as2lib - eine kleine Einführung

Christoph Atteneder, Martin Heidegger,
Michael Herrmann, Alexander Schliebner und Simon Wacker

3. September 2004

Inhaltsverzeichnis

Kapitel 1

Enstehungsgeschichte

Abbildung 1.1: www.as2lib.org

Die *as2lib*, eine *Open Source ActionScript 2* Library, wurde September 2003 als Projekt zur Schaffung besserer Programmiermöglichkeiten unter *ActionScript 2* ins Leben gerufen. Das Projektteam setzt sich mit den Kernproblemen von Flash auseinander und versucht gezielt die täglichen Probleme beim Programmieren mit Flash zu beheben. Als wichtiges Merkmal lässt sich hervorheben, dass die *as2lib* unter der *MPL* (Mozilla Public License) veröffentlicht wird. Dies bedeutet freie Benutzung in privaten, wie kommerziellen Projekten. Nach der Festlegung der Programmierrichtlinien und dem Grundkonzept, wurde die Arbeit der einzelnen Packages von fünf Projektmitarbeitern in Angriff genommen(siehe Tab. 1.1).

Name	*Aufgabe*	*Website*	*Nationalität*
Atteneder Christoph	Entwickler	www.cubeworx.com	Austria
Heidegger Martin	Projektleitung	www.traumwandler.com	Austria
Herrmann Michael	Entwickler		Austria
Schliebner Alexander	Mitinitiator	www.schliebner.de	Germany
Wacker Simon	Chef Entwickler	www.simonwacker.com	Germany

Tabelle 1.1: Aktive Mitglieder *as2lib*

Kapitel 2

Core Package

Beweggrund: Das Festlegen und zur Verfügung stellen gewisser Funktionalitäten in allen Klassen vereinfacht die Entwicklung und Fehlerbehebung während der Entwicklung.

Lösungsansatz: Alle Klassen, Interfaces und Packages der *as2lib* unterliegen gewissen Vorgaben. Die wichtigsten Kernklassen befinden sich im ***core***-*Package*.

2.1 BasicInterface

Zur besseren Definition von Klassenfunktionalitäten wird in der *as2lib* intensiv von *Interfaces* Gebrauch gemacht. Jedes erstellte Interface der *as2lib* erweitert das *BasicInterface*, um folgende Funktionalität in jeder *as2lib* Klasse sicherzustellen:

- **getClass()**:*ClassInfo* - Diese Methode liefert genauere Informationen zur Klasse, in der diese Funktion aufgerufen wird. Die zurückgegebene Information ist vom Typ *ClassInfo* und beinhaltet zusätzlich deren Klassennamen, Informationen wie Methoden, Eigenschaften, Klassenpfad und Superklasse.
- **toString()**:*String* - Diese Methode gibt einen String der die Klasse repräsentiert zurück.

Die Logik der *getClass* Methode wird in der *BasicClass* Klasse zur Verfügung gestellt (siehe Abb. 2.1, S. 4).

2.2 BasicClass

Die Grundklasse der *as2lib* ist die *BasicClass* Klasse. Alle Klassen der *as2lib* sind direkt oder indirekt von der *BasicClass* Klasse abgeleitet. Sie imple-

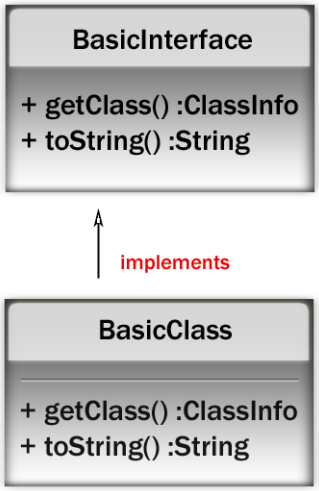

Abbildung 2.1: Hauptbestandteile des *core* Packages

mentiert das *BasicInterface* und stellt über die *ReflectUtil*[1] Klasse und die *ObjectUtil*[2] Klasse die Logik für folgende Methoden zur Verfügung:

- **getClass()**:*ClassInfo* - Erklärung siehe BasicInterface. Das Erstellen der Klasseninformationen wird durch das *as2lib reflection* Package, siehe Kapitel 6, ermöglicht.
- **toString()**:*String* - Diese Methode liefert eine Darstellung der Klasse, vom Typ String, zurück.

