functor call object as an argument
- com.Foo.bar(IArgs args) Static method with the argument list as an argument.

Be sure the syntax is correct for “source”, which is the class name, followed by a “.”, followed by the method name, followed by “()”, regardless of the concrete method signature.

Example:

```
<perform
  type="Static"
  source="com.foo.MyFunctor.myMethod()" />
```

Perform the static method “myMethod” in class “com.foo.MyFunctor”.

1.4.3.2.3 Processor

The source is the id of a com.cabaret.claptz.common.processor.IProcessorFactory. Performing the CodeExit will create a processor, start it and return the result.

1.4.3.2.4 ProcessorFactory

The source is the id of a com.cabaret.claptz.common.processor.IProcessorFactory. Performing

the `CodeExit` will create a processor and return it (without starting it). This is most often used in a double dispatch scenario, where a processor factory forwards a request for creation to one of its arguments.

1.4.3.2.5 Wizard

The source is the id of a `com.cabaret.claptz.common.wizard.IWizardFactory`. Performing the `CodeExit` will create a wizard, start it and return the result. Depending on the availability of an associated processor, that processor will be performed automatically.

1.4.3.2.6 WizardFactory

The source is the id of a `com.cabaret.claptz.common.wizard.IWizardFactory`. Performing the `CodeExit` will create a wizard and return it (without starting it). As with `ProcessorFactory`, this can be used in double dispatch scenarios.

1.4.3.2.7 Script

The source code is the logical name of a script. Performing the `CodeExit` will hand the name to the scripting framework. A script with this name will be looked up in the scripting search paths and executed. Special usage contexts of `CodeExit` may provide additional default search paths (as, for example, in the Ultra Light Server implementation, where the web application base directory is added to the search path).

The script lookup supports hot code replacement.

1.4.3.2.8 ScriptFile

The source code is the physical name of a script, including the file suffix. Performing the `CodeExit` will hand the request to the scripting framework to execute the script file. If the filename is relative, the current VM working directory is the parent. Special usage contexts of `CodeExit` may provide other default parents (as, for example, in the Ultra Light Server implementation, where the web application base directory is used as the script file root).

- Script files are not searched in the search paths!
- This supports hot code replacement.

1.4.3.2.9 JavaScript

The source code is a literal JavaScript. Performing the `CodeExit` will request the JavaScript interpreter to perform the source.

1.4.3.2.10 JavaScriptTemplate

The source code is a literal JavaScript template. Performing the **CodeExit** will evaluate the template.

1.4.3.2.11 Action

The source is the id of a registered action. Performing the **CodeExit** will trigger the action.

1.4.3.2.12 Program

The source is a valid shell command. Performing the **CodeExit** will launch an operating system process executing the command.

The source is expanded and may contain string variables (see “String Expansion” in the “Operators Guide”). The following namespaces are currently supported besides the platform standard.

- **args** The **CodeExit** arguments

1.4.3.3 CodeExit declarations

A **CodeExit** supports argument declarations to modify the call issued by the client. With an argument declaration you can adapt the ordering scheme presented by the client to that of the receiver.

If, for example, the client code supplies an indexed argument list, but your code needs named arguments “a” and “b” this declaration will do the job:

```
<perform ...  
  <declarations>  
    <arg name="a"/>  
    <arg name="b"/>  
  </declarations>  
</perform>
```

It is also useful to provide default values, if not already supplied by the client.

```
<perform ...  
  <declarations>  
    <arg name="a" value="foo"/>  
    <arg name="b" value="bar"/>  
  </declarations>  
</perform>
```

The default values are supplied as plain strings. No conversion, no evaluation!

1.4.3.4 Summary

For demonstration purposes we will show you the declarations from the chapter about functors in **CodeExit** notation

```
<extension point="de.intarsys.claptz.lifecycle">
  <install>
    <perform
type="Functor"
source="com.cabaret.demo.app.helloworld.HelloWorldInstaller"/>
  </install>
</extension>
```

Used often in Sign Live! CC: The JavaScript flavor.

```
<extension point="de.intarsys.claptz.lifecycle">
  <install>
    <perform
      type="JavaScript"
source="java.util.logging.Logger.getLogger('foo').warning('bar')"/>
  </install>
</extension>
```

To give you an idea of other handlers you could write, how about:

- HTTP
- Your own process / batch / work unit abstraction
- ...

1.4.4 Action

In the Sign Live! CC terminology, an action is a “decorated” functor. Besides the “perform” method, implementing the functors behavior, there are three other important features:

- “isEnabled” can check if the action is available
- “isChecked” returns the state for a two state action
- it implements *de.intarsys.tools.presentation.IPresentationSupport*. making it immediately available for GUI use

You shouldn’t be surprised that actions are used to define the behavior of widgets, for example, menu or toolbar items.

1.4.5 Action registry

For action objects there is a single registry within Sign Live! CC to work with named actions. Actions in the registry can be referenced and reused by other objects, for example, widgets.


```
<extension point="com.cabaret.claptz.action.actions">
  <action
    id="com.cabaret.demo.app.helloworld.action.Foo">
    <effect>
      <perform
        type="Functor"
        source="com.cabaret.demo.app.helloworld.FooFunctor" />
      </effect>
    </action>
  </extension>
```

Again, this is an example from the “HelloWorld” demo. Here we have an action named `com.cabaret.demo.app.helloworld.action.Foo`, whose effect is implemented using the `IFunctor` `com.cabaret.demo.app.helloworld.FooFunctor`.

In much the same way you could have used a JavaScript `CodeExit`

```
<extension point="com.cabaret.claptz.action.actions">
  <action
    id="com.cabaret.demo.app.helloworld.action.Foo">
    <effect>
      <perform
        type="JavaScript"
        source="app.alert('foo')" />
      </effect>
    </action>
  </extension>
```

1.5 Widget

1.5.1 Overview

Widgets are in between an abstraction layer from the platform window toolkit and a declaration utility for customizing the GUI. Most of the time you will not touch their programming, as the widget hierarchy is declared in the `instrument.xml` or customized using interactive tools like preferences.

Widgets are used wherever it is desirable for the user to get involved in the GUI, for example:

- Menubar
- Toolbar
- Statusbar
- System tray menu

1.5.2 Declaration

1.5.2.1 Extension Point

Widgets are declared in the instrument.xml using the extension point *com.cabaret.claptz.widget.widgets*.

If you reference other widgets via the **parent** attribute, be sure to declare a prerequisite in your instrument to the instrument that contains the parent declaration. This will ensure that the parent widget is loaded 'before your widget can reference it. Otherwise, chances are good (and not predictable) your declaration will be ignored (there will be a warning in the log that the parent is missing)...

WIDGET

Element	Content		
widget	The widget declaration		
.		Attribute	Content
	?	id	The id for this widget. If not available, a unique id is created automatically. This name defines the local name of the widget within its parent.
	?	register	An optional id for the widget to allow global lookup in the registry.
	?	parent	If the widget hierarchy is not defined using nested widget declarations (see element widget), you can declare the parent to be used via a global lookup id (see attribute register).
	?	type	The widget type to be instantiated in the platform toolkit. Most of the time this is derived from the context (like toolitem for entries below the com.cabaret.widget.toolbar).

	?	style	Style flags for the widget. The available flags depend on the widget type used.
	?	label	A label for presentation purposes
	?	tip	A short string with information about the widget - usable in a tooltip.
	?	description	A long string with information about the widget, usable in a context help.
	?	icon	A string referencing an icon reachable via the widget's classpath.
		Element	Content
	*	widget	The child widgets
	*	on	Event handling declarations
	*	property	Generic widget properties

ON

Element	Content		
on	Widget action mapping. You can map widget events to your business logic. The widget is later acting as a bridge between a platform control and the business logic. When the control triggers an event, the widget is informed and in turn triggers the the behavior (a functor) declared using on .		
		Attribute	Content
	?	event	A event name representing some event of the control.

	?	action	The id of an action to be executed when event is triggered. This is a shortcut for declaring a CodeExit with an “Action” type.
	?	class	The implementation class of an <code>de.intarsys.tools.functor.IFunctor</code>
		Element	Content
	?	perform	A CodeExit declaration

PROPERTY

Element	Content		
property	A generic property for the widget		
.		Attribute	Content
	?	name	The property name
	?	value	The property value
		Element	Content
		accessor	Declaration of an <code>de.intarsys.tools.reflect.IFieldHandler</code> instance that bridges the property to your business logic.

ACCESSOR

Element	Content
accessor	The definition of the business logic for accessing a property

		Attribute	Content
	?	class	A class implementing de.intarsys.tools.reflect.IFieldHandler
	?	value	The property value (optional)
		Element	Content
		get	A functor implementing the “get” behavior for this property
		set	A functor implementing the “set” behavior for this property

1.5.3 Naming convention

We highly recommend following a clean naming convention - or you will end in the same mess we found ourselves at the end of product line “3”! Many features and enhancements, each with a different naming scheme...

- Start with a prefix identifying your own namespace - the Java package naming convention is just fine.
- use a tag to identify the role of the id. This seems to be unnecessary, but will ease searching in the instrument declarations with a text editor. For widgets, we use widget.
- Indicate the context of your widget. Top level contexts are for example menubar or toolbar. If you want you can follow this hierarchy further, for example,
com.cabaret.widget.menubar.file.open.

If you provide NLS for your widgets, use the suffix starting at the widget role and append label or tip and so on. Example:
widget.menubar.file.open.label.

For the icon you should use the same convention, resulting in
widget.menubar.file.open.ico. Maybe you can omit the role (widget) and context (menubar) if you reuse the icon or the name is unique.

1.5.4 Namespaces

The widget naming in Sign Live! CC follows a naming scheme to ease extension and customizing.

This list gives the root structure for the widget tree:

- `com.cabaret.widget.root`
 - `com.cabaret.widget.menubar`
 - `./left`
 - `./additions`
 - `./right`
 - `com.cabaret.widget.toolbar`
 - `com.cabaret.widget.statusbar`
 - `com.cabaret.widget.systemtray`

1.5.4.1 Menus

There are some predefined menu widgets for the application. Most of the groups are structured in subgroups using “top”, “additions” and “bottom” as children by default.

You can use this grouping to achieve an intuitive appearance of your widgets on the user interface.

- `com.cabaret.widget.menubar.file`
 - `./top`
 - `./additions`
 - `./bottom`
 - `com.cabaret.widget.menubar.file.open`
 - `com.cabaret.widget.menubar.file.save`
 - `com.cabaret.widget.menubar.file.print`
 - `com.cabaret.widget.menubar.file.exit`
- `com.cabaret.widget.menubar.edit`
 - `./top`
 - `./additions`
 - `./bottom`
- `com.cabaret.widget.menubar.view`
 - `./top`
 - `./additions`
 - `./bottom`
 - `com.cabaret.widget.menubar.view.sidebar`

- `com.cabaret.widget.menubar.tools`
 - `./top`
 - `./additions`
 - `./bottom`
- `com.cabaret.widget.menubar.extras`
 - `./top`
 - `./additions`
 - `./bottom`
- `com.cabaret.widget.menubar.windows`
 - `./top`
 - `./additions`
 - `./bottom`
- `com.cabaret.widget.menubar.help`
 - `./top`
 - `./additions`
 - `./bottom`

1.5.4.2 Toolbars

There are no predefined children for the toolbar yet.

1.6 Lifecycle events

1.6.1 Overview

It is common to perform some action upon platform start or termination. There is a simple way to support this in claptz.

With the extension point `de.intarsys.claptz.lifecycle` you can add behavior to the lifecycle of your instrument.

1.6.2 Instrument startup

As you already know, the bootstrap process involves ordering the instruments according to their prerequisites and then starting them one-by-one. The instrument itself will then register its extensions in the order of their declaration. What is done upon registration depends on the extension point semantics, defined by the provider.

The extension point `de.intarsys.claptz.lifecycle` will execute all functors defined in the element **install** upon registering and all functors in the element *uninstall* upon deregistering of the extension.

Binding an extension's lifecycle is a bit more flexible than adding to the lifecycle of your instrument itself, as you can determine the exact place within the instrument's internal startup by the sequence of your

extension declarations. You can even add different behaviors at different places during startup.

1.6.3 Example

```
<extension point="de.intarsys.claptz.lifecycle">
  <install
    class="com.cabaret.demo.app.helloworld.action.InstallHelloWorld"/>
  <uninstall>
    <perform
      type="Functor"
      source=
        "com.cabaret.demo.app.helloworld.action.UninstallHelloWorld"/>
    </uninstall>
  </extension>
```

This example is taken from the **DemoHelloWorld** instrument and will execute the functor *com.cabaret.demo.app.helloworld.action.InstallHelloWorld* upon instrument startup and *com.cabaret.demo.app.helloworld.action.UninstallHelloWorld* upon shutdown.

As you can see, you have the option of a shortcut using a functor implementation directly or by using the ubiquitous concept of a `CodeExit`.

1.7 Preferences

1.7.1 Overview

Preferences are used to allow for smooth customization of the product to the user's needs. This can be done using explicit input (like in a preferences page) or implicit knowledge (remember the last location where the user stored a file)

The quality of your product depends heavily on the support the user experiences when handling the GUI. You certainly know how annoying it is to select the same directory over and over again...

As we will show here it is very simple, so why not offer it to your users?

In Sign Live! CC preferences are supported at different levels.

- The component framework of `de.intarsys.tools.preferences.IPreferences` offers direct access to preferences structure and persistence. If you wanted to, you could implement your preferences strategy here. Store it in a database or select it from the internet to show the same GUI in every internet cafe... (by the way: if you implement this, make it an instrument and make it accesible to the public).

- The standard preferences strategy used with Sign Live! CC. Sign Live! CC uses a 3 level preferences store (DEFAULT, SYSTEM and USER), persisting the user preferences in a personal directory for the user. The preferences namespace itself is “segmented” to support modularity and improve performance. The first segment in a preferences name designates a physical file, all other segments navigate to the persistent nodes within this file.

Additionally, a framework for preferences pages is provided to manage these preferences using a GUI.

- The preferences protocol is included in standard Sign Live! CC components, like the processor factories. Implementing a processor factory is often linked with providing user adjustable preferences. So it is easy for you to implement it.

1.7.2 The preferences framework

1.7.2.1 API

The framework around `de.intarsys.tools.preferences.IPreferences` is quite similar to the standard `java.util.preferences` and it is just a further abstraction to be free to determine the concrete implementation and to implement some nifty features (like scopes).

The usage pattern is much the same as in other libraries. There is an `de.intarsys.tools.preferences.IPreferencesFactory` and a singleton access at `de.intarsys.tools.preferences.PreferencesFactory.get()`.

Accessing concrete preferences is as simple as

```
static protected IPreferences createPreferences() {
    String name = "com.cabaret.demo.app.helloworld";
    IPreferences root = PreferencesFactory.get().getRoot();
    IPreferences result = root.node(name);
    return result;
}
```

1.7.2.2 XML serialization

Sign Live! CC uses the XML serialization by default and we will show the format here shortly.

A preferences node is represented by the “node” element, which itself can contain other “node” or “property” elements.

NODE

Element	Content
---------	---------

node	A preferences node		
		Attribute	Content
	1	name	The node name within its parent
		Element	Content
	?	node	Declaration of child nodes
	?	property	Declaration node properties

PROPERTY

Element	Content		
property	A property node		
		Attribute	Content
	1	name	The property name
	1	value	The property value

1.7.3 Sign Live! CC preference usage

1.7.3.1 Standard preferences

Sign Live! CC installs its standard preferences with the ClaptzCommon instrument.

This implementation uses these basic concepts:

1.7.3.2 Scopes

A single preference property exists in multiple user definable layers. It is initialized by default with the scopes “DEFAULT” for transient initialization values, “GLOBAL” for installation of system wide preferences and “USER” for user specific property bindings. Each of these layers can hold a value for the property on its own. The higher

layer hides the value of the lower layer. This way you can define program defaults, the administrator can customize them in the installation and a user can again change this settings to satisfy their own needs. A read operation is performed on all (from “USER” to “DEFAULT”) layers. A write operation is performed on the “USER” layer. Each (persistent) scope is mapped to a physical directory - “DEFAULT” is transient, “GLOBAL” is found in the “preferences” subdirectory of the applications base directory (the installation directory), “USER” is found in the “preferences” subdirectory of the users profile directory.

1.7.3.3 Root preferences

The first segment is always translated to a file holding all descendants of this node. This way the preferences namespace you use is spread over multiple independent files, allowing for better modularity and performance.

In the following example the first two preferences reference two properties in the file “com.cabaret.prefs.foo.prefs”. The third path references a property in another file: “com.cabaret.prefs.bite”.

```
/com.cabaret.prefs.foo/bar/x  
/com.cabaret.prefs.foo/boo/y  
/com.cabaret.prefs.bite/bark
```

1.7.3.4 Preferences serialization

By default, preferences serialization is done using the XML mapping described above.

