as2lib - a little introduction

Christoph Atteneder, Martin Heidegger,
Michael Herrmann, Alexander Schliebner and Simon Wacker.
Special thanks to Vera Fleischer[1] and Jayaprakash A.[2] for support in translation.

3. September 2004

[1] www.mediasparkles.com
[2] www.as2max.com

Contents

Chapter 1

as2lib - Founding History

The *as2lib*, an *Open Source ActionScript 2* Library, was founded in September 2003 as a project to create better programming possibilities for *ActionScript 2*. The project team deals with core problems of Flash and tries to solve daily problems while programming with Flash. One of the most important characteristics is that *as2lib* is published under the *MPL* (Mozilla Public License). That means free usage in private as well as commercial projects. After the definition of programming guidelines and the basic concept, the work on the separate packages was set about by five project members.

Name	*Duty*	*Website*	*Nationality*
Atteneder Christoph	Developer	www.cubeworx.com	Austria
Heidegger Martin	Project Manager	www.traumwandler.com	Austria
Herrmann Michael	Developer		Austria
Schliebner Alexander	Co-Initiator	www.schliebner.de	Germany
Wacker Simon	Chief Developer	www.simonwacker.com	Germany

Table 1.1: Active Members *as2lib*

Figure 1.1: www.as2lib.org

Chapter 2

Core Package

Motivation: The definition and provision of specific functionalities in all classes simplify the development and failure correction during development.

Solution: All classes, interfaces and packages of as2lib follow certain guidelines. The most important core class resides in the ***core*** *package*[1].

2.1 BasicInterface

For a better definition of class functionalities *interfaces* are used intensively within *as2lib*. Each interface created in *as2lib* extends the *BasicInterface* to ensure the following functionality in every class:

- **getClass**():*ClassInfo* - This method offers exact information of the class in which this function is called. The returned information is of type *ClassInfo* and contains their class names, information like methods, properties, class paths and super classes.
- **toString**():*String* - This method returns a String representation of the class.

The logic of the *getClass* method is provided by the *BasicClass* class (see fig. 2.1, S. 4).

2.2 BasicClass

The base class of *as2lib* is the *BasicClass*. All classes of *as2lib* are directly or indirectly derived from the *BasicClass* class. It implements the *BasicIn-*

[1] org.as2lib.core.*

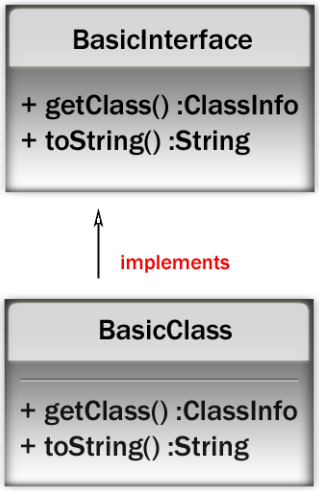

Figure 2.1: Main part of the *core* package

terface and provides through the *ReflectUtil*[2] and the *ObjectUtil*[3] classes the logic for the following methods:

- **getClass()**:*ClassInfo* - Explanation see BasicInterface. The creation of class information is made possible by the *as2lib reflection* package, see chapter 6.
- **toString()**:*String* - This method returns a class representation of type String.

[2] org.as2lib.env.util.ReflectUtil
[3] org.as2lib.util.ObjectUtil

Chapter 3

Output Handling

Motivation: The normal output of applications in Flash is done via the native operation

```
trace(expression);
```

The trace output is nevertheless only visible inside the development environment that supports the operation. In all other cases (e.g.: in a web application) no standard output is defined. A library should provide a standardized output for users as well as for the developer that can be used everywhere.

Solution: To have multiple output possibilities in every runtime environment[1], the *Out*[2] class is used. Given this class, it is possible to e.g.: save error messages on the server side, to ensure that failures are not only perceptible to the client but also to the developer. The *Out* class deals with all incoming requests and forwards them depending on the configuration to one or more *OutputHandler*[3]. *As2lib* offers a standardized output possibility for unlimited interfaces.

Usage: A simple use case of the *as2lib* Output Handling is visualized in figure 3.1 on page 6. After an instance of the *Out* class is created and the provided TraceHandler[4] added, an output can take place. The sequence of the single actions matches the order in which they are numbered.

The *Out* class can as in figure 3.1 on page 6 access a pre-defined output handler like the *TraceHandler* or a custom implementation of the *OutHandler* (e.g.: an output in the *Macromedia* Alert Component)[5]:

[1] Flash can run in a Browser, Flash Player, in *Macromedia Central*, in a compiled application(*.exe),...

[2] org.as2lib.env.out.Out

[3] org.as2lib.env.out.OutputHandler, org.as2lib.env.out.handler.*

[4] org.as2lib.env.out.handler.TraceHandler

[5] The Macromedia Alert Component must be in the library and you must own Flash MX Professional 2004, to access it via the Alert class.