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

Kapitel 3

Output Handling

Beweggrund: Die normale Ausgabe von Applikationen in Flash wird über den internen Befehl

```
trace(ausdruck);
```

durchgeführt. Die trace Ausgabe ist jedoch nur in Entwicklungsumgebungen möglich, die den Befehl auch unterstützen. Bei allen anderen Fällen (z.B.: in einer Webapplikation) ist keine Standardausgabe definiert. Eine Library sollte eine standardisierte Ausgabe sowohl für den User als auch für den Entwickler zu Verfügung stellen und überall möglich sein.

Lösungsansatz: Um in jeder Laufzeitumgebung[1] eine oder mehrere Ausgabemöglichkeiten zu haben, wird die *Out*[2] Klasse verwendet. So kann z.B.: eine Webapplikation Fehlermeldungen auch serverseitig abspeichern, um sicherzustellen das Fehler sich nicht nur beim Kunden bemerkbar machen. Die *Out* Klasse behandelt alle eingehenden Anfragen („Ausgabeversuche“) und leitet sie in Abhängigkeit von der Konfiguration an einen, oder mehrere *OutputHandler*[3] weiter. Die *as2lib* bietet eine standardisierte Ausgabemöglichkeit für beliebig viele Schnittstellen.

Anwendung: Ein einfacher Anwendungsfall des *as2lib* Output Handling wird in Abbildung 3.1 auf Seite 6 dargestellt. Nachdem eine Instanz der *Out* Klasse erstellt und der zur Verfügung stehende TraceHandler[4] hinzugefügt wurde kann eine Ausgabe erfolgen. Der Ablauf der einzelnen Aktionen entspricht der Reihenfolge ihrer Nummerierung.

[1] Flash kann im Player, in *Macromedia Central* oder in einer kompilierten Applikation(*.exe) laufen

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

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

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

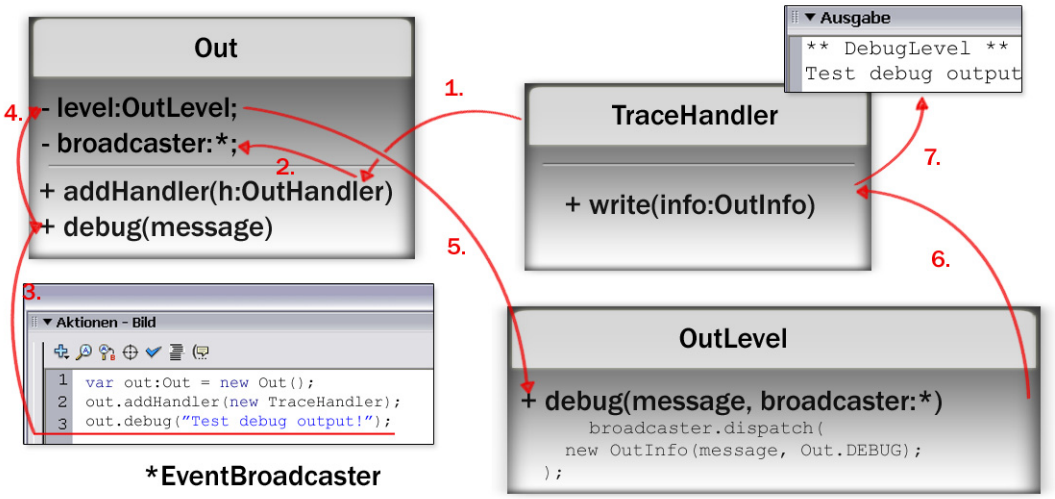

Abbildung 3.1: Anwendungsfall des *as2lib* Output Handlings

Es kann wie in Abbildung 3.1 auf bereits definierte Ausgaben wie den *TraceHandler* zugegriffen oder ein eigener *OutHandler* erstellt werden(z.B.: eine Ausgabe über die *Macromedia* Alert Komponente)[5]:

```
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());
  }

}
```