1.7.3.5 Preferences initialization

In some cases you may want to define defaults for the preferences to be used. The extension point com.cabaret.claptz.preferences.declarations does allow this.

```
<extension point="com.cabaret.claptz.preferences.declarations">  
  <node name="/com.cabaret.demo.app.helloworld">  
    <property name="greeting" value="high!" />  
  </node>  
</extension>
```

This example initializes the property “greeting” in the node “com.cabaret.demo.app.helloworld” to “high!”. The notation follows the standard preferences serialization format.

1.7.3.6 Preferences pages

To support direct editing of preferences you can add a preferences page. For the “HelloWorld” demo it looks like this

```
<extension point="com.cabaret.claptz.preferences.pagefactories">
  <pagefactory
    class=
      "com.cabaret.claptz.common.preferences.swt.GenericPreferencesPageFactory"
    pageclass=
      "com.cabaret.demo.app.helloworld.HelloWorldPreferences"
    parent="com.cabaret.demo.app.DemoApplications"/>
</extension>
```

Your implementation can provide `com.cabaret.claptz.common.preferences.IPreferencesPageFactory` or you can use the generic factory `com.cabaret.claptz.common.preferences.swt.GenericPreferencesPageFactory`, submitting only the name of the pageclass implementing `com.cabaret.claptz.common.preferences.IPreferencesPage`.

Anyway your preferences page will be included in the preferences dialog.

You can also use a parent preferences page under which your own page is located in the dialog. If you omit the parent page, your page is in the root of the preferences dialog.

1.7.3.7 Preferences namespace

Like always, there is a naming convention for preferences with Sign Live! CC.

After the well known prefix `'com.cabaret` there is the suffix `"prefs"`, followed by an identifier representing the semantic context for the preferences.

For document type related preferences this identifier will always be `document` followed by a shortform (for example, the well known file extension) for the document type.

```
com.cabaret.prefs.document.pdf
```

will reference the preferences root node for "PDF" document related preferences.

1.7.3.8 Best practices

- **Support the user**
Think from a users perspective. Make sure they never (or almost never) have to do things twice. Remember their choices.
- **Make your implementation configurable**
Never use hardcoded values. It's quite easy to use variables or arguments, so why not do it.
- **Do it the standard way**
If you provide configuration options or preferences, do it the standard way. Not with another properties file, a new dialog or other nonsense. The user needs consistency. If you don't like the

available design then get rid of it. Make a new standard and then do it the standard way.

1.8 Working with processors

1.8.1 Overview

An `IProcessor` is the abstraction of a “work unit”, a process, a stateful functor. Examples of `IProcessor` implementations in Sign Live! CC are “Loader”, “Viewer” or “Page Rotator”.

Some of the uses for this abstraction are

- Standard API for accessing different kinds of behavior
- Sharing of common infrastructures, such as wizard support, preferences,...
- Support the design of reusable behaviors.

The design and implementation of the processor concept is important for understanding and using available processors, as well as developing your own.

The following chapters describe Sign Live! CC standard implementation for `com.cabaret.claptz.common.processor.IProcessorFactory`, `com.cabaret.claptz.common.processor.CommonProcessorFactory`. If you just subclass these, all the features described here are already available. If you decide to implement your own strategy, you can consult the code to get an idea of how to do it.

1.8.2 Processor declaration

PROCESSORFACTORY

Element	Content		
processorfactory	Add a new <code>IProcessorFactory</code>		
		Attribute	Content
	1	class	The class name of the processor factory to register.
	?	id	An unique id for the processor factory within the platform. It is good practice to

			<p>follow the Java package naming conventions.</p> <p>The default value for id is the value of the class attribute.</p>
	?	service	The type of service provided by this processor expressed as a type (class or interface) name.
	?	preferences	The path to the preferences node for this processor factory.
		Element	Content
	?	alias	
	?	declarations	

ALIAS

Element	Content		
alias	Define another entry for this processor factory in the outlet using another lookup id.		
		Attribute	Content
	1	id	Another id where the processor factory can be looked up.

DECLARATIONS

Element	Content		
declarations			

		Attribute	Content
		Element	Content
	?	arg	

ARG

Element	Content		
arg			
		Attribute	Content
	?	name	The name of the argument for the processor
	?	value	The default value for the argument if not specified by the client (== null).
	?	modifier	Optional modifier tags. Modifiers are separated by “;” Supported modifiers: transient Indicates that the argument value is not to be stored in the preferences.
		Element	Content
	?	arg	

1.8.3 Processor arguments

1.8.3.1 Argument chain

A processor factory preprocesses the arguments before it creates the processor instance. In Sign Live! CC access to the processor arguments is implemented in multiple layers:

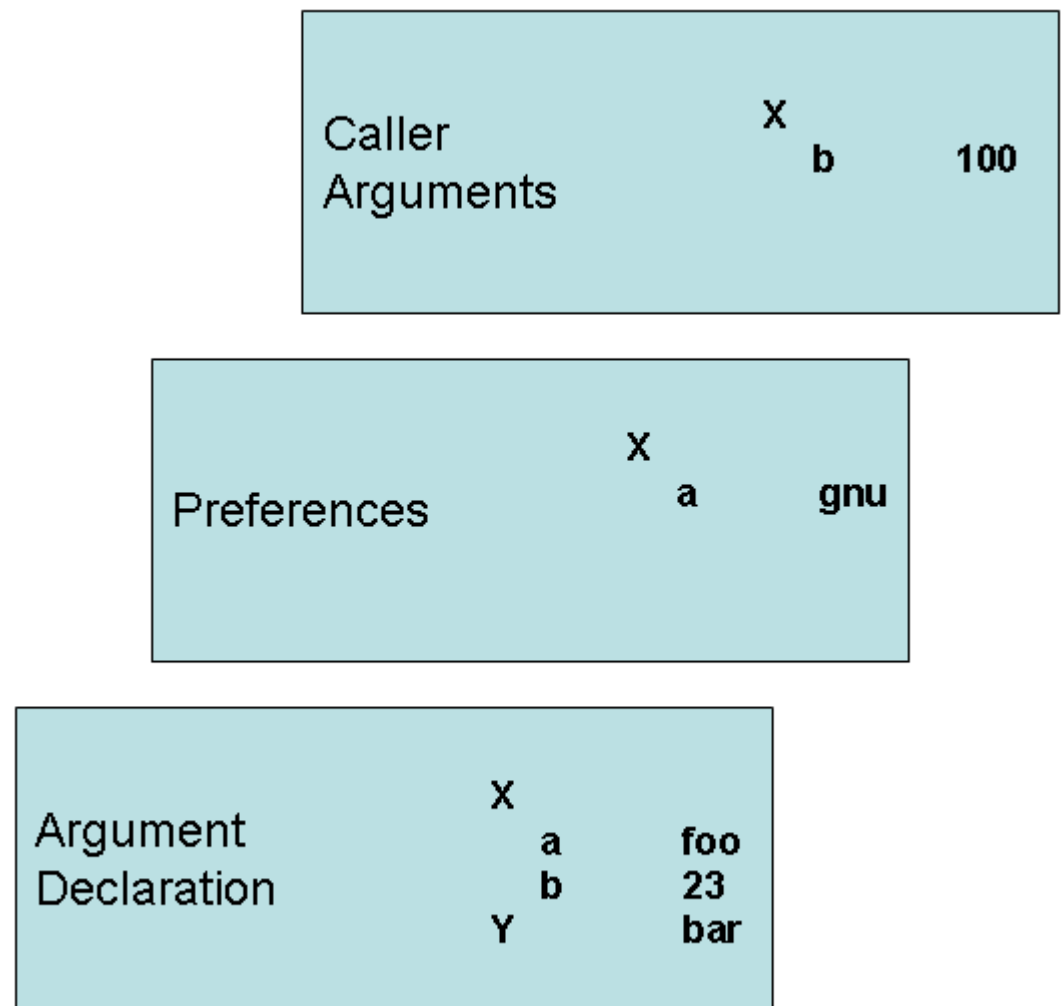
- **Declaration**
For every processor you can declare the argument and its default values within the processor factor declaration. Use the `<declarations>` child element to do this. The default argument values are always made available as a fallback layer in the concrete `IProcessor` argument chain. If a client does not supply a concrete value for an argument, the processor will see the declared default. Remember that this implies that you can not supply “null” as a legal argument value, as this is interpreted as the absence of a value.
- **Preferences**
Some of these values will be changed more often or changed by users not comfortable with updating an “instrument.xml” file. These values should be stored along with the `IProcessor` preferences and published with a preferences page.

The “preferred values” for the arguments are found in the “args” sub node of the processor preferences.

The preferences values are selected using the argument names from this node - thus, be sure to use correct preferences names if you want them to be automatically mapped.

The `IProcessor` will never write back argument values to its preferences.

- **Caller**
Of course, the caller can provide arguments for the processor itself.



This picture shows a typical argument chain. In the processors declaration argument default values are provided for "Y" and the nested values "a" and "b" within "X". The declaration will look like this:

```
<processorfactory ...
...
<declarations>
  <arg name="X">
    <arg name="a" value="foo"/>
    <arg name="b" value="23"/>
  </arg>
  <arg name="Y" value="bar"/>
</declarations>
...
</processorfactory>
```

The serialized preferences will look something like this:

```
<node...  
  ...  
<node name="args">  
  <node name="X">  
    <property name="a" value="foo"/>  
  </node>  
</node>  
  ...  
</node>
```

The call, here in JavaScript:

```
ProcessorReflector.callArgs("MyProcessor", {  
  X: {  
    b: 100  
  }  
});
```

Altogether the processor instance will see the following argument values:

```
X: {  
  a: 'gnu';  
  b: 100  
}  
Y: 'bar'
```

1.8.4 Processor preferences

In this section we will only deal with the preferences part. If you want your processor to handle preferences the Sign Live! CC way, you will require some additional steps.

1.8.4.1 Preferences namespace selection

By default a processor factory uses no preferences. If you want to activate preferences support, you will first have to determine: What is the preferences namespace (the physical file and node within it) where the preferences are stored? This will depend on your design decisions, based on modularity and performance considerations.

For example, there is a preferences namespace for every document type in Sign Live! CC. The standard processor factories like “Viewer”, “Printer” and so on are implemented to store their preferences in a subnode of the document preferences.

Some other components simply have a “per instrument” preferences namespace. Again, subcomponents installed with the instrument store their preferences in subnodes.

The preferences associated with a processor factory are accessed using `getPreferences`.

To activate preferences with your processor factory you can simply redefine one of the methods `createPreferences` or `getPreferencesName`. Even simpler is to configure your choice in the `instrument.xml`.

```
<extension point="com.cabaret.claptz.processor.processorfactories">
  <processorfactory
    class="com.cabaret.document.pdf.PDFViewerFactory"
    preferences="/com.cabaret.document.pdf/viewer">
    ...
  </processorfactory>
  ...
</extension>
```

This is an excerpt from the “PDFDocument” instrument. The factory declares fully qualified access to the preferences node `viewer` in the file associated with `com.cabaret.document.pdf`.

1.8.4.2 Preferences content

The next decision is: What to store?

You have already learned how to “redefine” argument default values using preferences. You do this by defining preference values in the “args” subnode of the processor preferences.

Of course, you can store other information as well. In the Sign Live! CC implementation there is nothing else stored in the preferences in the basic default implementation.

1.8.5 Processors & Code Exit integration

Sometimes it would be nice to start a processor directly via an action or a comparable event source. Its straightforward to provide a `CodeExit` type that accepts the processor factory id as its source. Using

```
<perform
  type="Processor"
  source="com.foo.bar.MyProcessorFactory"/>
```

will create and run processor created by the referenced factory. The result of the `CodeExit` is the result of the processor executed.

If you need to configure even more sophisticated scenarios, you may need the processor itself as the result of a `CodeExit` (for example, when declaring object extensions for a document type where the method semantics asks for the return of the load processor for this document type). So a `CodeExit` type “ProcessorFactory” is provided as well. Its source then references a registered processor factory id. The result is the processor instance created (which is not started).

```
<perform  
  type="ProcessorFactory"  
  source="com.foo.bar.MyProcessorFactory"/>
```

1.8.6 Processors & Commandline integration

While you already know that our fundamental concepts “Commandline”, “perform” option and CodeExit usage are built orthogonal, it may be useful to recapulate: Its quite simple to start a processor from the commandline:

```
SignLiveCC.exe -perform -pt Processor -ps \  
  com.foo.bar.MyProcessorFactory
```

1.8.7 String expansion

The common processor implementation is equipped with the powerful string expansion component. For more information on the syntax and features of this component see Sign Live! CC Operator’s Guide chapter “String expansion”.

Here we will provide a quick overview of how to access the API and what variables are provided.

“getStringEvaluator” provides an `de.intarsys.tools.expression.IStringEvaluator` that can be used to access context information commonly required when configuring a processor. The following namespaces and variables are always provided:

- **processor** or **no prefix** (root namespace)
Variables local to the processor
- **autoid**
A VM wide id for the processor
- **serial**
A VM wide id for the processor
- **args**
Access the arguments of the processor

The string expansion in the processor is always decorated with the processing utility as described in Sign Live! CC Operator’s Guide chapter “String expansion”. This way you can post the process result in many ways, including recursive re-evaluating or string processing using a user defined functor.

1.8.8 The Processor API

1.8.8.1 Overview

The processor API is covered in depth by the respective JavaDoc. Here we will give some general information and entry points.

1.8.8.2 The outlet

The `com.cabaret.claptz.common.processor.IProcessorOutlet` is a registry for `com.cabaret.claptz.common.processor.IProcessorFactory` instances.

According to the naming conventions in Sign Live! CC, there is a singleton `com.cabaret.claptz.common.processor.ProcessorOutlet` that will give you access to the system wide instance.

With this instance Sign Live! CC will register and lookup all processor factories.

1.8.8.3 The factory

The factory `com.cabaret.claptz.common.processor.IProcessorFactory` is responsible for creating and managing the `com.cabaret.claptz.common.processor.IProcessor` instances. Processor factories can be declared using the extension point `com.cabaret.claptz.processor.processorfactories`.

1.8.8.4 The processor

The `com.cabaret.claptz.common.processor.IProcessor` is the workhorse of the application. This is the abstraction for a (quite high level) unit of work.

1.8.8.5 Processor tools

Accessing the processor framework is supported by the tool class `com.cabaret.claptz.common.processor.ProcessorTools`. Here you will find shortcuts for creating and processing processor instances.

1.9 Working with wizards

1.9.1 Overview

A wizard is provided to guide the user when entering complex or lots of data. The data is partitioned into subforms and presented page-by-page at a pace set by the user.

The implementation is very similar to the processor framework and in many respects it is closely intertwined to allow for the smooth combination of wizards and processors.

The following chapter describes the standard Sign Live! CC implementation of a wizard component. If you simply subclass this, all the features described here will already be available.

1.9.2 Wizard declaration

A wizard factory is declared in the extension point `com.cabaret.claptz.wizard.wizardfactories`. The wizard is made available by its factory, an instance of `com.cabaret.claptz.common.wizard.IWizardFactory`.

WIZARDFACTORY

Element	Content		
wizardfactory	Add a new IWizardFactory		
		Attribute	Content
	1	class	The class name of the wizard factory to register.
	?	id	<p>A unique id for the wizard factory within the platform. It is a good idea to follow the Java package naming conventions.</p> <p>The default value for id is the value of the class attribute.</p>
	?	processorfactory	A link to a processor factory. Use this to declare an association to the factory that creates a processor that acts upon the arguments collected by the wizard.
	?	preferences	The path to the preferences node for this wizard factory. It is a good idea to mimic the preferences name of an associated procesor factory and add a suffix (like ".wizard").
		Element	Content
	?	declarations	

ALIAS

Element	Content
---------	---------

alias	Define another entry for this wizard factory in the outlet using another lookup id.		
		Attribute	Content
	1	id	Another id where the wizard factory can be looked up.

DECLARATIONS

Element	Content		
declarations			
		Attribute	Content
		Element	Content
	?	arg	

ARG

Element	Content		
arg			
		Attribute	Content
	?	name	The name of the argument
	?	value	The default value for the argument if not specified by the client (== null).