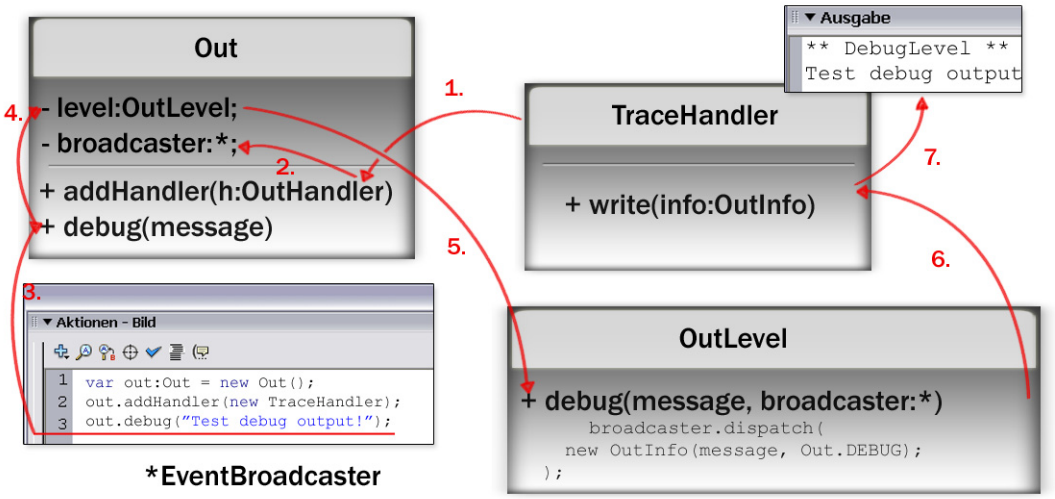

Figure 3.1: Use case of the *as2lib* Output Handling

```
import org.as2lib.env.event.EventInfo;
import org.as2lib.env.out.OutHandler;
import org.as2lib.env.out.OutInfo;
import org.as2lib.env.out.OutConfig;
import org.as2lib.core.BasicClass;
import mx.controls.Alert;

class test.org.as2lib.env.out.handler.UIAlertHandler
  extends BasicClass implements OutHandler {

  public function write(info:OutInfo):Void {
    Alert.show(info.getMessage(),
      getClass().getName());
  }

}
```

The definition of the output levels (e.g.: *out.setLevel(Out.DEBUG)*) makes it possible to prohibit the output of certain information. The possible output levels are:

- Out.ALL
- Out.DEBUG
- Out.INFO

- Out.WARNING
- Out.ERROR
- Out.FATAL
- Out.NONE

These levels allow developers to "switch on" display of debug messages during development or maintenance, these messages can then be easily "switched off" or "filter" these messages before publishing the final application. If during development, you wish to view all types of messages above *DEBUG*, you would set the level to *Out.DEBUG*, This way all messages of type *DEBUG*, *INFO*, *WARNING*, *ERROR* and *FATAL* are displayed, while all *LOG* messages are supressed.

```
var out:Out = new Out();
out.addHandler(new TraceHandler());

out.setLevel(Out.DEBUG);

out.log("log me Please!");
out.debug("debug me Please!");
out.info("inform me Please!");
out.warning("warn me Please!");
out.error(new Exception("Output Error", this,
  arguments));
out.fatal(new FatalException("Fatal Output Error",
  this, arguments));
```

Should you choose to display on fatal failures in your finished application, this can be accomplished by changing a single line.

```
out.setLevel(Out.FATAL);
```

Chapter 4

Exception Handling

Motivation: Uncaught failure messages are put out with[1]:

```
trace(Error.toString());
```

Besides the fact that the printed out information is only little or not at all informative (Only "Error" or the passed String to the constructor is printed out. See fig. 4.1, p. 8) the display of the failure message is only possible in *Flash MX 2004*.

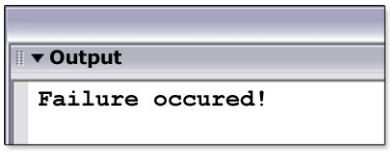

Figure 4.1: Output of an Error (*throw new Error("Failure occured!");*) in Flash MX 2004.

Solution: *As2lib* contains classes based on the *Macromedia* native *Error* class, which implements the methods of the *Throwable*[2] interface. The new functionalities of Exception Handling are:

- All operations that are called before the exception is thrown are saved in a Stack to help easily identify the points of failure. With the help of *Reflections*, see chapter 6, the name of the failure message is automatically generated.
- Exceptions can be easily wrapped into other exceptions.

[1] Online documentation on livedocs.macromedia.com/flash/mx2004/main/12_as217.htm
[2] org.as2lib.env.except.Throwable

The following predefined exception classes are provided:

- *AbstractException*: All exception classes are derived from the AbstractException class. It implements the methods that the Throwable interface defines (see fig. 4.2, p. 10).
- *Throwable*: Is an interface that enforces the implementation of the following methods:
  - **getStackTrace(Void):***Stack* - Returns all operations that have been called before the exception is thrown.
  - **initCause(cause:Throwable):***Throwable* - Declares the reason for the exception and can only be set once. This method is normally used in order not to lose information when an exception is the cause of another exception.
  - **getCause(Void):** *Throwable* - Returns the cause of the exception.
  - **getMessage(Void):***String* - Returns the message that was passed to the constructor of the exception.
- *Exception*: Is a standard implementation of the Throwable interface and extends the AbstractException class.
- *FatalException*: FatalException is of higher priority then Exception.
- *As2lib* already defines some exceptions:
  - IllegalArgumentException
  - IllegalStateException
  - UndefinedPropertyException
  - ...

Usage: In the case of an incorrectly passed parameter, an IllegalArgumentException can be thrown with the following code:

```
import org.as2lib.env.except.IllegalArgumentException;
...
throw new IllegalArgumentException("Wrong Parameter.",
  this,
  arguments);
```

For all exceptions three parameters are necessary:

1. *message* e.g.: “Wrong Parameter.”- A text which makes the reason of the exception clear.