Durch die Festlegung von Ausgabe Levels(z.B.: *out.setLevel(Out.DEBUG)*)

[5]Die Macromedia Alert Komponente muss sich in der Bibliothek befinden und Sie müssen Flash MX 2004 Professional besitzen, um über die Alert Klasse darauf zugreifen zu können.

kann die Ausgabe bestimmter Informationen verhindert werden und unterschiedliche Ausgabe Levels verwenden. Die möglichen Ausgabe Levels sind:

- Out.ALL
- Out.DEBUG
- Out.INFO
- Out.WARNING
- Out.ERROR
- Out.FATAL
- Out.NONE

Diese Abstufung ermöglicht einerseits ein übersichtlicheres Debuggen bei der Entwicklung, als auch ein schnelleres Umschreiben der Ausgabe in einer fertigen Applikation. Verwendet man bei der Entwicklung z.B.: *Out.DEBUG* werden bei dieser Konfiguration alle Informationen die sich in einem niedrigeren Level, als das gesetzte(*DEBUG*), befinden ausgegeben: *DEBUG*, *INFO*, *WARNING*, *ERROR*, *FATAL*. Nur die *LOG* Ausgabe wird unterdrückt.

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

out.setLevel(Out.DEBUG);

out.log("log mich bitte!");
out.debug("debug mich bitte!");
out.info("informiere mich bitte!");
out.warning("warne mich bitte!");
out.error(new Exception("Output Error", this,
  arguments));
out.fatal(new FatalException("Fatal Output Error",
  this, arguments));
```

Möchte man in der fertigen Applikation nur die schwerwiegenden Fehler ausgeben, lässt sich das durch eine einfache Zeile in der Applikation bewerkstelligen.

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

Kapitel 4

Exception Handling

Beweggrund: Nicht abgefangene Fehlermeldungen werden von der Entwicklungsumgebung mit

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

ausgegeben[1]. Neben der Tatsache, dass die ausgegebene Information nur wenig bis gar nicht aufschlussreich ist (Es wird nur „Error“ bzw. der dem Konstruktor übergebene String ausgegeben. Siehe Abb. 4.1) ist eine Anzeige der Fehlermeldungen nur in *Flash MX 2004* möglich.

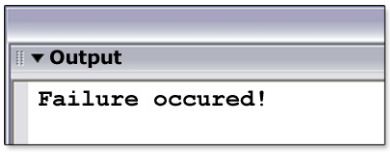

Abbildung 4.1: Ausgabe eines Errors(*throw new Error(„Fehler ist aufgetreten“);*) in Flash MX 2004.

Lösungsansatz: In der *as2lib* stehen, basierend auf der *Macromedia* internen *Error* Klasse, unterschiedliche *Exception* Klassen zur Verfügung, die die Methoden des *Throwable*[2] Interfaces implementieren. Die neuen Funktionalitäten des Exception Handlings sind:

- Alle Operationen die aufgerufen werden, bevor die Exception geworfen wurde, werden in einem Stack abgespeichert, um die Fehlersuche zu beschleunigen. Mit Hilfe von *Reflections*, siehe Kapitel 6, wird der Name der Fehlermeldung in der *as2lib* automatisch generiert.

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

- Exceptions können als Ursache anderer Exceptions gespeichert werden.

Folgene vordefinierte Exception Klassen und Interfaces stehen zur Verfügung:

- *AbstractException*: Alle Exception Klassen werden von der AbstractException Klasse abgeleitet. Sie implementiert die Methoden, die das Throwable Interface definiert (siehe Abb. 4.2, S. 10).
- *Throwable*: Ist ein Interface, dass die Implementierung folgender Methoden erzwingt:
  - **getStackTrace(Void):***Stack* - Gibt einen Stack aller Operationen zurück, die aufgerufen wurden, bevor diese Exception geworfen wurde.
  - **initCause(cause:Throwable):***Throwable* - Gibt den Grund der Exception an und kann nur einmal gesetzt werden. Die Methode wird normalerweise verwendet wenn eine Exception in eine andere Exception umgewandelt und weitergeworfen wird, um keine Information zu verlieren.
  - **getCause(Void):***Throwable* - Gibt den Grund der Exception zurück.
  - **getMessage(Void):***String* - Gibt die Nachricht, die bei der Erstellung des Exception Objektes im Konstruktor übergeben wurde, zurück.
- *Exception*: Ist eine Standard-Implementierung des Throwable Interfaces und wird von der AbstractException Klasse abgeleitet.
- *FatalException*: Zum Unterschied einer Exception Implementierung, hat eine FatalException eine höhere Priorität bzw. ein höheres Level als eine normale Exception, das abgefragt werden kann.
- In der *as2lib* sind bereits einige Exceptions definiert:
  - IllegalArgumentException
  - IllegalStateException
  - UndefinedPropertyException
  - ...

Anwendung: Bei einer fehlerhaften Parameterübergabe kann mit folgendem Code eine IllegalArgumentException geworfen werden:

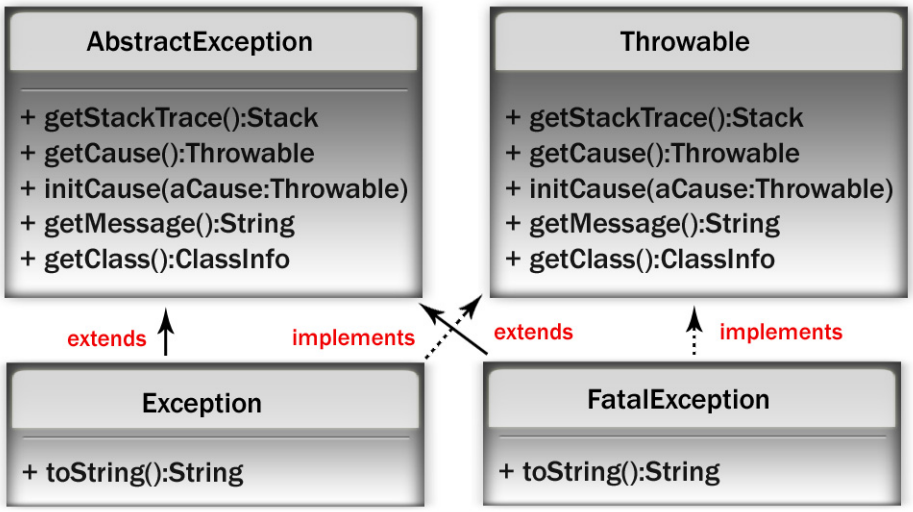

Abbildung 4.2: Grundlegende Hierarchie des *as2lib Exception* Packages

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

1. **message** - Ein beliebiger Text, der den Grund der Exception deutlicher hervorbringen soll.
2. **thrower** - Referenz auf die Klasse bzw. das Objekt, dass die Exception geworfen hat.
3. **arguments** - ist eine intrinsische Variable von Flash, die alle Parameter einer Funktion beinhaltet

Eine geworfene IllegalArgumentException kann im ausführenden Code mit einem *try-catch* Block abgefangen wird.