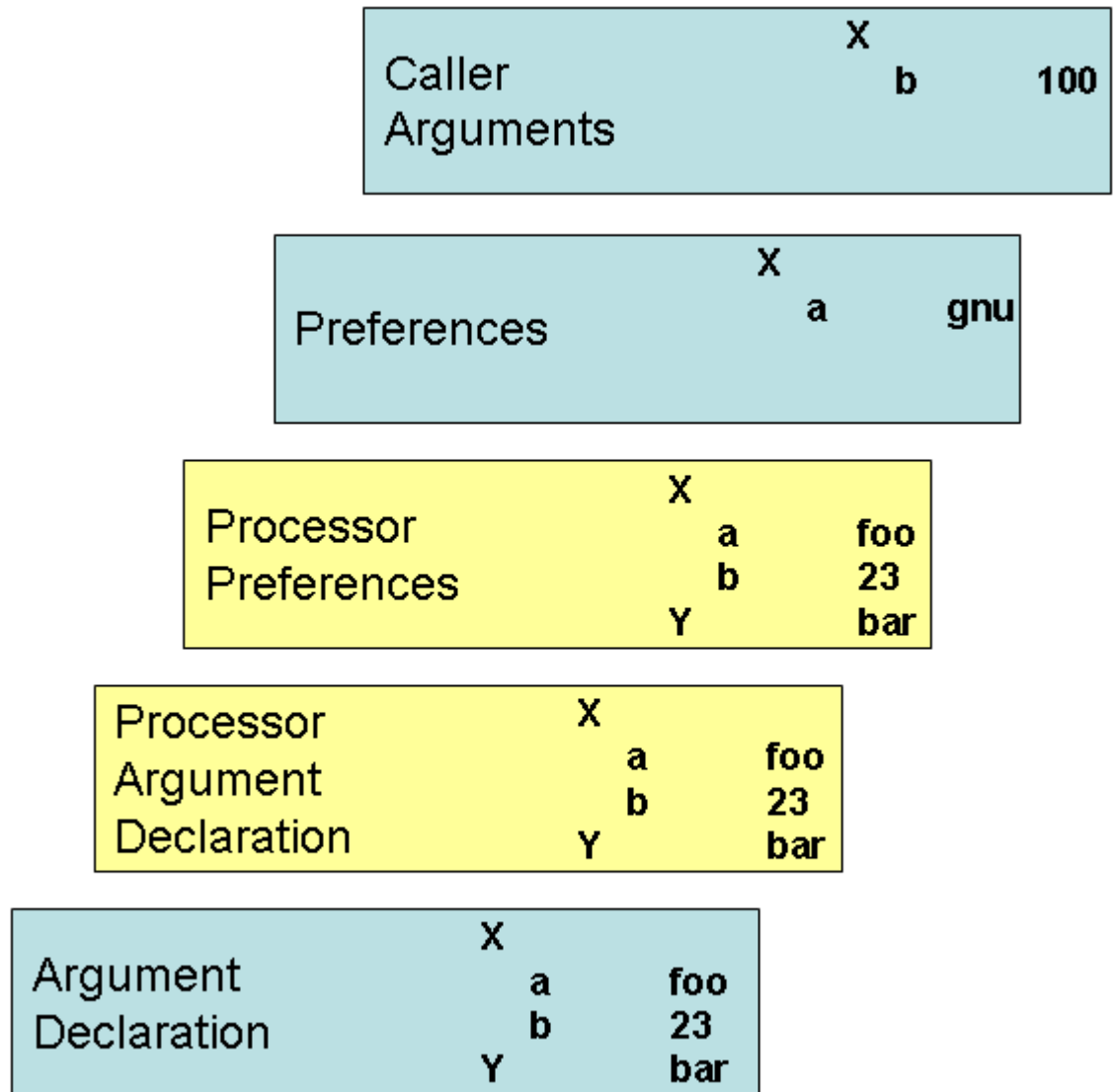
	?	modifier	Optional modifier tags. Modifiers are separated by “;” Supported modifiers: transient Indicates that the argument value is not to be stored in the preferences.
		Element	Content
	?	arg	

1.9.3 Wizard arguments

1.9.3.1 Argument chain

The wizard factory pre-processes the arguments before it creates the wizard instance - same as the processor factory. As a wizard may be used to provide the necessary argument input to a processor, the default layers of the processor argument chain are also included.

- **Declaration**
For each wizard you can declare the argument default values within the wizard declaration. Use the **<declarations>** element to do this. This declaration facility should only be used if no associated processor factory is defined for the `IWizardFactory` to avoid redundancies.
- **Processor declaration**
If a processor factory is linked to the wizard, the corresponding declarations are mixed in the argument chain. This way you will only have to declare argument values for this type of wizard / processor pair once.
- **Processor preferences**
Similarly, the preferences of an associated processor factory are mixed in the argument chain. All preferences under the node `args` will be put into the argument chain.
- **Preferences**
This is the argument section of the preferences of the wizard. The wizard stores its previous arguments in the “args” subnode of its preferences upon each call. Only declared arguments without the modifier **transient** are stored by the wizard. This way you see the most recently used argument value upon the next wizard call.
- **Caller**
Certainly we provide the caller arguments.



1.9.4 Wizards & Processor integration

In many cases the output of a wizard is the input for a processor. While separating the two helps to modularize your application (GUI and non GUI, headless versions), improve code readability and provide for more sophisticated scenarios where you use the wizard to create “argument profiles” that are reused when batch calling your processor later, it is highly desirable to have an integration for the common scenario of “run the processor after inquiring arguments from the user”.

There are two ways to accomplish this integration: First, the developer might have had a specific association in mind and has declared it for you. That’s fine, the first step is done. Alternatively you can provide a processor factory in the wizard arguments. The argument name is `processorFactory` and it accepts a factory id or the factory instance itself.

After associating the wizard with a processor factory, the processor is created and called when the wizard is finished.

If this is not the behavior you desire, you can explicitly switch off the processor call using the wizard argument `launchProcessor` (see below) and setting it to `false`.

1.9.5 Wizards & Code Exit integration

Sometimes it would be nice to start a wizard directly, via an action or a comparable event source. Its straightforward to provide a `CodeExit` type that accepts the wizard id as its source. Using

```
<perform
  type="Wizard"
  source="com.foo.bar.MyWizardFatory"/>
```

will open the referenced wizard and, by default, call an associated processor (if available) with the entered arguments. The result of the `CodeExit` is the result of the processor executed. If no processor was associated or the wizard was requested not to perform the processor, the result is the set of arguments collected by the wizard. If the wizard is cancelled, the result is null.

If you need to configure even more sophisticated scenarios, it may happen that you will need the wizard itself as the result of a `CodeExit` (for example, if you declare object extensions for a document type where the method semantics asks to return the creation wizard for this document type). So a `CodeExit` type "WizardFactory" is provided as well. Its source also references a registered wizard id. The result is the wizard instance created.

```
<perform
  type="WizardFactory"
  source="com.foo.bar.MyWizardFactory"/>
```

1.9.6 Wizards & Commandline integration

While you all already know that the fundamental concepts "Commandline", "perform" Option and `CodeExit` usage are built orthogonal, it may be useful to recap: Its quite simple to start a wizard from the commandline:

```
SignLiveCC.exe -perform -pt Wizard -ps com.foo.bar.MyWizardFactory
```

1.9.7 Standard arguments

1.9.7.1 Overview

There are a small number of standard arguments that are supported by most wizards.

1.9.7.2 launchProcessor

If you don't want the wizard to launch its associated processor after execution, you can specify 'launchProcessor=false'. The default for this argument is true'.

You can use this if you only want to collect the arguments from the user, for example to store them away or apply them on multiple processors later.

Example

```
Wizard.callArgs(  
    "foo.bar.Wizard",  
    {  
        launchProcessor: false  
    }  
);
```

1.9.7.3 viewDocument

If your wizard deals with documents, the document argument is by default opened in a viewer upon wizard execution. You can request the wizard to execute without opening the document by setting "viewDocument=false".

Example

```
Wizard.callArgs(  
    "foo.bar.Wizard",  
    {  
        document: "c:/temp/trollbat.pdf",  
        viewDocument: false  
    }  
);
```

1.9.8 The Wizard API

1.9.8.1 Overview

The wizard framework API is covered in depth by the respective **JavaDoc**. Here we will give some general information and entry points.

1.9.8.2 The outlet

The com.cabaret.claptz.common.wizard.IWizardOutlet is a registry for com.cabaret.claptz.common.wizard.IWizardFactory instances.

According to the naming conventions in Sign Live! CC, there is a singleton com.cabaret.claptz.common.wizard.WizardOutlet that will give you access to the system wide instance.

With this instance Sign Live! CC will register and lookup all wizard factories.

1.9.8.3 The factory

The factory `com.cabaret.claptz.common.wizard.IWizardFactory` is responsible for creating and managing the `com.cabaret.claptz.common.wizard.IWizard` instances. Wizard factories are declared using the extension point `com.cabaret.claptz.wizard.wizardfactories`.

1.9.8.4 The wizard

The `com.cabaret.claptz.common.wizard.IWizard` is the wizard instance itself.

1.9.8.5 Wizard tools

Accessing the wizard framework is supported by the tool class `com.cabaret.claptz.common.wizard.WizardTools`. Here you will find shortcuts for creating and processing wizard instances.

1.10 Working with documents

1.10.1 Overview

A `com.cabaret.claptz.common.document.IDocument` is the abstraction of the “data under construction”. It is the state, the file, the ... document.

Most processors work in some way on documents, either they create them (a loader or importer), they transform them (a signer or encryptor) or they simply present them in some way (the viewer or report generator).

This abstraction allows for uniform access of the platform to a variety of totally different files. While a PDF document, a license and a preferences setting are quite different, this level allows the platform to treat them all the same and give the user a common look & feel. Just drag & drop it on the application. The associated behavior (provided by a processor, of course) is triggered automatically.

The following chapter describes the default implementation for `com.cabaret.claptz.common.document.IDocument`

1.10.2 Document type declaration

1.10.2.1 Extension point

New document types are added to the platform via the extension point `com.cabaret.claptz.document.documenttypes`.

DOCUMENTTYPE

Element	Content

documenttype	Add a new document type		
		Attribute	Content
	1	class	The class name of the document type to register.
	?	id	An unique id for the document type within the platform. It is good practice to follow the Java package naming conventions. The default value for id is the value of the class attribute.
	?	preferences	The path to the preferences node for this document type
		Element	Content

ASSOCIATE

Element	Content		
associate	Add information about extensions and mimetypes for the document type		
		Attribute	Content
	1	id	The id of a registered document type. The extensions and mimetypes defined here will be associated with this document type.
	?	extensions	A “;” separated list of extensions associated with the document type. The extension itself is given as “<extension> <label>” as for example in “pdf PDF File”.

	?	mimetypes	A “;” separated list of mime types associated with the document type. The mime type is given as the mime type string as for example in “application/pdf”.
	?	defaultextension	The default extension to be used with the document type. This is given as the “<extension>” string only.
	?	defaultmimetype	The default mime type to be used with the document type.
		Element	Content

1.10.3 Processor double dispatch pattern

1.10.3.1 Processor Double Dispatch

The basic processor pattern has already been introduced:

- Lookup a factory
- Create a processor
- Make it do its work...

Now, it is very common to have a special processor for each document type, for example, if you want to use a viewer. There is a simple solution for handling this case, using “double dispatch”.

A generic processor factory, in our example a “ViewerFactory”, is created. The factory methods for the processor and the wizard are implemented dispatching to the document argument:

```
try {
    IDocument doc = (IDocument) args.get(IDocumentProcessor.ARG_DOCUMENT);
    if (doc == null) {
        throw new IllegalArgumentException("factory '"
            + getId()+ "' ARG_DOCUMENT may not be null");
    }
    // double dispatch to document
    return (CommonProcessor) doc.invoke("Viewer", args); //$NON-NLS-1$
} catch (Exception e) {
    throw new ObjectCreationException(e);
}
```

Using the object extension mechanics you can register the concrete PDFViewerFactory with the “Viewer” method in the “instrument.xml”.

```
<extension point= "com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.document.pdf.PDFDocument"
    name="Viewer">
    <perform
      type="ProcessorFactory"
      source="com.cabaret.document.pdf.PDFViewerFactory"/>
    </method>
  </extension>
```

The “invoke” instance will now perform the method “Viewer” which is mapped to a **CodeExit** of type “**ProcessorFactory**” - a **CodeExit** that will return the processor created by the factory referenced by “source”.

The following **CodeExit** types are provided for your convenience:

- **Processor**
Performs the processor referenced by “source” and returns the processor result.
- **ProcessorFactory**
Creates the processor referenced by “source” and returns the processor (without starting it).
- **Wizard**
Performs the wizard referenced by “source” and returns the arguments entered. The processor will not be created or executed.
- **WizardFactory**
Creates the wizard referenced by “source” and returns it (without starting it).

The following naming conventions have been established for the dispatch methods

- **To Processor**
Use a verb, starting with lower case (for example “view”)
- **To ProcessorFactory**
Use a noun, starting with upper case (for example “Viewer”)
- **To Wizard**
Use a verb, starting with lower case, prefixed with “wizard_” (for example “wizard_view”)
- **To WizardFactory**
Use a noun, starting with upper case, prefixed with “Wizard” (for example “WizardViewer”)

2. Advanced Use

2.1 Toolkit

2.1.1 Overview

The Toolkit provides an abstraction from the current windowing toolkits being used, for example, from SWT and AWT or even (when running headless) “none”. It provides access to common tasks and objects needed when writing interactive applications.

2.1.2 Dispatch execution

The Toolkit has methods to dispatch execution of a Runnable to the GUI thread. Most toolkits need access to all of their resources to be executed in the GUI thread. So it is wise to decorate code that may be triggered from other threads (for example in an automation context) and that will access GUI components with a “Toolkit.invoke...” flavored method.

- **Toolkit.invokeNow**
will run the Runnable in the GUI thread synchronously. The current thread (if not the GUI thread) is suspended and the GUI is requested to process the runnable. After processing, the current thread is resumed. Careful: synchronous calls to the GUI thread from other threads are prone to deadlock.
- **Toolkit.invokeLater**
will run the Runnable in the GUI thread asynchronously. The current thread (if not the GUI thread) enqueues the runnable for processing the GUI thread. Whenever the GUI thread becomes available, the Runnable is processed. The caller thread resumes immediately.
- **Toolkit.invokeUpdate**
is a combination of both. It is useful to avoid deadlocks when calling GUI code from non GUI threads. If the caller is on the GUI thread, this method behaves like “invokeNow”. If not the method

behaves like “invokeLater”, thereby preventing a deadlock that often occurs when the GUI wants to access a resource held by a worker thread while the worker thread wants to update the GUI.

```
...
Toolkit.get().invokeNow(new Runnable() {
    public void run() {
        redisplayAll();
    }
});
...
```

This piece of code ensures that “redisplayAll” will run in the context of the GUI thread.

2.1.3 Dialogs

2.1.3.1 Overview

Every window-based system has its own API for creating Dialogs and returning user input. So do we...

To abstract from the current platform, the toolkit provides a simple framework for querying user input. Input is queried using dialog objects. The toolkit renders the dialog object with respect to the current platform.

These dialog types are currently provided:

- **MessageDialog**
Prompt the user with a message and return the button pressed.
- **FileDialog**
Prompt the user to enter / select a file or directory.
- **EntryDialog**
Prompt the user to enter a string value.
- **StateMonitorDialog**
Show the state of a running process

Most dialog object have two basic interesting features:

- **Toggle**
A dialog can be requested to show a generic toggle, as, for example, when you want to give an option to the user not to display the message again. This option may be ignored by the toolkit rendering (as it is, for example, the case for SWT file dialogs).
- **Preferences**
The dialog may be associated with a preferences object. This way the values for the dialog result and the toggle are read and stored directly with a preferences object.

2.1.3.2 Message dialog

Message dialogs are used to inform the user about important application states or to let them select between a couple of choices of how the application should proceed.

The decision to inform or query the user with a message box is not as easy as it seems. Just remember your favorite operating system, asking once again for confirmation about changing the mouse position... The message dialogs give you the opportunity to implement your application both for power users and beginners by providing user feedback about the dialog itself. Use this feature!

```
...
Dialog dialog = MessageDialog.createYesNo(
    null,
    "title",
    "Do you really want to hurt me?",
    Dialog.RC_NO);
dialog.open();
if (dialog.getResult() == Dialog.RC_NO) {
    return;
}
hurt();
...
```

- This is a simple example of how to create a basic message dialog.

```
int hurt = Dialog.RC_NO;
boolean dontAskAgain = false;
...
Dialog dialog = MessageDialog.createYesNo(
    null,
    "title",
    "Do you really want to hurt me?",
    hurt,
    "Don't ask me again",
    dontAskAgain);
hurt = dialog.openIfToggleOff();
dontAskAgain = dialog.isToggleValue();
if (hurt == Dialog.RC_NO) {
    return;
}
hurt();
...
```

This example provides a message dialog with a toggle checkbox. The code gets a little cluttered with all the necessary states - even if the code necessary to make the decision persistent is omitted here.