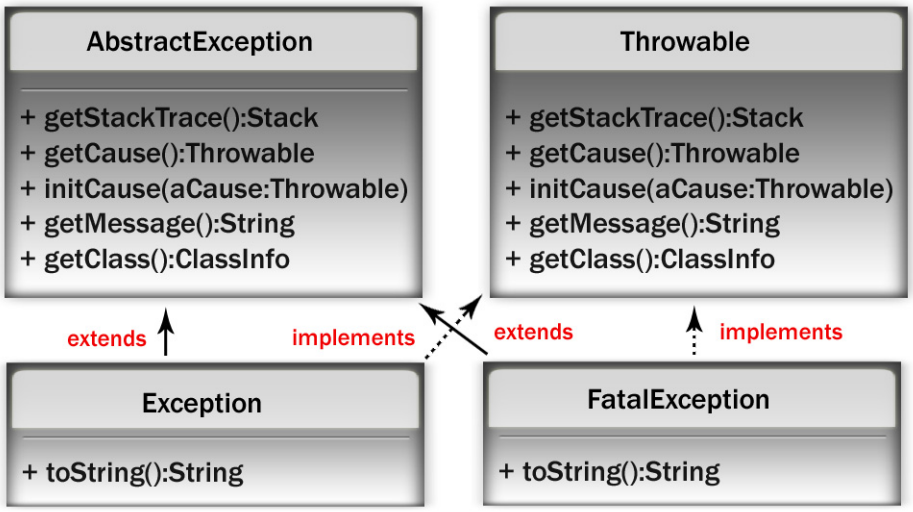

Figure 4.2: Basic hierachy of the *as2lib Exception* package

2. *thrower* e.g.: "this" - Reference to the class or the object that threw the exception.

3. *arguments* - An intrinsic variable of Flash, which contains parameters passed to the method.

A thrown IllegalArgumentException can be caught in the calling code with a *try-catch* block.

```
import org.diplomarbeit.ExceptionTest;
import org.as2lib.env.except.IllegalArgumentException;
import org.as2lib.env.out.handler.TraceHandler;
import org.as2lib.env.out.Out;

try {
  var myOut:Out = new Out();
  myOut.addHandler(new TraceHandler());
  var myET:ExceptionTest = new ExceptionTest();
  myOut.info(myET.getString());
}
catch(e:IllegalArgumentException){
  myOut.error(e);
}
```

You can define as many of your own exceptions as you want. If you for example want to create an OutOfTimeException class you must do the following:

```
import org.as2lib.env.except.Exception;

class OutOfTimeException extends Exception {

public function OutOfTimeException(message:String,
  thrower, args:FunctionArguments) {
  super (message, thrower, args);
}
}
```

The Exception class is extended and the constructor of the Exception class is called.

Chapter 5

Event Handling

Motivation: Events are pretty performance intensive in *Flash* and are a fundamental part of user interface development. Most developers use the intrinsic *AsBroadcaster*(an undocumented feature of Flash) or *EventDispatcher*[1] classes of *Macromedia* for this functionality. Exact definitions aren't available for *EventListener* nor for single events or for arguments. Some information and definitions of listeners for developers are also missing.

Solution: Problems to cope with:

- Object developer has to define which events can be caught.
- Listener developer has to implement all events.
- Object developer has to be able to dispatch events.
- Object developer should be able to add information to a specific event.

The *as2lib* supports *Event Handling* because it´s a core part of application development. If you use the Flash intrinsic *AsBroadcaster* it can result in an inefficient implementation. The *Macromedia EventDispatcher* isn´t free, i.e. is only available when you have purchased Macromedia Flash, and doesn´t support all of the required functionality.

The most important interface of the *event* packages a developer gets in touch with is the *EventBroadcaster*[2]. You can add as many listeners as you like,

```
addListener(listener:EventListener)
```

and remove them if you wish to.

```
removeListener(listener:EventListener)
```

[1] mx.events.EventDispatcher
[2] org.as2lib.env.event.EventBroadcaster

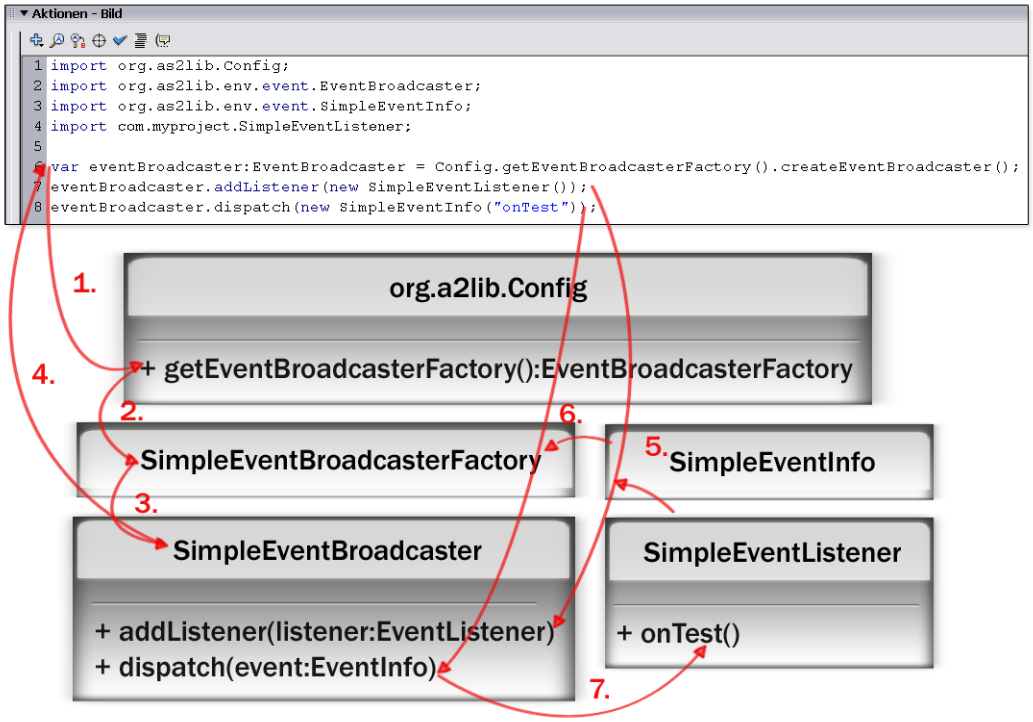

Figure 5.1: Process of an *as2lib* Event Handling example.

Listener objects have to implement the *EventListener* Interface. For your own projects it is recommended to implement and use your own EventListener interface.