```
import com.tests.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);
}
```

Es können beliebig viele eigene Exception definiert werden. Möchte man zB.: eine OutOfTimeException werfen, muss man folgende Implementierung vornehmen:

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

class org.as2lib.env.except.OutOfTimeException
  extends Exception {

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

In dem vorhergehenden Codebeispiel wird die Exception Klasse abgeleitet und der Konstruktor der Exception Klasse aufgerufen.

Kapitel 5

Event Handling

Beweggrund: Ereignisse(*Events*) brauchen in *Flash* sehr viel Rechenleistung und sind ein grundlegender Bestandteil bei der Entwicklung von Benutzeroberflächen. Viele Entwickler verwenden den in *Flash* inkludierten *AsBroadcaster*[1] oder die *EventDispatcher*[2] Klasse von *Macromedia* für diese Problematik. Jedoch gibt es weder genaue Spezifikationen für *EventListener* noch für einzelne Events und deren Übergabewerte. Die Festlegung von Schnittstellen über Interfaces ist für eine „saubere“ Programmierung unumgänglich. Es fehlen unter anderem für Entwickler von *Listenern*(Objekte die auf bestimmte Events warten) wichtige Informationen bezüglich Events und der Übergabeparameter.

Lösungsansatz: Um zu einer Lösung zu gelangen müssen folgende Probleme bewältigt werden:

- Der Objektentwickler muss definieren, welche Events abgewartet werden können.
- Der Listenerentwickler muss alle Events definieren.
- Der Objektentwickler muss die Möglichkeit haben, Events auszulösen.
- Der Objektentwickler sollte genauere Informationen einem Event hinzufügen können.

Die *as2lib* unterstützt *Event Handling*, da es ein Kernstück der Applikationsentwicklung ist. Verwendet man direkt den Flash internen *AsBroadcaster* kann es zu einer ineffizienten Implementierung kommen. Der *EventDispatcher* von *Macromedia* ist nicht frei zugänglich, kann also nur verwendet werden wenn man *Macromedia Flash* käuflich erworben hat, und stellt nicht alle benötigten Funktionalitäten zur Verfügung.

[1] Nicht dokumentiertes Feature von Flash MX 2004
[2] mx.events.EventDispatcher

Das wichtigste Interface des *event* Packages mit der man als Entwickler in Berührung kommt ist der *EventBroadcaster*[3]. Es können mehrere Listener(Zuhörer) zum EventBroadcaster(Ereignissverteiler) hinzugefügt,

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

aber auch wieder entfernt werden.

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

Wie man in den Methodenaufrufen erkennen kann müssen die Listener Objekte das EventListener Interface implementieren. Für eigene Projekte empfiehlt es sich ein eigenes EventListener Interface zu erstellen.