The last example provides a simple solution to save your state using preferences. As the storage is decoupled using an interface, you should not encounter problems migrating this to other scenarios.

```
...
Dialog dialog = MessageDialog.createYesNo(
    null,
    "title",
    "Do you really want to hurt me?",
    Dialog.RC_NO,
    "Don't ask me again",
    false);
dialog.setResultCode(new IntegerPrefValueHolder(getPreferences(),
    "dlgResultHurt", Dialog.RC_NO));
dialog.setToggleValue(new BooleanPrefValueHolder(getPreferences(),
    "dlgToggleHurt", false));
dialog.openIfToggleOff();
if (dialog.getResult() == Dialog.RC_NO) {
    return;
}
hurt();
...
```

The state is read and written automatically from the preferences store defined in the *de.intarsys.tools.valueholder.IValueHolder* object.

2.1.3.3 File dialog

File dialogs are used to select files and directories for input and output operations. Be aware that the “toggle” behavior is currently not supported for file dialogs.

```
...
FileDialog dialog = FileDialog.createDirectory(
    null,
    "My Application",
    "Select directory",
    getTempDir());
dialog.open();
if (dialog.getResultName() == null) {
    return;
}
...
```

This example lets the user select a directory.

```
...
FileDialog dialog = FileDialog.createFilenameLoad(
    null,
    "My Application",
    new String[] { "*.txt" },
    new String[] { "Text File" },
    getTempDir());
dialog.open();
if (dialog.getResultName() == null) {
    return;
}
...
```

This example lets the user select a file from within “getTempDir()”. A filter for “*.txt” files is predefined.

```
...
FileDialog dialog = FileDialog.createFilenameSave(
    null,
    "My Application",
    new String[] { "*.txt" },
    new String[] { "Text File" },
    getTempDir(),
    "new.txt",
    true );
dialog.open();
if (dialog.getResultName() == null) {
    return;
}
...
```

This example lets the user enter a filename for saving. A filter for “*.txt” files and a default filename “new.txt” is predefined. After entering a filename, the system will check if the filename already exists and, as necessary, overwriting has to be confirmed by the user.

2.2 Script Integration

2.2.1 Overview

Sign Live! CC comes with sophisticated scripting support. As always, you can use this on many levels:

- The plain intarsys scripting abstraction
This is very similar to the now standard javax.script component. The reason we use this framework is
- It already exists and we have a lot of experience with it and implementations that depend on it.
- It can be easily bridged in both ways
- It has some very interesting features that go beyond the design of javax.script. This is, for example, the cross scripting between any

registered engines, even mixed in a single call stack, automatic engine selection and much more.

- The claptz standard binding
- The CodeExit abstraction

A very good introduction to this topic you can find in the Sign Live! CC Scripting Tutorial.

2.2.2 Language Bindings

The scripting framework is capable of supporting multiple “language” transparently. The language appropriate for a given script is derived from its extension or defined explicitly by the caller.

Currently the following bindings are included.

2.2.2.1 JavaScript

This is the workhorse of scripting within Sign Live! CC. Many GUI element callbacks are implemented this way, but also complex features in some product components.

The JavaScript engine is provided by Mozilla and is known as Rhino.

2.2.2.2 JavaScript Templates

Powered by JavaScript Sign Live! CC provides a simple but powerful template engine to create plain text efficiently. All reports within the system are implemented using this language.

2.2.3 Environment and configuration

2.2.3.1 Overview

This chapter describes the system environment for running scripts and gives some insight in installation and configuration.

2.2.3.2 Search path

The scripting framework supports the definition of global search paths for looking up scripts.

Defaults are provided by the respective language binding. To define additional search paths you can choose out of two possibilities:

Add your path using the preferences dialog on the “Scripting” page. The definition is a “;” separated list of directories. All paths are interpreted relative to “\${environment.basedir}”. This search path has the highest precedence and is valid for all language bindings.

Add an extension to “com.cabaret.scripting.executableresolver”.

An example

```
<extension point="com.cabaret.scripting.executableresolver">
  <executableresolver
    class="com.cabaret.scripting.common.SearchPathResolver"
    searchpath="f:/foo/scripts"/>
</extension>
```

A searchpath defined this way will not be visible in the preferences dialog. The search path defined is valid for all language bindings and is visited **after** the search paths defined in the preferences page.

A relative definition will be resolved relative to the instrument base directory.

2.2.3.3 Initialization scripts

Often it is necessary to define global functions or variables upfront. The exact timing for execution of the scripts is not defined but is guaranteed to be before execution of any other script ("lazy" execution).

Defaults are provided by the respective language binding. To define additional global scripts you can choose out of two possibilities:

- Add a path using the preferences dialog on the "Scripting" page, entry "global scripts". The definition is a ";" separated list of directories. All paths are interpreted relative to the stage base directory. All scripts in all paths are guaranteed to be executed at the initialization time of the scripting framework. This global path has the highest precedence and is valid for all language bindings.
- Add an extension to "com.cabaret.scripting.globals". You can define here a directory with script to be executed, a single script or even literal code.

Example (literal code)

```
<extension point="com.cabaret.scripting.initscripts">
  <perform type="JavaScript" source='MyGlobal = "foo";' />
</extension>
```

Example (single script)

```
<extension point="com.cabaret.scripting.initscripts">
  <load path="./scripts/LibAnnots.js" />
</extension>
```

An init script defined this way will not be visible in the preferences dialog. The init scripts defined are valid for all language bindings and are visited **after** the init scripts defined in the preferences page.

A relative definition will be resolved relative to the instrument base directory.

2.2.4 Script Center

2.2.4.1 Overview

The script center is a simple and intuitive tool for development, management and all day use of scripts.

2.2.4.2 Script development

A main concern of the script center is script development. The script center will ease considerably creation, test and maintainance of script code.

Normally script code is deployed with an instrument, the plugin technology for Sign Live! CC. This is what renders them flexible and powerful. But to develop scripts in this environment is tedious. Instrument declarations tend to get long and hard to read. On every change in the declaration a reload is needed.

The script center allows for script development in small dedicated script files, execution and test in an environment that is similiar to the later production environment and “hot code replacement” - changes in the script are available for execution immediately.

2.2.4.3 Deployment

In a production environment it may be convenient to use the script center as a “tool box” for the execution of individual extensions to the system. You can offer your scripts in a clean, user definable structure.

2.2.5 CodeExit integration

2.2.5.1 Overview

CodeExit is the basic building block for new Sign Live! CC features. A CodeExit defines a single behavior, just as a method or procedure does. We have already seen many ways of implementing such a CodeExit, for example calling operating system processes or static Java code.

As you may guess, a CodeExit can also call into the scripting framework. This way you can for example define an action for a menu using any of the supported scripting languages.

The CodeExit has two main integration “type”s:

- **Script**
- **ScriptFile**

These are abstractions from script code written in some language binding and stored on the files system. The CodeExit implementation will look up the scripts, compile and execute them. While the compiled script code is cached, hot code deployment is supported!

“Script” is a logical name that will be looked up in all defined search paths. The first script with a matching name in the search path in any language binding is loaded.

“ScriptFile” is the physical file name of a script to be loaded and executed.

There may be further `CodeExit` types that are provided by the respective language bindings (for example “JavaScript” supporting literal JavaScript code). These are explained in the respective chapters.

2.2.6 Application globals

2.2.6.1 Overview

To support effective scripting, Sign Live! CC publishes the most important application objects as globals. This enables easy access with readable code. How to access this global depends on the language binding, for example with JavaScript you would access the processor outlet like that.

```
...
jProcessorOutlet.lookupProcessorFactory("foo.bar");
...
```

2.2.6.2 jApplication

Access the VM singleton for the application.

2.2.6.3 jProcessorOutlet

Access the VM singleton

`com.cabaret.claptz.common.processor.IProcessorOutlet`.

2.2.6.4 jWizardOutlet

Access the VM singleton

`com.cabaret.claptz.common.wizard.IWizardOutlet`.

2.2.6.5 jDocumentOutlet

Access the VM singleton

`com.cabaret.claptz.common.document.IDocumentOutlet`.

2.2.6.6 jVariables

Access the currently defined variables. *jVariables* is the current VM wide *de.intarsys.tools.variable.IVariableNamespaces* instance. From here you can access a single *de.intarsys.tools.variable.IVariableNamespace* using *getNamespace*. From the namespace you can access a single variable using *getVariable*.

Example

```
...  
var tempValue =  
jVariables.getNamespace("mycompany").getVariable("webserver");  
...
```

2.2.6.7 jPreferences

Access the `de.intarsys.tools.preferences.IPreferences` node that is returned from `de.intarsys.tools.preferences.IPreferencesFactory.getMain()`. This is the default application preferences node.

2.2.6.8 jPreferencesRoot

Access the `de.intarsys.tools.preferences.IPreferences` node that is returned from `de.intarsys.tools.preferences.IPreferencesFactory.getRoot()`. This is root node for all preferences.

2.3 Javascript Binding

2.3.1 Overview

JavaScript is an object oriented programming language, well suited and often used as an embedded scripting language in other applications or for embedding in dynamic documents. Technically there is no relation to "Java", only the syntax will remind you of this language. Java is a product of "SUN Microsystems". JavaScript stems from a Webbrowser extension provided by "Netscape". JavaScript is used standardized under the name "ECMA Script". Implementations conformant to this standard are commonly called "JavaScript" (wich indeed is a trademark of Sun Microsystems).

As JavaScript is a widespread technology there is lots of information, tutorials and tools around. Only drawback is that most of them are related to HTML documents and use in Web browsers.

2.3.2 Syntax and semantics

This is not a JavaScript tutorial or manual.

This is why you will not find a complete discussion on syntax or semantics. You will find the language reference at

<http://www.ecma-international.org/publications/standards/Ecma-262.htm>

More good documentation is available at

<https://developer.mozilla.org/En/JavaScript>

2.3.3 JavaScript engine

JavaScript support is implemented in Sign Live! CC using the **Rhino** JavaScript engine. This is an implementation provided by Mozilla. The version currently used is Rhino 1.7R2, which should be conformant to JavaScript 1.7 (ECMA-262 3. edition plus some extras + E4X).

Rhino itself is completely implemented in Java and a perfect candidate for embedded use. Many thanks to the implementers and supporters at Mozilla.

Rhino (and so Sign Live! CC) leave the ECMA spec behind in many ways. The most important feature is called “LiveConnect” and enables seamless switching between Java and JavaScript. Objects from each programming model are available for use in the other one.

The definitive site for information about Rhino is

http://developer.mozilla.org/en/docs/Rhino_documentation

For many of the topics here you will find more detailed information there.

2.3.4 Scripting framework binding

2.3.4.1 Discovery

JavaScript registers itself as responsible for scripts ending in “.js” within the scripting framework. These scripts are loaded, compiled and executed upon a call. The execution semantics is that of a plain script - not a function. This is why you can not “return” from such a script!

2.3.4.2 this

While JavaScript is an object oriented language and as such has a “receiver” or “implementor” for the scripts, you should treat the special variable “this” as being undefined most of the times as far as calls from the scripting framework are concerned. Unless explicitly stated the receiver context for executing a script can change without notice.

If you want to access Sign Live! CC context information, please use only the arguments and globals provided by the embedding system as documented in the respective context.

2.3.5 Reflection

“Reflection” is commonly understood as the ability to access information about the system, its objects and the runtime state itself (as opposed to the state held within the objects, programmed and accessible by your methods).

For example, the name and physical location of the script currently running is reflective information.

This kind of information is indispensable for complex or generic scripts, thats why the scripting framework provides such information. This

feature is mapped in the JavaScript runtime using the following global variables

- Reflector
- Resolver

The integration with other key concepts as “Script”, “CodeExit” or “Processor” is also provided in a generic way that disburdens you from the complex APIs of creating and performing such an object.

- Script
- ScriptFile
- CodeExit
- Processor
- Wizard

2.3.5.1 Reflector

The Reflector object provides information about the script’s runtime environment.

2.3.5.1.1 argAt

```
Object argAt(Number index [, Object default])
```

Argument at index index (0 based) or default if arguments are not index (but named) or index does not exist

Example

```
var argValue = Reflector.argAt(0);
```

get the first argument to the running script

2.3.5.1.2 argCount

```
Integer argCount()
```

Number of arguments to the running script

Example

```
var count = Reflector.argCount();
```

2.3.5.1.3 argNamed

```
Object argNamed(String name [, Object default])
```

Get the argument with name “name” or return “default” if it does not exist.

Example

```
var argValue = Reflector.argNamed („foo“);
```

2.3.5.1.4 argNames

```
List argNames()
```

The names of all arguments to the current script

Example

```
var names = Reflector.argNames();
```

2.3.5.1.5 args

```
IArgs args()
```

The arguments to the current script in their object representation. In Java this is a `de.intarsys.tools.functor.IArgs` instance.

Example

```
var args = Reflector.args();
```

2.3.5.1.6 declareArg

```
void declareArg(String name [, Object default])
```

Declare the usage of an argument “name” where a “default” value is used if not available. The argument declared can be used in the current script like any other local variable.

Example

```
Reflector.declareArg('param');  
if (param = "test"){  
  ...  
}
```

2.3.5.1.7 isArgIndexed

```
Boolean isArgIndexed()
```

“true” if the script arguments are indexed. Be aware that arguments can be both indexed AND named.

Example

```
if (Reflector.isArgIndexed()){  
  var arg = Reflector.argAt(0);  
}
```

2.3.5.1.8 isArgNamed

```
Boolean isArgNamed()
```

“true” if the script arguments are named. Be aware that arguments can be both indexed AND named.

Example

```
if (Reflector.isArgNamed()){  
  var arg = Reflector.argNamed("foo");  
}
```

2.3.5.1.9 locator

```
ILocator locator()
```

Get the location to the script source.

Example

```
var source = Reflector.locator();
```

2.3.5.2 Resolver

Resolver is a convenience method to access other scripts in a more readable way.

```
Resolver.wazamba("test");
```

Search and call the script “wazamba”. Search order is as follows

- In the directory where the calling script resides
- In all scripting framework search paths.

This is equivalent to calling

```
Script.call("wazamba", "test");
```

2.3.5.3 Script

The Script object gives way to calling other scripts in the framework

2.3.5.3.1 call

```
Object call(String scriptName [, Object arg]*)
```

Call the script with the logical name “scriptName”. The script is searched first relative to the running script, after that in the scripting framework search paths. All other arguments are forwarded as arguments to the script called.

Example

```
Script.call("hello", "world");
```

The script “hello” (which is not necessarily a “.js” one!!) is looked up and executed with the argument “world”.

2.3.5.3.2 callArgs

```
Object callArgs(String scriptName, Object args):
```

Call the script with the logical name “scriptName”. The script is searched first relative to the running script, after that in the scripting

framework search paths. The arguments defined by “args” are forwarded to the called script.

“args” can be a list or a plain JavaScript object for named arguments.

Example

```
var args = [ "world" ];  
Script.callArgs("hello", args);
```

2.3.5.4 ScriptFile

2.3.5.4.1 call

```
Object call(String scriptFileName [, Object arg]* ):
```

Lookup, compile and run the script stored in “scriptFileName”. The name is interpreted relative to the working directory of the application. In some special cases the context may support further root paths that are tried in sequence when a relative name is given.

All other arguments are forwarded as arguments to the script called.

Example

```
ScriptFile.call("c:\\temp\\hello.js", "world");
```

2.3.5.4.2 callArgs

```
Object callArgs(String scriptFileName, List argList ):
```

Lookup, compile and run the script stored in “scriptFileName”. The name is interpreted relative to the working directory of the application. In some special cases the context may support further root paths that are tried in sequence when a relative name is given.

The arguments defined by “args” are forwarded to the called script. “args” can be a list or a plain JavaScript object for named arguments.

Example

```
var args = [ "world" ];  
ScriptFile.callArgs("c:\\temp\\hello.js", args);
```

2.3.5.5 CodeExit

You can call a CodeExit directly from JavaScript code using CodeExit

2.3.5.5.1 call

```
Object call(String type, String source[, Object arg]*)
```

Call the **CodeExit** with type and source. All other arguments are forwarded as arguments.

Example

```
CodeExit.call("Functor", "com.foo.MyFunctor", "test");
```

An **IFunctor** `com.foo.MyFunctor` is instantiated and executed with “test” as its argument.

2.3.5.5.2 callArgs

```
Object callArgs(String type, String source, Object args):
```

Call the **CodeExit** with type and source. The arguments defined by “args” are forwarded.

“args” can be a list or a plain **JavaScript** object for named arguments.

Example

```
var args = [ "world" ];  
CodeExit.callArgs("Functor", "com.foo.MyFunctor", args);
```

2.3.5.6 Processor

You can call a **Processor** directly from **JavaScript** code using the **Processor** type.