If you like to use the *EventBroadcaster* just do it the following way. The figure 5.1 on page 13 shows a simple example application of *as2lib* Event Handling. Aside from the SimpleEventListener only *as2lib* classes are used.

```
import org.as2lib.env.event.EventListener;
import org.as2lib.core.BasicClass;

class com.myproject.SimpleEventListener
  extends BasicClass implements EventListener {

  public function onTest(){
    trace("onTest");
  }
}
```

Chapter 6

Reflections

Motivation: In Java it is possible to obtain information about the name of a class, its methods and properties via the *Reflections API*. This built-in functionality of Java is not integrated in *ActionScript 2* and therefore supported by *as2lib*.

Solution: To use these functionalities in *Flash* the reflections were implemented after the following schema. Starting from the "'root package"' *_global* the ActionScript classes are searched through. If an object is found, it is recognized as a package. A sub-object of type function is marked as a class.

One usage among many of *Reflections* in *as2lib* is for instance the BasicClass, see chapter 2. The *getClass* method of the *BasicClass* uses the *getClassInfo* method of the *ReflectUtil*[1] class and returns a *ClassInfo* instance that provides all important information about the class. The *reflect*[2] package uses different algorithms to get the information about the class. The collection containing all algorithms is located in the *algorithm*[3] package of the *as2lib*. The functionality can, of course, also be accessed directly through the *ReflectUtil* class.

```
import org.as2lib.env.util.ReflectUtil;
import org.as2lib.env.reflect.ClassInfo;
import edu.test.TestClass;

var test:TestClass = new TestClass();
var classInfo:ClassInfo = ReflectUtil.getClassInfo(
        test);
```

The created *ClassInfo* instance contains methods which allow the developer

[1] org.as2lib.env.util.ReflectUtil
[2] org.as2lib.env.reflect
[3] org.as2lib.env.reflect.algorithm

to get further information about the class. Its methods are:

- **getName**():*String* - Name of the class e.g.: "'TestClass"'.
- **getFullName()**:*String* - Fully qualified name of the class including information about the package e.g.: "'edu.test.TestClass"'.
- **getType()**:*Function* - Reference to the class the information is about.
- **getConstructor()**: *ConstructorInfo* - Returns the constructor of the class wrapped into a ConstructorInfo instance.
- **getSuperType**():*ClassInfo* - Information about the superclass, if existing.
- **newInstance(args:Array)** - Creates a new instance of the class the information is about.
- **getParent()**:*PackageInfo* - Information about the package the class is located in.
- **getMethods()**:*Array* - Array, see chapter 7, containing information about every single method of the class.
- **getMethod(methodName:String)**:*MethodInfo* - Returns information about the method whose name was passed as a ClassInfo instance.
- **getMethod(concreteMethod:Function)**:*MethodInfo* - Returns information about the given method in form of a MethodInfo instance.
- **getProperties()**:*Array* - Array containing information about properties of the class specified by getters and setters.
- **getProperty(propertyName:String)**:*PropertyInfo* - Returns information about the property whose name was passed in form of a PropertyInfo instance.
- **getProperty(concreteProperty:Function)**:*PropertyInfo* - Returns information about the given property as a PropertyInfo instance.

The connection between the info classes is shown in figure 6.1 on page 16.