Möchte man den *EventBroadcaster* verwenden kann man seine Funktionalitäten einfach nach folgender Methode erwerben, siehe Abb. 5.1, S. 14. Diese Abbildung zeigte eine einfach Anwendung des *as2lib* Event Handlings. Es werden bis auf den SimpleEventListener nur Klassen der *as2lib* verwendet.

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

class edu.diplomarbeit.listener.SimpleEventListener
  extends BasicClass implements EventListener {

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

Die einzelnen Schritte des as2lib EventHandlings werden durch die nummerierten Pfeile dargestellt.

[3] org.as2lib.env.event.EventBroadcaster

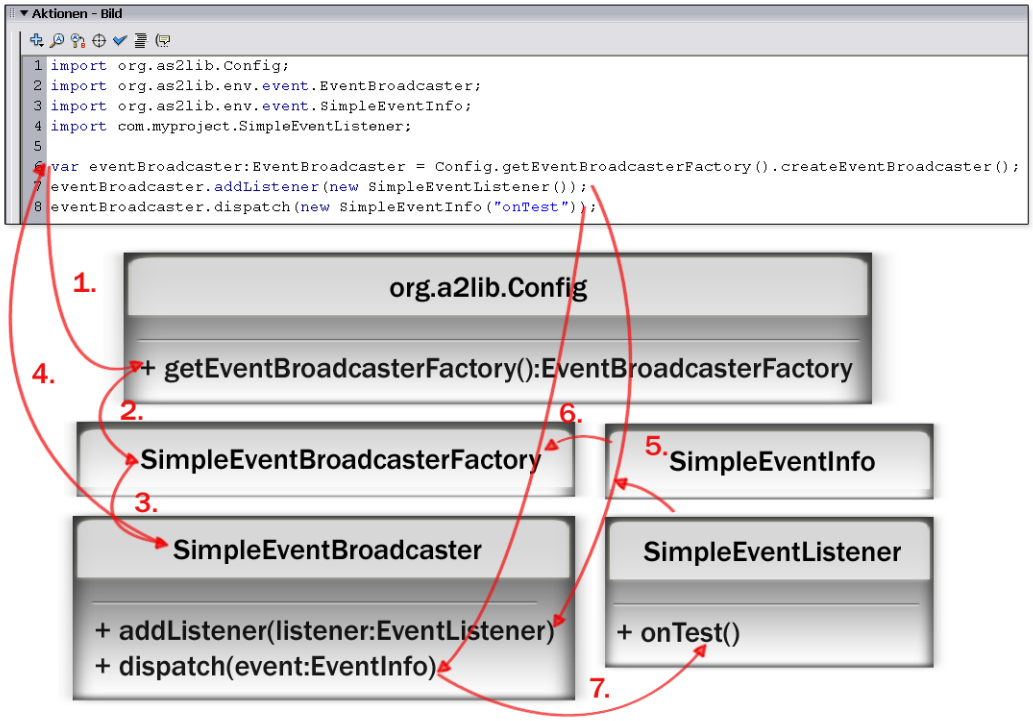

Abbildung 5.1: Ablauf eines einfachen Event Handling Beispiels.

Kapitel 6

Reflections

Beweggrund: In Java ist es möglich Informationen über den Namen der Klasse, deren Methoden und Eigenschaften über Reflections zu ermitteln. Diese eingebaute Funktionalität von Java ist in *Flash* bzw. *Actionscript 2* nicht integriert und wird deshalb von der *as2lib* unterstützt.

Lösungsansatz: Um diese Funktionalitäten in *Flash* verwenden zu können wurden die Reflections nach folgendem Schema implementiert. Ausgehend vom Namensbereich *_global* wird die Actionscript Klassenstruktur durchsucht. Wird ein Objekt gefunden, wird es als Package erkannt. Ein Unterobjekt vom Typ Function wird als Klasse gekennzeichntet.

Eine der vielen Anwendungen von Reflections in der *as2lib* ist zB.: die *BasicClass*, siehe Kapitel 2. Die Methode *getClass*, der BasicClass greift auf die Methode *getClassInfo* der ReflectUtil[1] Klasse zu und gibt ein *ClassInfo* Objekt zurück, dass alle wichtigen Informationen der Klasse zur Verfügung stellt. Das *reflect*[2] Package arbeitet mit unterschiedlichen Algorithmen um an die Klasseninformationen zu gelangen. Die Sammlung aller Algorithmen befindet sich im *algorythm*[3] Package der *as2lib*. Die Funktionaliäten der Reflections können auch direkt über die *ReflectUtil* Klasse verwendet werden.

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

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

Das erzeugte Objekt vom Typ *ClassInfo* beinhaltet Methoden um genauere

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

Informationen der Klasse zu erhalten. Seine Methoden sind:

- **getName**():*String* - Name der Klasse z.B.: „TestClass“.
- **getFullName()**:*String* - Name der Klasse inklusive Packageinformationen z.B.: „edu.test.TestClass“.
- **getType()**:*Function* - Klasse zu der die Klasseninformation erstellt wurde.
- **getConstructor()**: *ConstructorInfo* - Gibt Konstruktor der Klasse als ConstructorInfo zurück.