2.3.5.6.1 call

```
Object call(String processorFactoryId [, Object arg]*)
```

Create and call a processor using `processorFactoryId`. All other arguments are forwarded as arguments.

Example

```
Processor.call("DocumentOpenerFactory", "c:/test/mypdf.pdf");
```

"c:/test/mypdf.pdf" is opened.

2.3.5.6.2 callArgs

```
Object callArgs(String processorFactoryId, Object args):
```

Create and call a processor using processorFactoryId. The arguments defined by "args" are forwarded.

"args" can be a list or a plain JavaScript object for named arguments.

Example

```
var args = [ "c:/test/mypdf.pdf" ];  
Processor.callArgs("DocumentOpenerFactory", args);
```

2.3.5.7 Wizard

You can call a Wizard directly from JavaScript code using the Wizard type.

2.3.5.7.1 call

```
Object call(String wizardFactoryId [, Object arg]*)
```

Create and call a wizard using wizardFactoryId. All other arguments are forwarded as arguments.

Example

```
Wizard.call("MyWizardFactory", "arg1");
```

2.3.5.7.2 callArgs

```
Object callArgs(String wizardFactoryId, Object args):
```

Create and call a wizard using wizardFactoryId. The arguments defined by "args" are forwarded.

"args" can be a list or a plain JavaScript object for named arguments.

Example

```
var args = [ "arg1" ];
Wizard.callArgs("MyWizardFactory", args);
```

2.3.6 CodeExit Integration

Besides the standard integration in the Scripting Framework via the “Script” and “ScriptFile” types there is support for literal CodeExits using the type “JavaScript”.

```
<perform type="JavaScript" source="app.alert('hello, world')"/>
```

is an example we can not spare you from reading a lot of times in this book..

2.3.7 Tipps & Tricks

2.3.7.1 String expansion

It may be necessary to use the string expansion technique within your scripts to get access to system state. You can do this via the VM singletons `de.intarsys.tools.expression.TemplateEvaluator.get()` and `de.intarsys.tools.expression.ExpressionEvaluator.get()`. This will give you access to the evaluation context established Sign Live! CC. Be aware that this is the global evaluation context - you can not expect to get information from your servlet or processor here. In these cases you have to select the string evaluation context associated with these instances.

Example

```
var args = Packages.de.intarsys.tools.functor.Args.EMPTY;
var string =
Packages.de.intarsys.tools.expression.TemplateEvaluator.get().evaluate
("${stage.basedir}", args);
app.alert(string);
```

2.3.8 LiveConnect features and pitfalls

2.3.8.1 Overview

Using LiveConnect is a wonderful thing most of the time. But, you should be aware of some pitfalls all the time.

2.3.8.2 Type system

The “typeof” operator will return “object” for Java objects, exactly as it does for JavaScript.

```
typeof(jApplication) // == "object"
```

It may be confusing, but it is **not** true, that every Java object is of JavaScript type “Object”, using the “instanceof” operator.

```
jApplication instanceof Object // == false
```

Instead it is

```
jApplication instanceof java.lang.Object // == true
```

Accept this as a feature, in earlier versions you had no way of a Java class based “instanceof” check. Now you can use this operator just as you would in Java.

```
jApplication instanceof  
Packages.com.cabaret.claptz.application.model.IApplication // == true
```

2.3.8.3 Class access

Via LiveConnect you can access Java classes via the “Packages” global.

```
Packages.de.intarsys.tools.file.FileTools
```

You can call static methods, access static properties and create new objects.

```
var name =  
Packages.de.intarsys.tools.file.FileTools.getBaseName("foo/bar.txt");  
  
var args = Packages.de.intarsys.tools.functor.Args.EMPTY;  
  
var locator = new  
Packages.de.intarsys.tools.locator.FileLocator("foo/bar.txt");
```

The reflection methods of “java.lang.Class” are not available here, but you can use this other class representation also:

```
var fileToolsClass =  
java.lang.Class.forName("de.intarsys.tools.file.FileTools");
```

Be aware that the two “de.intarsys.tools.file.FileTools” representations are different objects.

2.3.8.4 Implementing Java Interfaces

LiveConnect allows the implementation of interfaces in JavaScript.

You do this in a two step way. First you define a Java Script object with function members for each interface method.

```
var listenerMethods = {  
    handleEvent: function(event) {  
        app.alert("oops");  
    }  
};
```

Then you create an instance (!) of the Java interface. Internally this will be mapped in the creation of an on-the -fly Java class implementing the interface using the functions of the JavaScript object as its implementation.

```
var listenerObj = new  
Packages.de.intarsys.tools.event.INotificationListener(listenerMethods  
);
```

You can use this object now

```
var processor =  
Packages.com.cabaret.claptz.common.processor.ProcessorTools.createProc  
essor(  
    "DocumentOpenerFactory", {}  
);  
var event = Packages.de.intarsys.tools.event.OkEvent.ID;  
processor.addNotificationListener(event, listenerObj);
```

You will get notified upon successful termination of the opener processor.

More information an this implementation technique you will get at <http://www.mozilla.org/rhino/ScriptingJava.html>

2.3.8.5 Primitive types

With “LiveConnect” it is quite easy to get confused by the different primitive objects of the two languages, for example a Java string and a JavaScript string.

Switching from Java to JavaScript (and vice versa) will automatically “marshall” to the primitive type of the other language. If you use JavaScript to get a String typed property from a Java object, you will deal with a JavaScript String. You can not call “endsWith” on this String - you will get an error “endsWith is not a function”.

```
var javaObject = ...;
var resultString = javaObject.getName();

// Runtime Error!

if (resultString.endsWith("test")) {
    ...
}
```

2.3.8.6 Property access (Getter and Setter)

LiveConnect lets you seamlessly access Java objects. Java methods will be mapped to **JavaScript** calls automatically. The Java method

```
public void foo(String stringValue, int intValue) {
    //
}
```

will be called from **JavaScript**

```
object.foo("bar", 42);
```

Methods that comply to the Bean notation can be further abbreviated, just omit “get” and “set” and access the property like a **JavaScript** property (no parantheses). If there is no setter, the property is assumed read only.

```
private String name;
private boolean ready;

public void setName(String pName) {
    name = pName;
}

public String getName() {
    return name;
}

public void setReady(boolean pReady) {
    ready = pReady;
}

public boolean isReady() {
    return ready;
}
```

can be accessed in **JavaScript** using

```
if (object.ready) {  
    object.name = "foo";  
}
```

2.3.9 Best practices

2.3.9.1 Variable declaration

JavaScript supports the implicit declaration of variables upon an assignment.

```
Foo = "bar";
```

If “Foo” is not present in the current scope, it will be created in the root scope. This way “Foo” is accessible from now on in all Scripts. Most of the time this is not the intention. Be at least aware of possible memory leaks.

Using

```
var Foo = "bar";
```

the variable is stored in the local scope (the “this” object). It does not matter if there is a variable with the same name available in some parent scope. This way you will not get accidentally in conflict with properties having the same name. Subsequent access to “Foo” will always use the local scope.

If you use variables in your scripts, you should always declare them using “var”.

2.3.9.2 Constants

To use constants in JavaScript it would be best to declare them on application level. This way they are visible to all scripts (and language bindings). You can accomplish this as described in the chapter on the scripting framework (using the extension point “com.cabaret.scripting.initscripts”).

If you define globals you should follow a common rule, either use a prefix or a “singleton” object.

```
// use prefix
ZPT_dwam= 1;
ZPT_dwup = 2;

// use common object
ZPTConst = new Object();
ZPTConst.dwam = 1;
ZPTConst.dwup = 2;
```

2.3.9.3 XML embedding

Sign Live! CC relies heavily on XML files and allows for the definition of **CodeExit** there. Using the literal type “JavaScript” you can easily embedd JavaScript code.

This comes with some drawbacks.

An XML parser may have its own thoughts on how to use whitespace - it gets normalized.

Code of the form

```
foo();
// look out for this clever code following!
bar();
```

will reach the application as

```
foo(); // look out for this clever code following! bar();
```

what is clearly not what was intended! Use always comments o the form `/**/`.

Quotes are used in XML to separate attribute values. In JavaScript they are used to delimit strings. It is quite easy to mess it up.

If you absolutely have to, be careful to use single quotes with XML and double quotes with JavaScript (or vice versa).

Will Work:

```
<perform type="JavaScript" source="app.alert('hello')"/>
```

Will Work:

```
<perform type="JavaScript" source='app.alert("hello")' />
```

Won't Work:


```
<perform type="JavaScript" source="app.alert("hello")"/>
```

Besides this technical drawback, it will get very hard to maintain large instrument definitions containing large JavaScripts.

Last not least - upon a change in the instrument file you must restart, while the scripting framework supports hot code deployment.

So, in most but the trivial cases you will be better of writing

```
<perform type="Script" source="foo"/>
```

and add a script “foo.js” at an appropriate place. In an instrument context for example, the instrument base directory is always automatically added to the search path.

2.3.10 Common Error

2.3.10.1 missing name after . operator

This error message from the JavaScript compiler claims a syntax error where you ommited a property name, for example

```
...  
tempFoo. = 2;  
...
```

But what if you are sure that you have a correct syntax, like in

```
...  
tempFile.delete();  
...
```

Well, this is a typicall LiveConnect problem. `delete` is a reserved token in JavaScript, so the compiler doesn't see a property!

In a situation like this you must have a look in the bag of tricks: as everything in a JavaScript object, even a function, is a member of the object, you can access `delete()` like this:

```
...  
tempFile["delete"]();  
...
```

Looks funny, isn't it.

2.3.10.2 Java class “foo” has no public instance field or method named “bar”

In most cases this can easily be tracked down to misspelling or navigation faults.

But sometimes there is no obvious reason for the failure. In this case it may be that you try to access a property that is available in some parent scope in an inappropriate way (name clash).

If for example you try to write to a variable you without declaring it, this is a common failure:

```
..  
log = "a useful message";  
..
```

This will fail if “log” is a property of “this” or some object in the scope chain that is read only. Just use always “var”.

```
..  
var log = "a useful message";  
..
```

2.3.10.3 **TypeError: foo is not a function.**

In most cases you misspelled an access to a property. An object with property “foo” can most of the times be access via

```
..  
object.getFoo();  
..
```

or

```
..  
object.foo;  
..  
..  
object.foo();  
..
```

will lead to the error message above.

Nearly as popular is accessing a would-be Java String with Java methods. Most of the time the error is caused because you have to deal with JavaScript Strings!

2.4 JavaScript Template Binding

2.4.1 Overview

Templates are “programming the other way round”. They are suitable for solving a class of problems where textual output is needed, interleaved with dynamic information. A typical usage scenario you will find in web applications where the html page is often built using template engines, for example Java Server Pages.

Sign Live! CC supports a template engine based on the JavaScript engine that is also included and described in a previous chapter. The template syntax resembles that of Java Server Pages, the expression syntax is plain JavaScript. No new concepts.

2.4.2 Motivation

A simple template will be

```
Hello, world.
```

“Evaluating” this template will result in “Hello, world”. At least no surprise here...

A little bit more useful

```
Hello, <%= user %>
```

Provided somewhere around the variable “user” is defined as “foo” you will see “Hello, foo”. From here, the direction should be clear: instead of writing lot of statements that add literal text to some stream object, just write the plain text and include the dynamic content in a special syntax.

2.4.3 Syntax

2.4.3.1 Argument

```
"<%$" name [":" defaultValue] "%>"
```

Declares “name” as an argument to the template. This is much the same as “Reflector.declareArg()”. From now on “name” can be used wherever an expression can be inserted.

Nothing is copied to the generated output.

2.4.3.2 Comment

```
"<%-" any chars "%>"
```

Treat all text between <%- and %> as comment and ignore. Nothing is copied to the generated output.

2.4.3.3 Expression

```
"<%= " JavaScript Expression "%>"
```

Evaluate the JavaScript expression and put the result, converted to a string to the generated output.

2.4.3.4 Code

```
"<% " JavaScript code fragment "%>"
```

This is the most powerful and dangerous one. The JavaScript code or code fragment is compiled and executed. This allows conditional output, repetitive output and so on. Any legal JavaScript code is allowed.

At this point the semantics of JavaScript Template should be defined.

A JavaScript Template is processed as if any literal text is replaced by a “output.write(literal);”. Any expression escape is replaced by “output.write(expression)”. The code escape is processed by literally copying the code. At the end this results in a (hopefully) syntactically correct JavaScript program that will be executed.

This is also where the dark side of this template implementation gets visible. If a syntax error occurs, it might get difficult to track back as we can not provide a correct mapping back to the template yet.

2.4.3.5 Directive

```
"<%@ " directive [ ":" value ] "%>"
```

directive is not case sensitive.

“Directives” are a general means for including directives to some context in the code that are not processed by the interpreter itself.

An example of a directive is

```
<%@ locatortype:html %>\
```

indicating that the outcome of the processing should be interpreted as if it had an extension “.html”.

2.4.3.6 Escaping

Using “\” (the backslash) you can escape the following character. The following escapes are defined:

- \\ The backslash itself

- `\n` A newline
- `\r` A carriage return
- `\t` A tab
- `\<any whitespace>` Ignore the whitespace. This way you can ignore line breaks in you template, allowing you to keep it readable.

2.4.4 Directives

2.4.4.1 locatortype

Set the locator type to be used with results from this template.

2.4.4.2 contenttype

Set the content type (mime type) to be used with results from this template.

2.4.5 Examples

```
Hello <%= recipient%>,  
  
<% if (greeting) { %>  
cheers, Michael.  
<% } else { %>  
asap!  
<%} %>
```

Assuming “recipient” is “Brudfug” and “greeting” is true, you will see

```
Hello Brudfug,  
  
cheers, Michael.
```

Gee, that is not what we wanted. Forgot the escapes.

```
Hello <%= recipient%>,  
  
<% if (greeting) { %>\  
cheers, Michael.\  
<% } else { %>\br/><%} %>
```

will result in

```
Hello Brudfug,  
cheers, Michael.
```

3. APIs

3.1 APIs

3.1.1 Overview

This section will give you insight on how to control the application at the programming level.

Out of the box we support a broad range of concepts and protocols such as:

- **Commandline integration**
Calls the application “one way”, handing all parameters as a commandline string. In addition to the standard commandline behavior you may be comfortable with, Sign Live! CC publishes its scripting framework at the commandline, giving you unlimited possibilities for external control.
- **Java embedded.**
There are multiple different approaches for embedding Sign Live! CC within your own Java application. You can leverage the generic API features found herein or even go down to the basic component APIs where you can control simply everything.
- **Shared Library**
At the moment of this release, shared library support is tested and released only for Windows platforms. Please contact our support team for further information. We provide a generic, reflective gateway for integrating Sign Live! CC within any context supporting shared libraries. The “methods” published by this API are declared using the “Functor” concepts of Sign Live! CC.
- **ActiveX**
The standard Windows API is provided as a generic, reflective gateway for accessing Sign Live! CC document instances. The “methods” published by this API are declared using the “Functor” concepts of Sign Live! CC.

- **HTTP**
- **XMLRPC**
- **SOAP**
- **Other protocols**
are easily included in the framework. Try it yourself or ask us about them - they may already be part of the team...

3.1.2 SDK

The complete SDK is deployed in the “sdk” subdirectory of your installation. You should find there anything you need to develop a client to one of the APIs in this document part.

- Client stubs
- Example code
- Example instruments

The directory is organized as follows

- **sdk** All the stuff relevant to interfacing Sign Live! CC
 - **<protocol>** The clients for a specific protocol, for example “HTTP”
 - **demo** Demo code and instruments for this protocol flavor
 - **<demo name>** An example implementation
 - **src** Source code examples for interfacing using this protocol
 - **bin** Binaries to run the demo clients
 - **instruments** Instrument declarations for running the example
 - **dev** Stuff that is related to the development of the client
 - **include** C include files (if applicable)
 - **vcproj** Visual Studio C related development artifacts.
 - **readme.txt** Some information related to the use of the protocol

3.2 Generic API Service Object

3.2.1 Overview

As Sign Live! CC is a very flexible platform, an adequate design is needed for accessing it via an API.

It's plain impossible to forward all useful behavior to an external dedicated API and as we can not foresee the requirements of a potential client we do not want to define a subset of the features to be published.

Instead, for the external APIs we opted for a highly extensible, generic approach. The API is modelled with a reflective signature similar to

```
Object call(String name, Object[] args) throws Exception;
```

meaning that we want to execute the function “name” with arguments “args”. The behavior is expected to be implemented by a server object in a simple, procedural way.

The trick is that the server object does not come with a predefined set of features. Instead, all functions are attached to the server object in a declaration. This way you can design your own, optimal subset of functions.