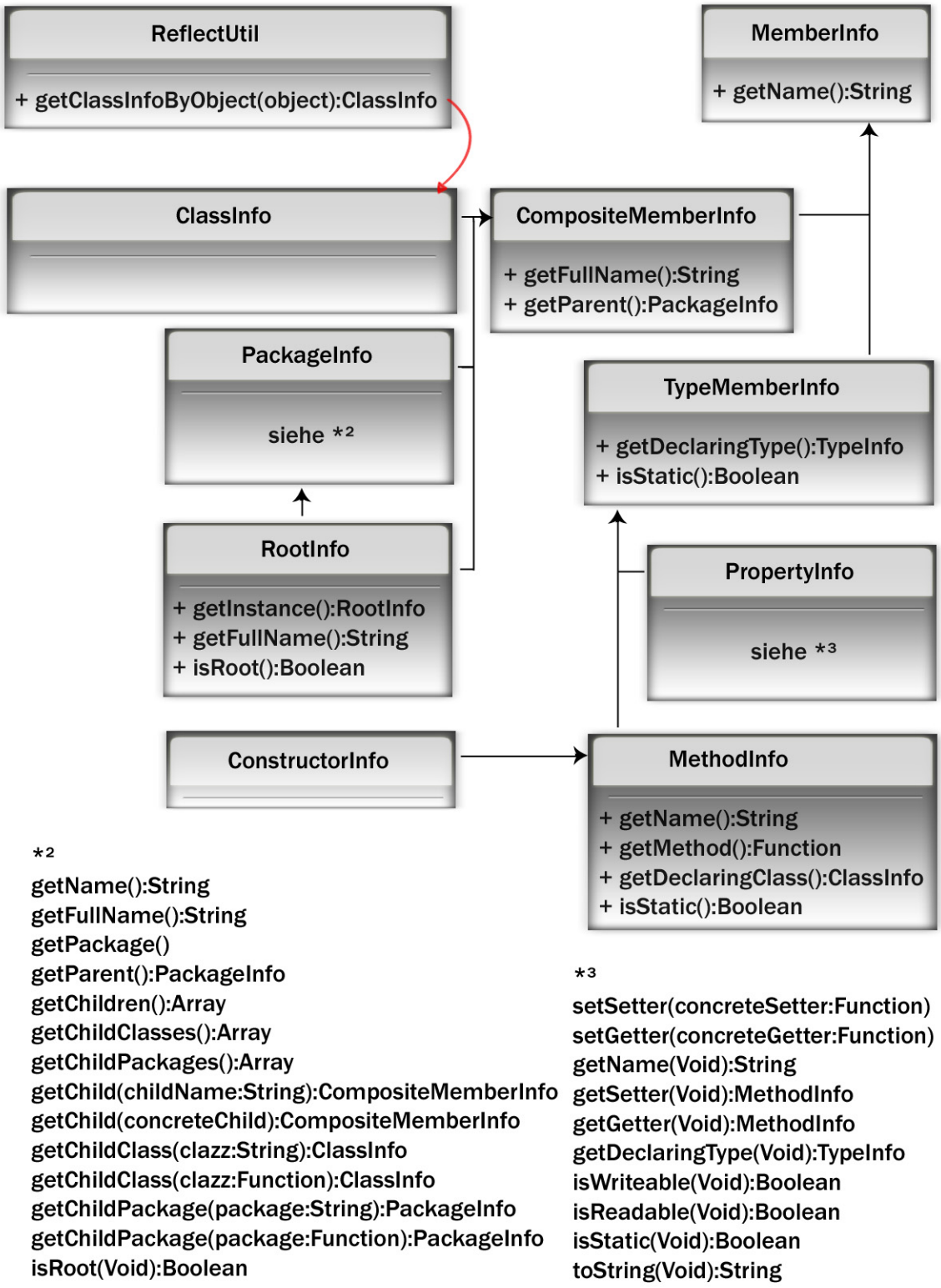

Figure 6.1: Hierarchy of the Info Classes in the *reflect* package

Chapter 7

Data Holding

Motivation: A typical problem when using *ActionScript 2* is that some data types (e.g. Array) can contain different data types (e.g. String, Number,...). This flexible notation could result, especially in teams, in misuse of these data holders.

Solution: *As2lib* not only provides a strictly typed Array class(TypedArray[1]) but also a lot of other data types for data holding.

TypedArray: The TypedArray class provides strict typing of an Array. It extends Array and also makes it possible to use a *ArrayIterator*[2].

```
import org.as2lib.data.holder.array.TypedArray;

var myA:TypedArray = new TypedArray(Number);
myA.push(2);
myA.push("Hello");
```

In this code snippet an Array of type Number is created. If you try to add a string to the TypedArray(*myA.push*("*Hello*")) the compiler throws the following exception:

```
** FatalLevel **
org.as2lib.env.except.IllegalArgumentException:
Type mismatch between [Hello] and [[type Function]].
  at TypedArray.validate(Hello)
```

Additionally you can pass an existing Array as second parameter value. TypedArray provides the same functionality as a regular *Macromedia* Array class.

More *as2lib* data types are:

[1] org.as2lib.data.holder.array.TypedArray
[2] org.as2lib.data.holder.array.ArrayIterator

- **Map**[3]: A data type which holds keys and their values. It has every common method of a regular Map (see Java).

```
import org.as2lib.data.holder.Map;
import org.as2lib.data.holder.map.HashMap;

var aPerson:Person = new Person("Christoph",
  "Atteneder");
var bPerson:Person = new Person("Martin",
  "Heidegger");

var nickNames:Map = new HashMap();
nickNames.put(aPerson,"ripcurlx");
nickNames.put(bPerson,"mastaKaneda");

trace(nickNames.get(aPerson));
trace(nickNames.get(bPerson));
```

Output:

```
ripcurlx
mastaKaneda
```

The *map* package also includes:

  - MapIterator
  - MapStringifier
  - PrimitiveTypMap
  - PriorityMap
  - TypedMap

- **Stack**[4]: You can add values to a Stack with the *push* method and remove them with the *pop* method. Only the last added value can be accessed.

```
import org.as2lib.data.holder.Stack;
import org.as2lib.data.holder.stack.SimpleStack;

var myS:Stack = new SimpleStack();
myS.push("uuuuuuup?!");
myS.push("what's");
myS.push("Hi");
```

[3] org.as2lib.data.holder.Map
[4] org.as2lib.data.holder.Stack

```
trace(myS.pop());
trace(myS.pop());
trace(myS.pop());
```

Output:

```
Hi
what's
uuuuuuup?!
```

The *stack* package also includes:

- StackStringifier
- TypedStack

- **Queue**[5]: In contrast to the Stack you can only access the first element. You can add values with the *enqueue* method and remove them with the *dequeue* method. If you don´t want to remove the first element while accessing it you can use the *peek* method.

```
import org.as2lib.data.holder.Queue;
import org.as2lib.data.holder.queue.LinearQueue;

var aLQ:Queue = new LinearQueue();
aLQ.enqueue("Hi");
aLQ.enqueue("whats");
aLQ.enqueue("uuuuup?!");
trace(aLQ.peek());
trace(aLQ.dequeue());
trace(aLQ.dequeue());
trace(aLQ.dequeue());
```

Output:

```
Hi
Hi
whats
uuuuup?!
```

The *queue* package also includes:

- QueueStringifier
- TypedQueue

[5] org.as2lib.data.holder.Queue

In addition to these data types *Iterators*[6] are also provided:

- **ArrayIterator** : Because of the fact that the additional data types are based internally on Arrays, every special Iterator indirectly uses the ArrayIterator[7].
- **MapIterator** : If you call the *iterator()* method in a HashMap it returns a MapIterator to you.
-

If you like to print every element of a HashMap, you can do so with a MapIterator[8]:

[6] An iterator makes it easier to access elements of a collection without knowing its structure.

[7] org.as2lib.data.holder.array.ArrayIterator

[8] org.as2lib.data.holder.map.MapIterator

```
import org.as2lib.data.holder.Map;
import org.as2lib.data.holder.map.HashMap;
import org.as2lib.data.holder.Iterator;

var aPerson:Person = new Person("Christoph",
  "Atteneder");
var bPerson:Person = new Person("Martin",
  "Heidegger");

var nickNames:Map = new HashMap();
nickNames.put(aPerson,"ripcurlx");
nickNames.put(bPerson,"mastaKaneda");

var it:Iterator = myH.iterator();

while(it.hasNext()){
  trace(it.next());
}
```

Output:

```
Christoph,Atteneder
Martin,Heidegger
```

Figure 7.1 on page 22 shows a hierarchical structure of data holders in the *holder* package.