- **getSuperType**():*ClassInfo* - Informationen zur Vaterklasse, falls vorhanden.
- **newInstance(args:Array)** - Neue Instanz der Klasse zu der die Klasseninformation erstellt wurde.
- **getParent()**:*PackageInfo* - Informationen zum Package in dem sich die Klasse befindet.
- **getMethods()**:*Array* - Array, siehe Kapitel 7, die Informationen zu den einzelnen Methoden der Klasse beinhaltet.
- **getMethod(methodName:String)**:*MethodInfo* - Gibt Information zur Methode dessen Name übergeben wurde als MethodInfo zurück.
- **getMethod(concreteMethod:Function)**:*MethodInfo* - Gibt Information zur Methode die übergeben wurde als MethodInfo zurück.
- **getProperties()**:*Array* - Array, die Informationen zu den einzelnen Eigenschaften, falls sie eine get set Methode besitzen zurückliefert.
- **getProperty(propertyName:String)**:*PropertyInfo* - Gibt Information zur Eigenschaft dessen Name übergeben wurde als PropertyInfo zurück.
- **getProperty(concreteProperty:Function)**:*PropertyInfo* - Gibt Information zur Eigenschaft die übergeben wurde als PropertyInfo zurück.

Die Verknüpfungen der einzelnen Info Klassen zeigt die Abbildung 6.1 auf Seite 17.

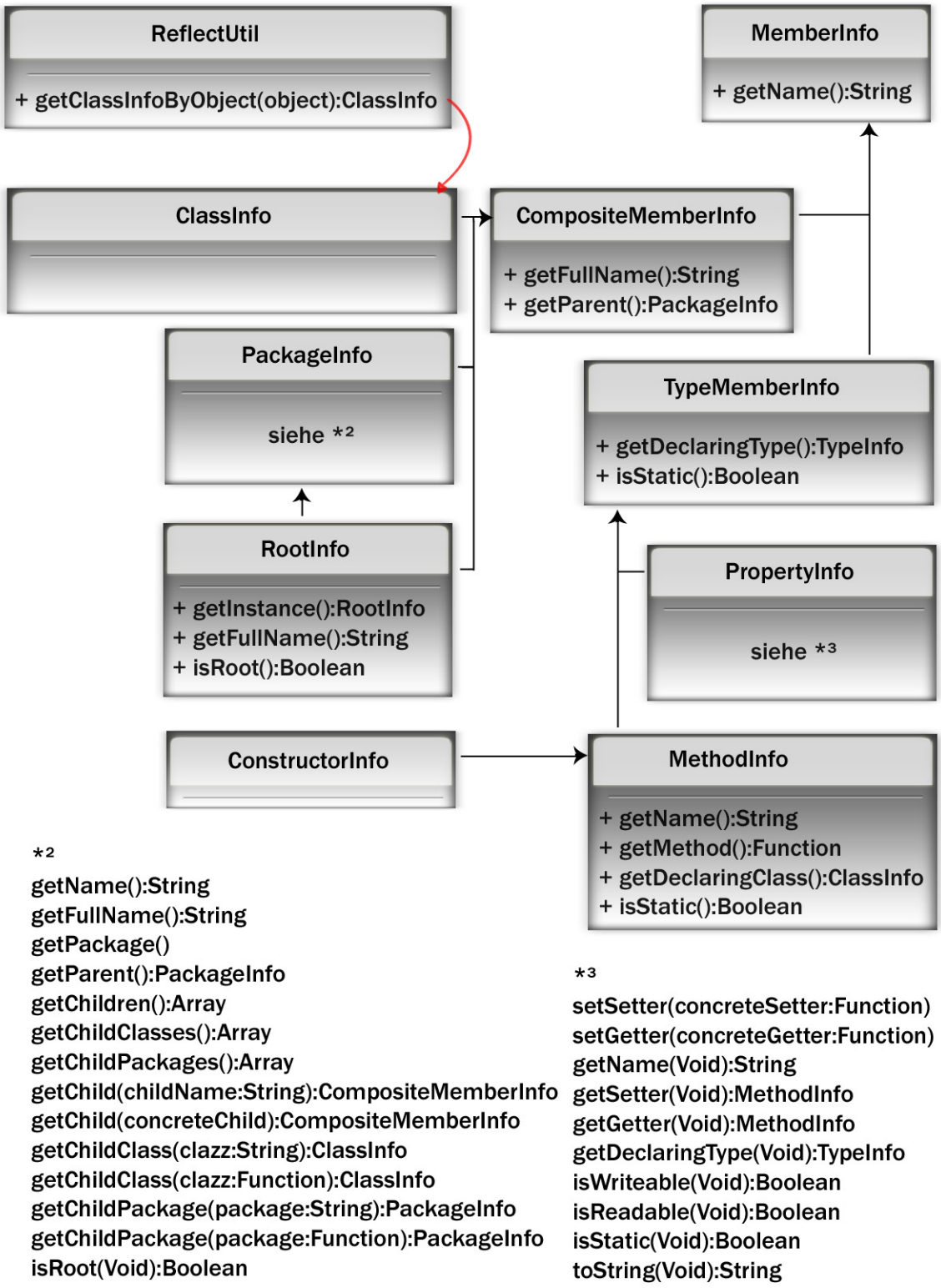

Abbildung 6.1: Hierarchie der Info Klassen im *reflect* Package

Kapitel 7

Data Holding

Beweggrund: Ein typisches Problem bei der Programmierung mit *Actionscript 2* ist das einige Datentypen (zB.: Array) unterschiedliche Datentypen (zB.: *String* und *Number*) beinhalten können. Diese nicht strikte Notation kann vor allem bei einer Arbeit in Teams leicht zu Problemen führen.

Lösungsansatz: In der *as2lib* gibt es nicht nur eine Array Klasse(TypedArray[1]), die nur bestimmte Datentypen zulässt, sondern auch eine Vielzahl anderer Datentypen zur Datenhaltung.

TypedArray: Die Klasse TypedArray erlaubt eine strikte Datentypisierung eines Arrays.

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

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

In dem Codebeispiel wird ein Array vom Typ Number erstellt. Versucht ein Entwickler einen String zum TypedArray hinzuzufügen(*myA.push*(„*Hallo*“)) wirft der Compiler einen Fehler.

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

Zusätzlich zum Typ des Arrays kann dem TypedArray Konstruktor ein bereits bestehendes Array als zweiter Übergabeparameter mit übergeben werden. TypedArray besitzt die gleichen Funktionaliäten wie die normale Array Klasse von *Macromedia*.

[1] org.as2lib.data.holder.TypedArray

Weitere *as2lib* Datentypen sind:

- **HashMap**: Ein Datentyp dem ein Key und der dazugehörige Wert übergeben werden kann. Er besitzt alle Methoden einer normalen HashMap(siehe 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));
```