This generic technique is available to nearly all protocols

- Embedded Java
- Shared Library
- ActiveX
- HTTP
- XML RPC
- SOAP

3.2.2 Mechanics

The behavior of a server object is modeled using the functor design pattern that you find all over in Sign Live! CC. The functor is attached to the server object using a simple declaration and is now part of the published behavior of the server object. The functor is an instance of *de.intarsys.tools.functor.IFunctor*, an example for that is the already presented *CodeExit* with all its *type* flavors.

Now, if a function is requested from the server object, it will look up the implementation in this registry and execute the associated functor, forwarding the incoming named and indexed arguments. The functor result is returned. The functor arguments can be converted to named arguments using the *declaration* feature of the *CodeExit*.

3.2.3 Declaration

To declare a function for a server object you use the extension point *com.cabaret.claptz.objectmodel.members*. You use the *method* element for a new method member. The attribute *implementor* designates the server object class to which we will attach the method. For the generic API use “com.cabaret.api.GenericAPI”. *name* defines the name of the

method to be published. The argument signature is determined by the attached functor. To keep things simple, overloading methods is not supported.

Example

```
<extension point="com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.api.GenericAPI"
    name="hello">
    <perform type="JavaScript" source="app.alert('hello');"/>
  </method>
</extension>
```

An example for named arguments

```
<extension point="com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.api.GenericAPI"
    name="hello">
    <perform type="JavaScript" source="app.alert(message);">
      <declarations>
        <arg name="message"/>
      </declarations>
    </perform>
  </method>
</extension>
```

Now, an object of type *com.cabaret.api.GenericAPI* supports the invocation of method *hello* by simply forwarding it to the declared *CodeExit*.

The server object instance to be used for the different APIs, the exact invocation signature and argument marshalling procedures depends on the protocol implementation. You will find further information in the respective chapters.

3.2.4 Modifiers

Along with the methods you can declare “modifiers” that will fine tune the behavior of the execution. To do this, add the attribute modifiers to the method element. By convention, modifiers are lowercase tags separated by a “;”. The semantics of the modifiers are up to the implementation using the object model.

```

<extension point="com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.api.GenericAPI"
    name="hello"
    modifiers="gui">
    <perform type="JavaScript" source="app.alert(message);">
      <declarations>
        <arg name="message"/>
      </declarations>
    </perform>
  </method>
</extension>

```

If a modifier is supported depends on the protocol implementation and will be explained in the respective paragraphs.

3.2.4.1 Modifier “oneway”

This modifier indicates that there is no result for this method. The caller will not wait for the computation to finish and does not expect any information about the ending or outcome of the computation.

Note that this is different from an asynchronous call semantic, as there still may be a “callback” or other notification available.

3.2.4.2 Modifier “gui”

The API implementations in this chapter use the modifier `gui` to indicate that the execution of the method has to be performed in the GUI thread, regardless of the caller thread.

3.2.4.3 Modifier “synchronized”

Indicate that no two threads may enter this method (or any other method of the service object marked `synchronized`) at the same time.

3.2.5 Object selector

There is a nifty feature with the object model we have not mentioned yet. The object model classification is not tied to java classes - it's only a simple and convenient default. With the object model you can select special instances to be the target of your declaration. The “selector” is simply a string id separated by a “:” from the class name.

```

<method
  implementor="com.cabaret.api.GenericAPI:SharedLib"
  name="hello">
  <perform type="JavaScript" source="
    app.alert('oops');
    'oops';
  "/>
</method>

```

This declaration attaches the implementation for “hello” to a dedicated API object with the id “SharedLib”.

The exact lookup sequence in the object model is as follows:

- Search an implementation for the class + instance id
- Search an implementation with a matching instance id up in the class hierarchy
- Search an implementation for the class itself (without instance id)
- Search an implementation for a class (without instance id) up the class hierarchy
- Fail if nothing found.

Not every API implementation is able to select between different instances, see the respective chapter on your protocol for more information.

For those protocols supporting instances, the selector is always created as follows:

```
com.cabaret.api.GenericAPI:<protocol name>
```

The concrete value depends on the protocol implementation. For example, a generic HTTP API call would try to lookup in this order

```
com.cabaret.api.GenericAPI:HTTP
java.lang.Object:HTTP
com.cabaret.api.GenericAPI
java.lang.Object
```

3.3 Commandline

3.3.1 Overview

Commandline integration is a loosely coupled model of interaction between applications.

Using the underlying operating system calls, the caller application starts another process, handing parameters in string representation via the “commandline”. The standard way of returning a result is an integer value representing the outcome.

This simple model is somewhat getting complicated by the way the Java VM launcher is currently working - there is simply **no** way of calling and awaiting a useful result synchronously. To work around that, there are some standard means of dealing with synchronization files.

Use the commandline if you have an application integration that does not necessarily need sophisticated synchronization. The complexity of the actions to take does not matter here, the commandline is very powerful - as long as you don’t need it to return a value... The possible range of use for the CLI is from ad-hoc calls to open or print a document

to sophisticated commandlines involving multiple options or scripting calls.

The border between “programming interface” and operative use is not very clear, so there is another chapter on CLI in the “Operator’s Guide”. In the “Operator’s Guide” you will find information about

- CLI semantics
- Standard CLI options
- Scripting with the commandline

Here we will go for extending the commandline.

3.3.2 Adding a new option

If you prefer to implement some complex logic in Java, you can add a new CLI option yourself. For doing this, there is an extension point **com.cabaret.claptz.cli.optionprocessors**. Every CLI option processor is declared using the **optionprocessor** element. The implementation must conform to the *de.intarsys.tools.cli.ICLIOptionProcessor* interface.

Example processor implementation

```
package com.cabaret.demo.ext.commandline;

import java.util.logging.Level;
import java.util.logging.Logger;

import de.intarsys.tools.cli.ICLIOptionProcessor;
import de.intarsys.tools.cli.ICLIProcessor;
import de.intarsys.tools.cli.CLIException;
import de.intarsys.tools.cli.ICLIOption;

/**
 * A simple demo for adding a new command line option processor.
 */
public class CLIOPHelloWorld implements ICLIOptionProcessor {

    public void initOption(ICLIProcessor cliProcessor, ICLIOption option)
        throws CLIException {
        // no initialization needed
    }

    public boolean isMainOption() {
        // process immediately
        return true;
    }

    public void processOption(ICLIProcessor cliProcessor, ICLIOption option)
        throws CLIException {
        Logger.getLogger("HelloWorld").log(Level.INFO, "Hello World");
    }
}
```

Example instrument declaration

```
<instrument
  id="com.cabaret.demo.ext.commandline.helloworld"
  name="Demo CLI helloworld">

  <requires>
    <prerequisite instrument="com.cabaret.claptz.common"/>
  </requires>

  <extension point="com.cabaret.claptz.cli.optionprocessors">
    <optionprocessor
      opt="hi"
      longopt="hello"
      class="com.cabaret.demo.ext.commandline.CLIOPHelloWorld"/>
    </extension>
  </extension>
</instrument>
```

You can perform this option on a commandline like this

```
SignLiveCC.exe -hi
```

or

```
SignLiveCC.exe --hello
```

3.4 Shared library

3.4.1 Overview

At the time of this writing, shared library support is tested and released only for windows platforms.

This interface allows access to a Sign Live! CC implementation from plain C or C++ client code. It provides a generic way for integrating Sign Live! CC.

As with the ActiveX implementation, you can embed any action, up to a complete wizard or launch of a document viewer implementation in your application seamlessly.

3.4.2 Client setup and Installation

There is no special installer necessary or available.

At compile time your development environment needs access to the include files and libraries for your platform in the “sdk/SharedLib/dev and “sdk/IJLauncher/dev subdirectories of your installation.

At runtime your code must have access to the shared library “stagesl”. By default it is already installed in the appropriate folders for your platform (on Windows in the folder “<installation_dir>/bin”).

The Sign Live! CC platform itself is launched when accessing the shared library from your code. Launch behavior can be customized in the launch configuration file (“stagesl.dll.lcnf” for Windows). This shared

library indirectly uses the JVM launcher library (“ijlauncher.dll” on Windows), you can change the launch configuration for this one, too. The launch configuration settings should be fine for the use cases in the demos. For more information on launch configuration files see the “Operator’s Guide”.

3.4.3 Demo setup and Installation

Each demo directory contains a prebuilt executable in the “bin” subdirectory and an instrument file with some declarations in the “instruments” subdirectory. Copy the contents of the “instruments” directory to “instruments” in the main application directory. You can also copy the contents of the “bin” directory to the main “bin” or run the demo from the demo directory. In both cases make sure the stagesl library and the ijlauncher library are in your path (or library path on non-Windows).

For demos that require a method name as input you can find the valid names in the instrument files. You can also configure your own methods. See Server implementation below.

The demos also include build files for Visual Studio 2010. Be aware that you will need the 32 bit version of the application to build the demos on 64 bit-Windows, because Visual Studio is itself a 32 bit application and needs information from the registry.

3.4.4 Design

The C/C++ API is designed as a generic gateway to Sign Live! CC as described in Generic API Service Object. This allows for the greatest flexibility in accessing the features we or someone else may have provided with the Sign Live! CC application platform. While a vendor may provide some predefined method sets for your convenience, there is nothing that keeps you from customizing and extending these definitions.

In particular there is no need to recompile another C wrapper if you require further tuning. This will keep your productivity high, as you concentrate on the API content.

The API semantics (its methods behind the generic facade) are defined by attaching methods to the server object of type `com.cabaret.api.GenericAPI:SharedLib`.

3.4.5 API mechanics

3.4.5.1 Types and enumerations

Arguments and return values are wrapped in a generic type

```
typedef struct _ij_value_t ij_value_t;

struct _ij_value_t
{
    short type;
    union
    {
        bool bool_value;
        char char_value;
        wchar_t wchar_value;
        char *string_value;
        wchar_t *wstring_value;
        short int2_value;
        int int4_value;
        __int64 int8_value;
        float float4_value;
        double float8_value;
        jobject object_global_value;
        jobject object_local_value;
    };
};
```

This is the enumeration of currently supported types:

```
enum
{
    TYPE_VOID,
    TYPE_ERROR,
    TYPE_BOOL,
    TYPE_CHAR,
    TYPE_WCHAR,
    TYPE_STRING,
    TYPE_WSTRING,
    TYPE_INT2,
    TYPE_INT4,
    TYPE_INT8,
    TYPE_FLOAT4,
    TYPE_FLOAT8,
    TYPE_OBJECT_GLOBAL,
    TYPE_OBJECT_LOCAL
};
```

3.4.5.2 Marshalling arguments

Marshalling an `ij_value_t` to Java is straightforward

ij_value_t.type	Java type
TYPE_VOID	Ignored on input
TYPE_ERROR	Ignored on input
TYPE_BOOL	Create a <code>java.lang.Boolean</code>
TYPE_CHAR	Create a <code>java.lang.Byte</code>

TYPE_WCHAR	Create a java.lang.Character
TYPE_STRING	Create a java.lang.String using the platform default encoding
TYPE_WSTRING	Create a java.lang.String
TYPE_INT2	Create a java.lang.Short
TYPE_INT4	Create a java.lang.Integer
TYPE_INT8	Create a java.lang.Long
TYPE_FLOAT4	Create a java.lang.Float
TYPE_FLOAT8	Create a java.lang.Double
TYPE_OBJECT_GLOBAL	Forward to Java
TYPE_OBJECT_LOCAL	Forward to Java

If you want to use a native pointer as an argument and want to be platform independent, you should use the greatest integer TYPE_INT8 as type.

3.4.5.3 Marshalling return values

The result is always encapsulated in a `ij_value_t` struct.

This table shows the native types created for the Java result objects.

Java class	ij_value_t.type
java.lang.Byte	TYPE_CHAR
java.lang.Character	TYPE_WCHAR
java.lang.Short	TYPE_INT2
java.lang.Integer	TYPE_INT4
java.lang.Long	TYPE_INT8
java.lang.Float	TYPE_FLOAT4
java.lang.Double	TYPE_FLOAT8
java.lang.Boolean	TYPE_BOOL
java.lang.String	TYPE_WSTRING
<anything else>	TYPE_OBJECT_GLOBAL

3.4.5.4 Object scope

You should be aware that returning a Java object will create a global reference to it, keeping it from being garbage collected. Such references must always be released using `destroyValue`.

3.4.5.5 Error handling

If an exception is encountered, the exception is logged in the platform logging utility (for example the event log on a Windows platform, syslog on a UNIX based platform) and the call returns with an *ij_value_t* type of *TYPE_ERROR*. The value of the struct is undefined.

Currently you have no access to the exception object.

3.4.6 API reference

3.4.6.1 launch

```
ij_value_t  
launch(char* cmdline);
```

Launch the stage platform. If this is an interactive environment, the thread calling *launch* is considered to be the GUI thread. Call *launch* in a dedicated thread if you are launching a blocking environment. Accessing other library methods is not valid before “launch”.

cmdline is a commandline option string as defined in the “Operators Guide”

The call returns an int with the result of the launch. 0 means “no error”. Any negative value is an error code. If the result has type *TYPE_ERROR*, an exception has occurred. Error information is written to the system log.

3.4.6.2 await

```
ij_value_t  
await();
```

This method does not return until the stage platform is available or the launch has failed. No return value is expected. If the result has type *TYPE_ERROR*, an exception has occurred. Error information is written to the system log.

3.4.6.3 shutdown

```
ij_value_t  
shutdown();
```

This method initiates the shutdown of the platform. No return value is expected. If the result has type *TYPE_ERROR*, an exception has occurred. Error information is written to the system log.

3.4.6.4 syncCallArgs

```
ij_value_t  
syncCallArgs(char* name, int argc, ij_value_t *argv);
```

Perform a synchronous call to method *name*

name must be a configured method for the generic service object for this API (*com.cabaret.api.GenericAPI:SharedLib*). *argc* is the number of arguments and *argv* is the pointer to the argument array. Each argument is of type *ij_value_t*.

The call returns a *ij_value_t* with the result of the configured method. If the result has type `TYPE_ERROR`, an exception has occurred. Error information is written to the system log.

3.4.6.5 asyncCallArgs

```
ij_value_t  
asyncCallArgs(char* name, int argc, ij_value_t *argv, void *reserved);
```

Perform an asynchronous call to method *name*

name must be a configured method for the generic service object for this API (*com.cabaret.api.GenericAPI:SharedLib*). *argc* is the number of arguments and *argv* is the pointer to the argument array. Each argument is of type *ij_value_t*. *reserved* is reserved for later use.

The call returns immediately without a result. The computation is performed asynchronously according to the configuration for your shared library and service object declaration. No return value is expected. If the result has type `TYPE_ERROR`, an exception has occurred. Error information is written to the system log.

3.4.6.6 destroyValue

```
void  
destroyValue(ij_value_t value);
```

Destroy the global reference to a previously exported Java object. Java objects that are returned from a generic call are automatically made global to keep the objects valid after releasing the Java environment. The references to these objects must be freed using *destroyValue* to make them available for garbage collection. You can pass any *ij_value_t* to “*destroyValue*”, the method will test if there’s a global reference to be deleted or not.

3.4.7 Server implementation

The server is an instance of `com.cabaret.api.GenericAPI:SharedLib`. The public interface of this service object is defined in the object model repository as described in Generic API Service Object.

Example

```
<extension point= "com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.api.GenericAPI:SharedLib"
    name="hello">
    <perform type="JavaScript" source="'hello' + user)">
      <arg name="user"/>
    </perform>
  </method>
</extension>
```

Registering with the implementor

“`com.cabaret.api.GenericAPI:SharedLib`” will publish the method for use with the shared library.

3.5 ActiveX

3.5.1 Overview

Hint: Please note that this API is not available for all product variants.

While it may seem strange, there is nothing unusual about adding an ActiveX interface to a Java based application.

If you look at the Windows scripting engine controls, you will find exactly the same architecture. Its about encapsulating a “foreign” interpreter, for, let’s say, Visual Basic in a C, C++ or C# based code. You need not mind in which way the control implements the features of interest. Look at it as an adapter between syntactically different languages.

While there are some things you would not want to write in Java just to access them via ActiveX in another application, we talk about rich application semantics required for higher business integration, using synchronous communication. Access is clean, fast and stable.

Sign Live! CC provides an ActiveX implementation that allows you to access its internal document framework. You can embed any action, up to a complete wizard in your application seamlessly.

3.5.2 Client setup and Installation

The necessary ActiveX controls are installed automatically with the installation of the product.

At compile time you only need to reference the correct ActiveX progid.

At runtime, ActiveX controls are resolved automatically.

The Sign Live! CC platform itself is launched when accessing the ActiveX control from your code. Launch behavior can be customized in the launch configuration file ("signliveax.dll.lcnf"). The controls indirectly use the JVM launcher library ("ijlauncher.dll"), you can change the launch configuration for this one, too. The launch configuration settings should be fine for the use cases in the demos. For more information on launch configuration files see the "Operators Guide".