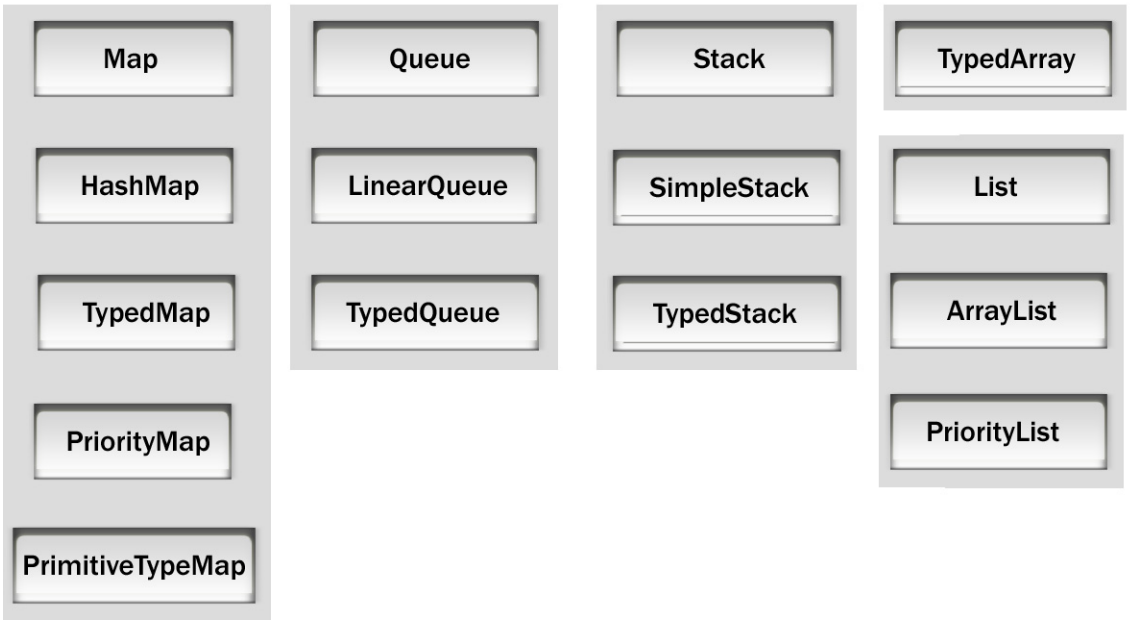

Figure 7.1: data holders of *holder* package

Chapter 8

Overloading

Motivation: As method overloading is supported in Java, a class can have one or more constructors whose only differences are the parameters passed to them. Trying to implement more than one constructor or method with the same name but different parameters in *Actionscript 2* results in the following error:

> "A class must have only one constructor."
> or as the case may be
> "The same member name may not be repeated more than once."

Solution: *As2lib* enables overloading in ActionScript 2 via the *Overload*[1] package. If a class needs three constructors, for example, *as2lib* provides a simple solution.

```
import org.as2lib.env.overload.Overload;

class TryOverload {
  private var string:String;
  private var number:Number;

  public function TryOverload(){
    var overload:Overload = new Overload(this);
    overload.addHandler([Number,
    String],
    setValues);
    overload.addHandler([Number],
    setNumber);
    overload.addHandler([String],
    setString);
    overload.forward(arguments);
```

[1] org.as2lib.env.overload

```
    }
    public function setValues(number:Number,
    string:String){
      this.number = number;
      this.string = string;
    }
    public function setNumber(number:Number){
      this.number = number;
    }
    public function setString(string:String){
      this.string = string;
    }
}
```

In the *TryOverload* class a constructor is created that doesn't define specific parameters. Creating an instance of the class *TryOverload* also creates an Overload object that receives a handler for each additional constructor that is required. In this specific case there is a constructor for Number and String, a constructor for a number and another one for a string. Finally the arguments are passed to the Overload object which calls the appropriate function.

A test of the *TryOverload* class would look like the following:

```
var overload1:TryOverload = new TryOverload("Hello");
var overload2:TryOverload = new TryOverload(6);
var overload3:TryOverload = new TryOverload(6, "y");
```

Chapter 9

Test Cases

A *Test Case* is a class/method that verifies the correct behaviour of a specific class/method. Every developer has to create test cases for his or her own classes and is assisted by a test case API that offers functionality for automated testing.

Motivation: Although two testing API's already exists for *Actionscript 2*, as2unit[1] and asunit[2], our own testing API was created because the following reasons:

- *As2unit* supports only few functionalities (as2unit 7 test methods - as2lib 17 test methods).
- *As2unit* is in spite of the disclosure of the source code still not open source.
- Official documentation of *as2unit* is still not available.
- *As2unit* is only available as a component.
- *As2unit* can always test only one class.
- *As2unit* is not build to support different views.
- *As2unit* does not support pause() and resume() during TestCases (Tests that have a time delay like using onEnterframe).

Solution: Two actions shouldn't be necessary (including the Flash component, setting the parameters) to perform test cases, as it is presently the case with *as2unit*. It should be possible for the developer to execute a test case through only one method call. A direct call of a class as well as a call of a whole package is possible.