3.5.3 Demo setup and Installation

Your installation comes along with a complete example, including a .NET demo container, to help you implement your solution. To run the demo container you will need .NET-Framework 4 or later.

Each demo directory contains a prebuilt executable in the "bin" subdirectory and an instrument file with some declarations in the "instruments" subdirectory. Copy the contents of the "instruments" directory to "instruments" in the main application directory. You can also copy the contents of the "bin" directory to the main "bin" or run the demo from the demo directory. The ActiveX infrastructure will determine the location of the ActiveX library via the Windows registry.

For demos that require a method name as input you can find the valid names in the instrument files. You can also configure your own methods. See Server implementation below.

The demos also include build files for Visual Studio 2010. Be aware that you will need the 32 bit version of the application to build the demos on 64 bit-Windows, because Visual Studio is itself a 32 bit application and needs information from the registry.

3.5.4 Design

The ActiveX API is designed as a generic gateway to Sign Live! CC as described in Generic API Service Object. This allows for the greatest flexibility in accessing the features we or someone else have provided with the Sign Live! CCapplication platform. While a vendor may provide some predefined method sets for your convenience, there is nothing to keep you from customizing and extending these.

In particular there is no need to recompile another C wrapper or deploy another type library if you require further tuning. This will keep your productivity high, as you concentrate on the API content.

The API semantics (its methods behind the generic facade) are defined by attaching methods to the server object of type `com.cabaret.api.GenericAPI:ActiveX.`

3.5.5 API mechanics

3.5.5.1 Overview

The ActiveX implementation supports both headless and interactive mode. This means you can instantiate an ActiveX document without a parent window.

The Java ActiveX implementation writes to the regular application logs. Additional messages created by the wrapper library will be written to the Windows event log.

3.5.5.2 API ProgId

The ProgId for the application control is `SignLive.ActiveApplication`. The application control implements the `IIJActiveApplication` interface (see below).

The ProgId for the active document is `SignLive.ActiveDocument`. The active document implements the `IIJActiveDocument` interface and the interfaces required for an active document control.

3.5.6 API reference

Here you will find the interface reference. If there is nothing special about an interface or a associated method, you will find no further information here - it just works as defined.

3.5.6.1 IIJActiveCall

Inherits from `IDispatch`.

3.5.6.2 AsyncCall

```
[id(0x60020000)]
HRESULT AsyncCall(
    [in] BSTR Name,
    [out, optional, defaultvalue(0)] long* Id);
```

Convenient method to access `AsyncCallArgs` without arguments.

3.5.6.3 AsyncCallArg

```
[id(0x60020001)]
HRESULT AsyncCallArg(
    [in] BSTR Name,
    [in] VARIANT Arg0,
    [out, optional, defaultvalue(0)] long* Id);
```

Convenient method to access `AsyncCallArgs` with a single argument.

3.5.6.4 AsyncCallArgArg

```
[id(0x60020002)]
HRESULT AsyncCallArgArg(
    [in] BSTR Name,
    [in] VARIANT Arg0,
    [in] VARIANT Arg1,
    [out, optional, defaultvalue(0)] long* Id);
```

Convenient method to access AsyncCallArgs with two arguments.

3.5.6.5 AsyncCallArgArgArg

```
[id(0x60020003)]
HRESULT AsyncCallArgArgArg(
    [in] BSTR Name,
    [in] VARIANT Arg0,
    [in] VARIANT Arg1,
    [in] VARIANT Arg2,
    [out, optional, defaultvalue(0)] long* Id);
```

Convenient method to access AsyncCallArgs with three arguments.

3.5.6.6 AsyncCallArgs

```
[id(0x60020004)]
HRESULT AsyncCallArgs(
    [in] BSTR Name,
    [in] VARIANT Args,
    [out, optional, defaultvalue(0)] long* Id);
```

Asynchronous call to an ActiveX method.

Name is the name of the method on the server document. This one is customized as **name** in the “instrument.xml” declaration file. ”

Args is an array of “VARIANT” types. Supported VARIANT types are VT_I2, VT_I4, VT_R4, VT_R8, VT_BOOL and VT_BSTR.

Id unused

3.5.6.7 GetLastErrorMessage

```
[id(0x60020005)]
HRESULT GetLastErrorMessage(
    [out, retval] BSTR* Message);
```

Can be called after having received an error code. Provides an error message if available.

3.5.6.8 GetLastErrorTrace

```
[id(0x60020006)]
HRESULT GetLastErrorTrace(
    [out, retval] BSTR* Trace);
```

Can be called after having received an error code. Provides a java stack trace if available.

3.5.6.9 SyncCall

```
[id(0x60020007)]
HRESULT SyncCall(
    [in] BSTR Name,
    [out, retval, optional, defaultvalue(0)] VARIANT*
    ReturnValue);
```

Convenient method to access SyncCallArgs without arguments.

3.5.6.10 SyncCallArg

```
[id(0x60020008)]
HRESULT SyncCallArg(
    [in] BSTR Name,
    [in] VARIANT Arg0,
    [out, retval, optional, defaultvalue(0)] VARIANT*
    ReturnValue);
```

Convenient method to access SyncCallArgs with a single argument.

3.5.6.11 SyncCallArgArg

```
[id(0x60020009)]
HRESULT SyncCallArgArg(
    [in] BSTR Name,
    [in] VARIANT Arg0,
    [in] VARIANT Arg1,
    [out, retval, optional, defaultvalue(0)] VARIANT*
    ReturnValue);
```

Convenient method to access SyncCallArgs with two arguments.

3.5.6.12 SyncCallArgArgArg

```
[id(0x60020000a)]
HRESULT SyncCallArgArgArg(
    [in] BSTR Name,
    [in] VARIANT Arg0,
    [in] VARIANT Arg1,
    [in] VARIANT Arg2,
    [out, retval, optional, defaultvalue(0)] VARIANT*
    ReturnValue);
```

Convenient method to access SyncCallArgs with three arguments.

3.5.6.13 SyncCallArgs

```
[id(0x60020000b)]
HRESULT SyncCallArgs(
    [in] BSTR Name,
    [in] VARIANT Args,
    [out, retval, optional, defaultvalue(0)] VARIANT*
    ReturnValue);
```

Synchronous call to an ActiveX method.

Name is the name of the method on the server document. This one is customized as **name** in the “instrument.xml” declaration file. ”

Args is an array of “VARIANT” types. Supported VARIANT types are VT_I2, VT_I4, VT_R4, VT_R8, VT_BOOL and VT_BSTR.

ReturnValue is the object returned by the ActiveX server method implementation.

3.5.6.14 IIJActiveApplication

Inherits from IIJActiveCall.

3.5.6.15 IIJActiveDocument

Inherits from IIJActiveCall.

LoadFile

```
[id(0x60030000)]
HRESULT LoadFile(
    [in] BSTR FileName,
    [in] long Mode);
```

Convenient method for container implementations that only have access to the default interface. The implementation is the same as IPersistFile::Load.

3.5.6.16 IDispatch

ATL default implementation

- 3.5.6.17 IPersistStreamInit
ATL default implementation
- 3.5.6.18 IOleControl
ATL default implementation
- 3.5.6.19 IOleDocument
 - EnumViews
Only a single view is supported.
- 3.5.6.20 IOleDocumentView
 - **ApplyViewState**
E_NOTIMPL, not applicable
 - **Clone**
E_NOTIMPL, not applicable
 - **Open**
E_NOTIMPL, not applicable
 - **SaveViewState**
E_NOTIMPL, not applicable
 - SetRectComplex
E_NOTIMPL, not applicable
- 3.5.6.21 IOleObject
ATL default implementation.
 - DoVerb
Verbs
 - OLEIVERB_SHOW,
 - OLEIVERB_OPEN,
 - OLEIVERB_UIACTIVATE
- 3.5.6.22 IOleInPlaceActiveObject
ATL default implementation.
- 3.5.6.23 IViewObjectEx
ATL default implementation.
- 3.5.6.24 IOleInPlaceObjectWindowless
ATL default implementation.
- 3.5.6.25 IPersistFile
 - IsDirty
S_FALSE, no reply
 - Save
E_FAIL, not implemented
 - GetCurFile
E_FAIL, not implemented

- SaveCompleted
S_OK, not implemented

3.5.6.26 IPersistStorage
ATL default implementation.

You should not call this interface's methods in this version.

3.5.6.27 IDataObject
ATL default implementation.

3.5.7 Server implementation

The server object publishes the methods of `com.cabaret.activedoc.CommonActiveDoc`. The public interface of this object is defined in the object model repository as described in [APIs](#).

Example

```
<?xml version="1.0" ?>

<instrument
  id="com.foo.ActiveX"
  name="ActiveX Support">

  <requires>
    <prerequisite

instrument="com.cabaret.application.pdf.PDFApplicationInstrument"/>
    <prerequisite
      instrument="com.cabaret.scripting.javascript.engine"/>
    </requires>

    <extension point= "com.cabaret.objectmodel.members">
      <method
        implementor="com.cabaret.activedoc.CommonActiveDoc"
        name="sayHello">
        <perform type="JavaScript" source="app.alert('hello,
document') "/>
      </method>
      <method
        implementor="com.cabaret.activedoc.pdf.PDFDoc"
        name="sayHello">
        <perform type="JavaScript" source="app.alert('hello, PDF
document') "/>
      </method>
    </extension>
  </instrument>
```

Registering with the implementor

“`com.cabaret.activedoc.CommonActiveDoc`” will publish the method for all implementation subclasses, but you can use a subclass to be more specific about the document type you want to use at any time.

3.5.8 Problem tracking

If you encounter problems using the ActiveX controls, you may try some of the following tips.

3.5.8.1 32/64 Bit Platforms

Be sure all components use the correct binary, either 32 or 64 bit. If for example you have installed the 32 bit version of our application, you will need a 32 bit ActiveX client and a 32 bit Java VM.

Be especially careful when implementing .NET clients. By default the compiler will create hybrid executables that will either run on 32 and 64 bit machines. The drawback is that on a 64 bit machine with a 32 bit installation of a Sign Live! CC application, the .NET client still executes in 64 bit mode and will later fail in loading the ActiveX control. You must force this client in 32 bit mode at compile time or using the “CoreFlags” tool from Microsoft.

3.5.8.2 Installation

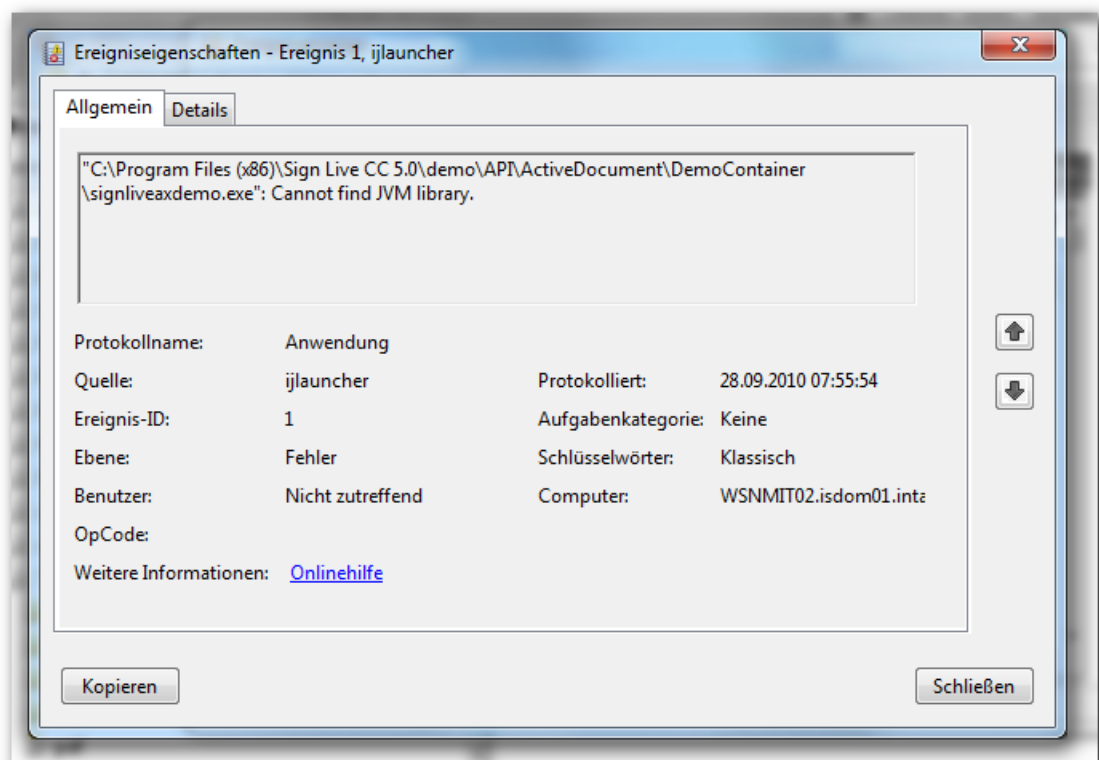
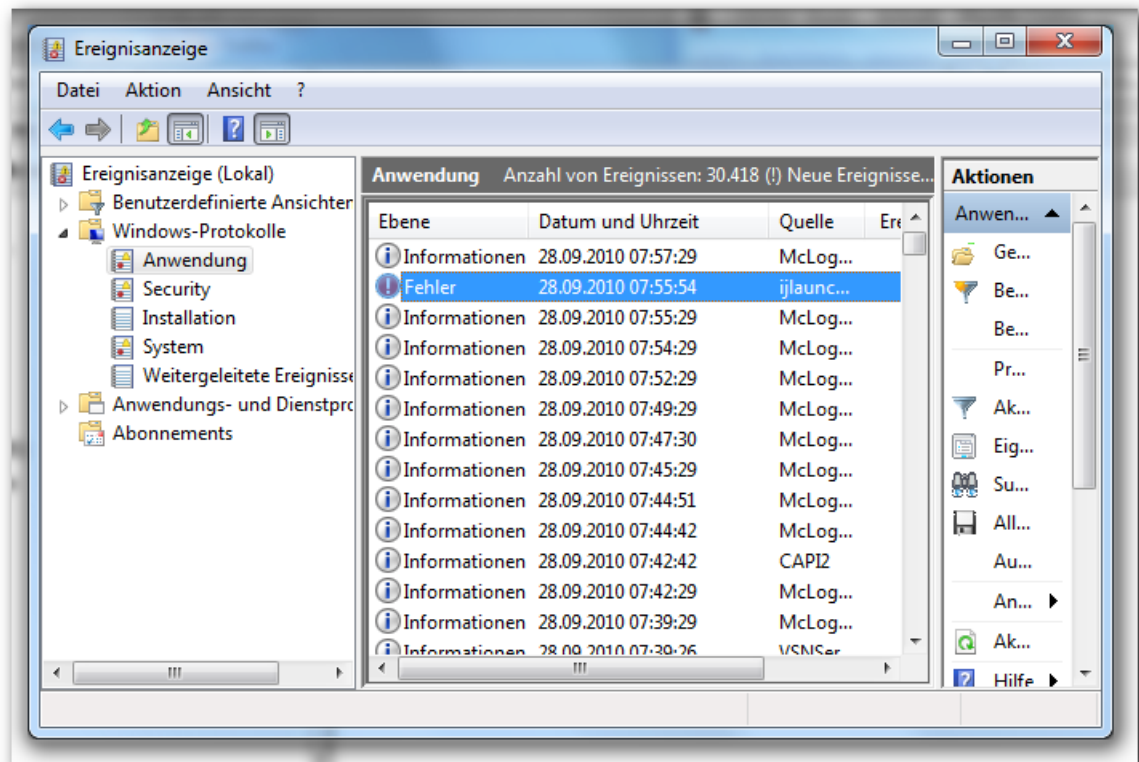
The ActiveX controls are implemented in **signliveax.dll**. These libraries are registered with the installation program only. Try to rerun the installation program or manually check the installation of the controls.

The ActiveX control tries to start the Java VM. This functionality is implemented in **jvmlauncher.dll**. This library is searched in the same directory as the ActiveX library.

Upon startup, a Java VM installation is searched as defined in the **jvmlauncher.dll.lcnf**. By default, this is first a JRE installed locally with the application in a “jre” subdirectory. Then the registry is searched for a valid JRE installation.

3.5.8.3 Windows Event Log

All errors that occur in the native part of the ActiveX implementation can't be logged to the standard logging files. You can find these error events in the standard event log of windows.



3.5.8.4 Standard log file

If the Java VM has started, the application code will log to the standard places. Look at the log file to check if you have correctly set up your object model and scripts.