[1] asunit.sourceforge.net
[2] www.asunit.org

```
// Use your TestCase here:
new test.org.as2lib.core.TReflections().run();
```

If you have more than one TestCase you can use a simplified way:

```
import org.as2lib.test.unit.TestSuiteFactory;
import test.org.as2lib.core.* // Import Statement, to reduce the Lines.

// References to classes to include them at compile-time.
TestReflections;
TestBasicClass;
TestEventHandling;

new TestSuiteFactory().collectAllTestCases().run();
```

The cases to test must be invoked before the test call so that they are available at run-time. Similar to test case APIs like: JUnit[3], a variety of methods are available for the developer (optional parameters are marked with []):

- **assertTrue**([*message:String*], *testVar1:Boolean*) - An error message will be reported if testVar1 is false.
- **assertFalse**([*message:String*], *testVar1:Boolean*) - An error message will be reported if testVar1 is true.
- **assertEquals**([*message:String*], *testVar1*, *testVar2*) - An error message will be reported if the passed parameters are not equal (!=).
- **assertNotEquals**([*message:String*], *testVar1*, *testVar2*) - An error message will be reported if the passed parameters are equal (==).
- **assertSame**([*message:String*], *testVar1*, *testVar2*) - An error message will be reported if the passed parameters are not the same object (!==).
- **assertNotSame**([*message:String*], *testVar1*, *testVar2*) - An error message will be reported if the passed parameters are the same object (===).
- **assertNull**([*message:String*], *testVar1*) - An error message will be reported if testVar1 is not null.
- **assertNotNull**([*message:String*], *testVar1*) - Error message if testVar1 is null.

[3] www.junit.org

- **assertUndefined**([*message:String*], *testVar1*) - Error message if testVar1 is not undefined.
- **assertNotUndefined**([*message:String*], *testVar1*) - Error message if testVar1 is undefined.
- **assertEmpty** ([*message:String*], *testVar1*) - If testVar1 is neither undefined nor null an error message will be reported.
- **assertIsNotEmpty**([*message:String*], *testVar1*) - If testVar1 is undefined or null an error message will be reported.
- **assertInfinity**([*message:String*], *testVar1*) - Error message if testVar1 is not infinite.
- **assertNotInfinity**([*message:String*], *testVar1*) - Error message if testVar1 is infinite.
- **fail**(*message:String*) - Adds a custom failure message to the whole error output.
- **assertThrows**(*exception:Function, atObject, theFunction:String, parameter:Array*) - If no exception is thrown during the execution of the passed function(theFunction) of the object(atObjekt) with the parameter values(parameter) an error message will be reported.
- **assertNotThrows**(*exception:Function, atObject, theFunction:String, parameter:Array*) - If an exception is thrown during the execution of the passed function(theFunction) of the object(atObject) with the parameter values(parameter) an error message will be reported.

A test case must extend the class TestCase[4]. All methods of the test case that start with "test" are executed by the testing API. The following coding example should show you how to use the Test Case API. A new instance of the Testcase will be created before each start of a method call to create a clean base for the method. .setUp() will be executed before each call to prepare the instance. .tearDown() will be executed after each call to clear the instance(for example: to close a open server connection).

```
import org.as2lib.test.unit.TestCase;
import org.as2lib.core.BasicClass;
import org.as2lib.env.reflect.ClassInfo;

class test.org.as2lib.core.TReflections extends TestCase {
  private var clazz:BasicClass;
```

[4] org.as2lib.test.unit.TestCase

```
  public function TReflections(Void) {}

  public function setUp(Void):Void {
    clazz = new BasicClass();
  }

  public function testGetClass(Void):Void {
    var info:ClassInfo = clazz.getClass();

    assertEquals(
      "The name of the basic class changed",
      info.getName(),
      "BasicClass");

    assertEquals(
      "Problems evaluating the full name",
      info.getFullName(),
      "org.as2lib.core.BasicClass");
  }

  public function tearDown(Void):Void {
    delete clazz;
  }
}
```

Chapter 10

Speed Tests

Motivation: Testing applications for their performance is inevitable because success of the application depends largely on performance.

Solution: A speed test can be executed according to the following schema:

- Creation of single test cases. Example for MyAddSpeedTest:

```
import org.as2lib.test.speed.TestCase;

class MyAddSpeedTest implements TestCase {
  private var a:Number = 0;
  public function run (Void):Void {
    a++;
  }
}
```

- Importing the SpeedTest and the Config class.

```
import org.as2lib.test.speed.Test;
import org.as2lib.Config;
```

- Creating an instance of the test class and passing the OutputHandler to it.

```
var test:Test = new Test();
test.setOut(Config.getOut());
```

- Setting the quantity of the test cases to be executed.

```
test.setCalls(1000);
```

- The test can start after the test cases to be executed have been added. The parameter value *true* in *test.run()* causes an immediate output of the test results.

```
test.addTestCase(new MyAddSpeedTest());
test.addTestCase(new MyMinusSpeedTest());
test.run(true);
```

Output:

```
** InfoLevel **
-- Testresult [2000 calls] --
187% TypedArrayTest: total time:457ms;
  average time/call:0.2285ms; (+0.106ms)
111% ArrayTest: total time:272ms;
  average time/call:0.136ms; (+0.014ms)
[fastest] 100% ASBroadcasterTest: total time:245ms;
  average time/call:0.1225ms;
175% EventDispatcherTest: total time:428ms;
  average time/call:0.214ms; (+0.092ms)
191% EventBroadcasterTest: total time:469ms;
  average time/call:0.2345ms; (+0.112ms)
```

The output of the SpeedTests show the quantity of calls, the elapsed overall time and the average time used. It identifies the fastest test case and displays the speed of the remaining test cases as percentages proportional to this test case.

Chapter 11

Preview

11.1 Connection Handling

Motivation: Connections to external data sources can be established in *Flash MX 2004* in different ways. External data sources can be accessed through Flash Remoting, Web Services, XML Socket Connection. The loading of XML or text files, requests to a URL (e.g.: CGI, PHP,...) or connections between separate SWF files(LocalConnection) can be accomplished as well. The implementation of these different data interfaces can at times differ greatly so that every data interface has to be implemented individually. Although data components in *Flash MX Professional 2004* already provide for certain data interfaces (Web Services and XML files), they differ in their parameter values, besides the fact that you have to own the Professional version. Pure ActionScript 2 solutions don´t exist for every data interface and they also can´t be used the same way.

Solution: *As2lib* provides a standard interface for each data interface. Every Connection is based on a Proxy, which in contrast to the regular Flash Remoting Proxy is strictly typed and allows for compile time checking.

11.2 as2lib console

11.2.1 Requirements

During the development of Flash applications you can use the console and the debugger. But the moment your application is published on a web server and problems occur at this stage, it is pretty hard to identify the source of error. This circumstance costs time and money. It should be possible to print your error and status output independently from your development environment. Because of this as2lib provides an external console to offer this functionality.

Requirements for the *as2lib console* are:

- Display of *as2lib* Output Handling, see chapter 3 on page 5, should be possible inside your development environment and also in online Flash applications.
- Display and debugging of connections to external data sources in your Flash application.
- Display and debugging of events in your Flash application.
- Display of all objects in your Flash applicaton in an object tree.
- Detailed information about every single MovieClip.
- RAM usage of each element.

11.2.2 The as2lib console

In the first prototype of the *as2lib console* we use *as2lib* Output Handling, see chapter 3 on page 5, and Connection Handling, see section 11.1 on page 32. The first basic functionality we support in the *as2lib console* is shown in figure 11.1 on page 33. The prototype supports debug output(e.g.: *Out.debug()*, *Out.info()*,...) of different Flash applications that are connected to the *as2lib console*. Flash applications can be located in a browser *Flash Player* as well as in the Flash development environment.

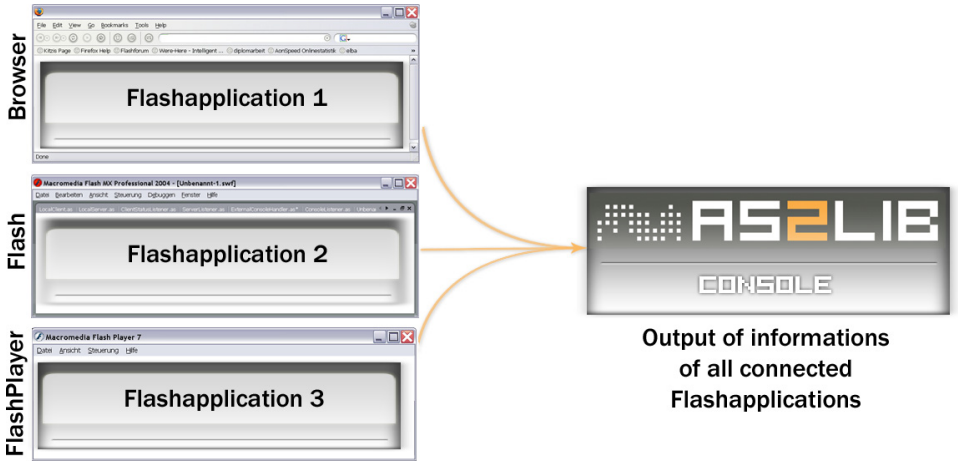

Figure 11.1: Usage of the *as2lib console*

Starting from this requirement a tentative draft was made which provides the desired functionality(see fig. 11.2, p. 34). The console consists of different tabs to display the output for each *OutLevel*. Tab *All* displays all Output.

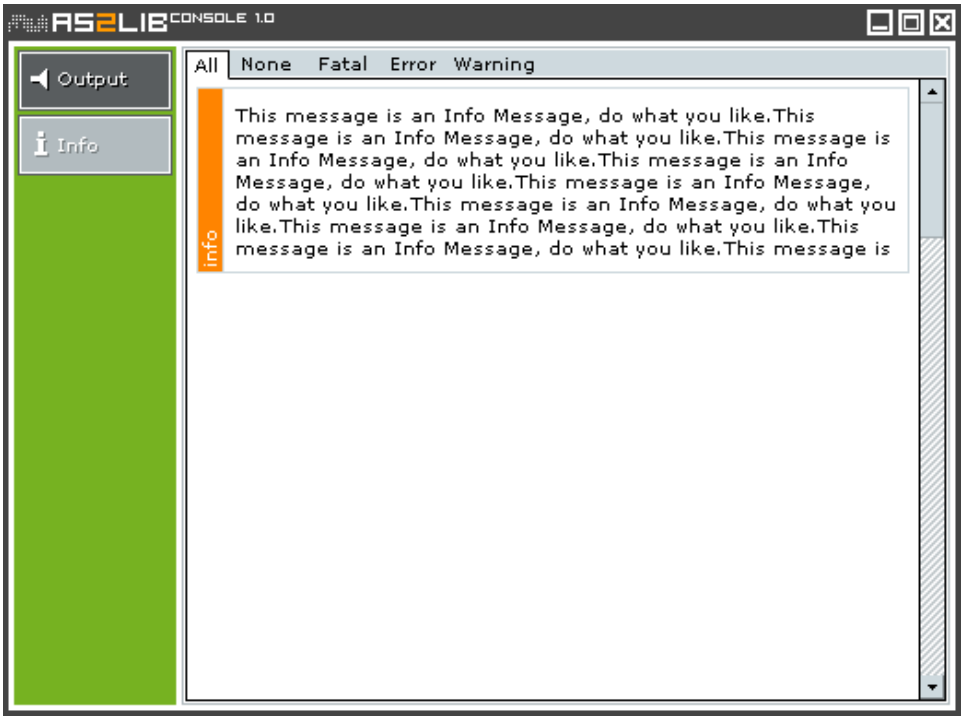

Figure 11.2: tentative draft of the *as2lib* console

11.3 Roadmap

see www.as2lib.org/roadmap.php