3.5.8.5 Standard Log Configuration

To change the standard logging behavior, you should add a Java **logging.properties** configuration when starting the virtual machine. This is a simple text file with configuration details for the logging subsystem

Example **ijlogging.properties**

```
.level=FINEST
handlers=java.util.logging.ConsoleHandler
java.util.logging.FileHandler
java.util.logging.ConsoleHandler.level=WARNING
java.util.logging.ConsoleHandler.formatter=de.intarsys.tools.logging.SimpleFormatter
java.util.logging.FileHandler.level=FINEST
java.util.logging.FileHandler.formatter=de.intarsys.tools.logging.SimpleFormatter
java.util.logging.FileHandler.limit=100000
java.util.logging.FileHandler.count=10
java.util.logging.FileHandler.pattern=%h/ij.vmid_%u.index_%g.log
```

This file is handed to the VM via a VM option parameter. You can configure this parameter in the **ijlauncher.dll.lcnf** file.

Example

```
...
<vmoption value="-
Djava.util.logging.config.file=ijlogging.properties"/>
...
```

Add this element in the **init** element of the launch configuration file. Either use a full qualified file name for the properties file or be sure to copy it to the working directory of the ActiveX host application.

3.5.8.6 Demo Container

For a quick check of your setup and your object model declarations, you can use the generic client application installed with the application.

3.5.9 Integration tips

The usage of ActiveX may be limited or need special handling in some environments. Here are some hints for special environments if you use ActiveX. This are not limits of our implementation, but is idiomatic for the respective environments.

3.5.9.1 Microsoft .NET

The management of ActiveX Controls in .NET differs from “normal” ActiveX behavior. The “COM Controls” are unmanaged and have to be explicitly released. If you use the ActiveX Control as an invisible Control, please release the Control after usage with *Marshal.ReleaseComObject(activexdoc)*.

E.g.:

```
protected override void Dispose(bool disposing)
{
    if (disposing && (components != null))
    {
        components.Dispose();
    }
    Marshal.ReleaseComObject(activatedoc);
    base.Dispose(disposing);
}
```

3.5.9.1.1 SAP ABAP

The usage of the command “NOFLUSH” in ABAP Code controlling the COM Object can lead to a deferred release of the ActiveX Control and its resources. Until the ActiveX Control is released by the SAP System, the files used by the ActiveX Control are locked and this can’t be accessed or deleted.

Please use the FLUSH command for flushing the Automation Queue after the Control is used.

For more Information about COM Objects inside ABAP use the SAP documentation “SAP Control Framework”:

German:

<http://help.sap.com/printdocu/core/Print46c/de/data/pdf/BCCIGOF/BCCIGOF.pdf>

English:

<http://help.sap.com/printdocu/core/Print46c/en/data/pdf/BCCIGOF/BCCIGOF.pdf>

3.6 Service Framework

3.6.1 Overview

A wide range of API integration is provided by the service framework. This is more related to configuration and mapping, so this topic is completely handled in the “Operators Guide”.

4. Embedding

4.1 Overview

We dedicated a chapter of its own to “Embedding Sign Live! CC” as this is not comparable to all the other APIs. With embedding, you can access any feature and change any behavior of Sign Live! CC. From programatic control from your Java client to seamless integration of existing viewer components you can do anything.

The price is the added complexity of the available APIs.

4.2 Interfacing Stage

First of all, you must acquire a handle to a *com.cabaret.api.launcher.StageLauncher*, the wrapper for handling the lifecycle of the application instance.

```
StageLauncher launcher = StageLauncher.get();
```

4.3 File Environment

4.3.1 The file environment

Stage acts in a “file environment” defined by four directories

- baseDir
- profileDir
- workingDir
- tempDir

Each of the directories has a predefined default and can be set using either accessor methods, the commandline or the config file.

4.3.2 The base directory

The base directory is used in many places in the platform to address relative file resources, for example the instruments or the config directory. You should be aware of such dependencies if you change the base directory.

By default the base directory is the directory where stage is installed, to be more exact the directory above the “bin” folder where the launcher exe is located.

You can define the base directory, in order of precedence:

- Using set setter method

```
launcher.setBaseDir(new File("c:/temp/mybase"));
```

- Setting the basedir attribute in the config file. If you do this, you must provide an absolute filename for the location of the config file. Otherwise the base directory is evaluated lazy before the config file can be read!

```
<stage basedir="c:/temp/mybase" >  
...  
</stage>
```

4.3.3 The profile directory

The profile directory is used to store information related to the actual stage invocation, such as user preferences, log files or other individual settings. Some implementations that like to save information under the base directory (such as licenses) may opt to save in the profile directory as a fallback.

By default the profile directory is the users home directory with the trailing “.<application name>_<major>.<minor>” directory. For example, under Windows you will find a typical profile directory at

```
C:\Dokumente und Einstellungen\foo\.SignLiveCC_7.1
```

Under Unix and Mac the leading “.” will hide this directory from normal inspection - remember this if you try to look up the log files.

You can define the profile directory, in order of precedence:

- Using set setter method

```
launcher.setProfileDir(new File("c:/temp/myprofile"));
```

- Setting the profile commandline option

```
launcher.addOption("profile", "c:/temp/myprofile");
```

- Setting the profiledir attribute in the config file.

```
<stage profiledir="c:/temp/myprofile" >
...
</stage>
```

If the value you set is not absolute, it is interpreted relative to the Java VM's "user.home".

4.3.4 The working directory

The working directory currently references the same directory as the base directory.

4.3.5 The temp directory

The temp directory is used in the platform if a location for anonymous temporary files is needed.

By default this is the temporary directory defined by the Java VM.

You can define the temp directory, in order of precedence:

- Using set setter method

```
launcher.setTempDir(new File("c:/temp/mytemp"));
```

4.3.6 The file environment singleton

By default, stage sets itself as the `de.intarsys.tools.environment.file.FileEnvironment` shared singleton in its initialize method when it is started.

4.4 Launching

4.4.1 Launching stage

Launching stage is normally done from the commandline with "java.exe" or using the included launcher executable. The code behind the application startup is roughly

```
StageLauncher launcher = StageLauncher.get();
launcher.launch();
try {
    launcher.await();
} catch (Exception e) {
    throw new IllegalStateException(e);
}
```

The kind of application started is in the arguments and the referenced "instrument" configuration. The art of embedding Stage is to know the desired target environment and provide the appropriate arguments. To

learn more about the available commandline and configuration options, see the Sign Live! CC Operator's Guide.

The launch process is described in more detail at [The big picture](#) and [A simple Main](#). A short summary: bootstrapping Stage means loading all extensions. The interesting extension here is `com.cabaret.claptz.main`, defining a class of type `com.cabaret.claptz.stage.main.IMain`. The “main” declaration supports the definition of an “id” and a “default” flag.

Example:

```
<extension point="com.cabaret.claptz.main">
  <main
    id="interactive"
    default="true"

class="com.cabaret.application.interactive.SWTStandaloneApplication"/>
</extension>
```

After loading all extensions, Sign Live! CC looks up the main implementation to be started in the following order:

- Lookup and start the main with the “id” matching the argument value of the “-launch” option.

Example: launch the “interactive” main

```
... -launch interactive
```

- If no match or no “-launch” option, launch the single main tagged with “default”
- If none of the above and there is a single main definition start this one.

The standard launching procedure assumes that the main will not return from “launch” as long as the application is running, the shutdown cycle follows immediately after the launch.

4.4.2 Using an interactive embedding

4.4.2.1 Overview

The interactive version of Sign Live! CC in the standard deployment is launched with the

```
-launch interactive
```

commandline option, e.g.

```
launcher = StageLauncher.get();
launcher.setBlocking(true);
launcher.selectLaunchInteractive();
launcher.launch();
```

will do the job. This will launch stage, keep it running until the user closes the application window and then returns from launch.

To run you own event loop use

```
-launch interactive noblock
```

or

```
launcher = StageLauncher.get();
launcher.setBlocking(false);
launcher.selectLaunchInteractive();
launcher.launch();
```

instructing the application to return immediately after the launch.

4.4.2.2 Customized embedding

To customize the embedding better to your individual needs, you must make some decisions:

- **Switch between blocking and no-blocking behavior**
This decision you have already seen in the preceding chapter, but it applies also to the more general embedding implementations.

Under blocking we understand that a call to launch will not return until the users finishes its use of the application. This is a kind of a standard application launcher with hit and run semantics.

noblocking means that the launch will return after it has created the application. Somebody else is now responsible to drive the GUI event loop.

This behavior is controlled using the block (default) or noblock launch option values. These exact semantics of this options depend on the installed `de.intarsys.swt.context.IDisplayProvider` (see below), as block means simply that `waitTerminate` is called for the display provider.

- **Drive the event loop**
Depending on your above decisions, you have to implement and drive the event dispatch loop. The event dispatch loop must run in the GUI thread, the thread that created the display and all the controls. If you choose *block* behavior, the application will run an event loop on its own.

4.4.2.3 Preconfigured option values

Instead of providing option values to your launch arguments, you can configure them in the respective extension points.

Option values that are also available for configuration are:

- block
- wakeonstart
- wakeonview

Example configuration

```
<extension point="com.cabaret.claptz.main">
  <main
    id="myembedded"
    default="true"
    block="false"
    wakeonstart="false"

class="com.cabaret.application.interactive.SWTDefaultApplication"/>
</extension>
```

4.4.2.4 And even more...

To achieve an even more deeper integration, you can drop the standard application implementation and provide an 'com.cabaret.claptz.stage.main.IMain' of your own.

Now you start with a look& feel of your choice, watch the processor lifecycle and decorate them with your desired window behavior. This kind of integration is briefly described in an upcoming chapter.

4.5 Embedding a servlet container

4.5.1 Running in a servlet container

In some scenarios it may be necessary to provide stage hosted features in a server environment. You can think of server based signing or validation for example.

While “low bandwidth” installations are supported with the built in ULS platform, we support integration into J2EE scenarios for improved scalability, stability and security. This chapter is about creating and deploying services into a servlet container. We use standard J2EE features to do this, while all examples are based on a Tomcat installation.

Remember that you need a professional license for this mode. The missing license is one of the most common reasons for startup failure. Check your log file. You may simply have missed installing the license or copied it to the wrong directory.

4.5.2 Deployment decisions

When embedding stage in servlet environment you can choose from the following installation options:

- Tweak the servlet container classpath

You can add the Stage libraries directly to the servlet container classpath, using a servlet container proprietary process. This will work as a quick hack or in small scale solutions. Expect conflicts if more than one web application needs Stage integration.

- Include the stage distribution within the web application.

This requires intimate knowledge of the internal structure of stage and you will need additional installations for every web application to be deployed. More over, there may be environments that must act on an existing, secure installation, relying on the self-check features of the stage platform.

So, while you can create such a deployment if needed we do not provide support here.

- Reference an existing installation

Your web application uses a standard installation of stage to launch. You can reuse the installation in different web applications and create a modular deployment.

This is the option we recommend and will describe here further.

4.5.3 Reference an existing installation

If you want to access stage from within your web application, it must be available to your web application classloader.

If you deploy in a “standard” servlet container, your web application must be self contained. For referencing an external resources most servlet containers, for example Tomcat and Jetty, support tweaking the web application classloader. **This will not work with security constraints set up on the servlet container environment.**

We deploy a solution that is developed and tested for the Tomcat environment. This may work for Jetty and others - as the source code is provided you can adapt it to your needs.

To support this embedding scenario, Sign Live! CC provides a “cabapiimpl.jar”, situated in the “lib” directory of the Sign Live! CC installation. You must add this file to your web application libraries to implement an integration as described in the following chapters.

The solution is implemented in *com.cabaret.api.embedded.servlet.StageInitializer* as a *javax.servlet.ServletContextListener* that initially tweak its web application context as needed.

The listener will manage the classpath mechanics and the Sign Live! CC lifecycle for you. To achieve this, you must at least setup a link to the Sign Live! CC installation. There are some different ways to do this:

- Setup an environment variable Sign Live! CC_HOME, pointing to your Sign Live! CC installation directory
- Define a Java system property Sign Live! CC_HOME, again pointing to your Sign Live! CC installation directory
- Install Sign Live! CC in the “SignLiveCC” subdirectory of the web application.
- Enter the path to the Sign Live! CC installation manually in the web.xml. Change the “de.intarsys.claptz.basedir” context parameter to the root of your installation.

Example

```
<listener>
  <listener-
class>com.cabaret.api.embedded.servlet.StageInitializer</listener-
class>
</listener>
<context-param>
  <param-name>de.intarsys.claptz.basedir</param-name>
  <param-value>C:\NoInstall\Sign Live! CC 7.1</param-value>
</context-param>
```

There are some more context parameters to the *com.cabaret.api.embedded.servlet.StageInitializer* that may come handy to your special installation. You will find the definition in a chapter below.

An example for setting up a complete web application environment you will find in the “snippets” that come along with the distribution.

4.5.4 Accessing Sign Live! CC

When bootstrapping an embedded stage, in addition to the standard instruments of the installation, the instruments located in the “instruments” subdirectory of your web application are always loaded.

4.5.5 Additional servlet parameters

This initializer supports the following servlet context initialization parameters:

- `de.intarsys.claptz.basedir`
The path to the Sign Live! CC installation. All other paths are interpreted with this parent if they are relative.
- `de.intarsys.claptz.classpath`
A “;” separated list of “jar” files and directories containing “*.classes” files. This is the classical Java classpath semantics. The final classpath is the addition of resources found here and the resources found with “classpath-scan”. The default is empty.
- `de.intarsys.claptz.classpath-scan`
A “;” separated list of directories containing “*.jar” files. This makes it easy to include a complete installation with its libraries. The final classpath is the addition of resources found here and the resources found with “classpath”. The default is “lib”.
- `de.intarsys.claptz.classpath-scan-exclude`
A “;” separated list of file names excluded from the list of the above “jar” files. This may be for example necessary to exclude the servlet API or some other implementations that are available in the referenced installation but should not be used in the servlet container runtime. The default is “servlet_api.jar”.
- `de.intarsys.claptz.stage-basedir`
The base directory for the stage environment. This defaults to “de.intarsys.claptz.basedir” defined above.
- `de.intarsys.claptz.stage-workdir`
The directory for the stage working directory. This is currently unused.
- `de.intarsys.claptz.stage-profiledir`
The directory for the stage profile information. If not defined, the stage default is used.
- `de.intarsys.claptz.stage-tempdir`
The directory for the stage temp directory. If not defined, the stage default is used.

- `de.intarsys.claptz.stage-config`
The optional stage config file.
- `de.intarsys.claptz.stage-define`
Optional “-define” flags for preprocessing purposes. The default is empty. Most of the time you may want to define “headless” here! This will keep most of the interactive parts of a standard installation out of the way.
- `de.intarsys.claptz.stage-instruments`
A “;” separated list of directories to search for instrument definitions. The default is “instruments”.

Example:

```
<listener>
  <listener-
class>com.cabaret.api.embedded.servlet.StageInitializer</listener-
class>
</listener>

<context-param>
  <param-name>de.intarsys.claptz.basedir</param-name>
  <param-value>C:/Programme/Sign Live! CC 7.1</param-value>
</context-param>

<context-param>
  <param-name>de.intarsys.claptz.classpath</param-name>
  <param-value>foo;bar</param-value>
</context-param>

<context-param>
  <param-name>de.intarsys.claptz.classpath-scan</param-name>
  <param-value>lib;extensions</param-value>
</context-param>

<context-param>
  <param-name>de.intarsys.claptz.classpath-scan-exclude</param-name>
  <param-value>servlet-api.jar</param-value>
</context-param>
```

4.5.6 The configuration file

The main task of the configuration in stage is the definition of the set of instruments to be loaded. Configuration file use and syntax is described in detail Sign Live! CC Operator’s Guide chapter „Configuration“.

To define the configuration file in a servlet container you can use the following servlet context parameters:

- `de.intarsys.claptz.stage-config`

If this is not an absolute path, it is interpreted relative to the web application root directory.

```
<context-param>  
  <param-name>stage-config</param-name>  
  <param-value>config/my.config</param-value>  
</context-param>
```

This will load the configuration file “config/my.config” defined within the web application deployment directory.

4.6 Controlling

4.6.1 Controlling Sign Live! CC

Now that you have running instance of Sign Live! CC - what do you do with it?

The main building blocks and APIs are presented in “Part I”, for example processors, wizards and documents. These are generic APIs that model business functions that can be plugged in in your workflow. The API is quite simple, but now the focus shifts to what business functions are provided for me and what are their arguments.

Many examples of how to work with these business objects are presented in the Sign Live! CC Scripting Tutorial. There should be no problem in migrating the code snippets to plain Java.

Here we will present the basic processors and wizards available in Sign Live! CC and the mechanics of using them.

More details on the business functions that you are looking for you can find in the specialized developer guides, for example on the topic of security applications.