Ausgabe:

```
ripcurlx
mastaKaneda
```

- **Stack**: Einem Stack können mit der Methode *push* Werte hinzugefügt bzw. mit *pop* wieder entfernt werden. Es kann immer nur auf das oberste Element zugegriffen werden.

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

var myS:Stack = new SimpleStack();
myS.push("gehts?!");
myS.push("wie");
myS.push("Hallo");
trace(myS.pop());
trace(myS.pop());
trace(myS.pop());
```

Output:

```
Hallo
wie
gehts?!
```

- **Queue**: Einer Queue können mit der Methode *enqueue* Werte hinzugefügt bzw. mit *dequeue* wieder entfernt werden. Es kann immer nur auf das erste Element zugegriffen werden. Mit der Methode *peek* kann zusätzlich auf das erste Element zugegriffen werden, ohne es aus der Reihe zu entfernen.

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

var aLQ:Queue = new LinearQueue();
aLQ.enqueue("Hallo");
aLQ.enqueue("wie");
aLQ.enqueue("gehts?!");
trace(aLQ.peek());
trace(aLQ.dequeue());
trace(aLQ.dequeue());
trace(aLQ.dequeue());
```

Output:

```
Hallo
Hallo
wie
gehts?!
```

Zusätzlich zu den Datentypen wurden auch *Iteratoren*[2] implementiert:

- **ArrayIterator**: Da die zusätzlichen *as2lib* Datentypen intern mit Arrays aufgebaut sind, wird bei allen speziellen Iteratoren indirekt der ArrayIterator verwendet.
- **MapIterator**: Wenn die Methode *iterator()* in einer HashMap aufgerufen wird, wird automatisch ein MapIterator zurückgeworfen.

Möchte man z.B.: alle Elemente einer HashMap ausgeben, kann diese Aufgabe mit einem MapIterator leicht gelöst werden.

[2]Ein Iterator ermöglicht den Zugriff auf die Elemente einer Sammlung ohne Kenntnis der Struktur der Sammlung.

```
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());
}
```

Ausgabe:

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

Die Abbildung 7.1 auf Seite 22 zeigt eine hierarchische Auflistung der Datenhalter des *holder* Packages.

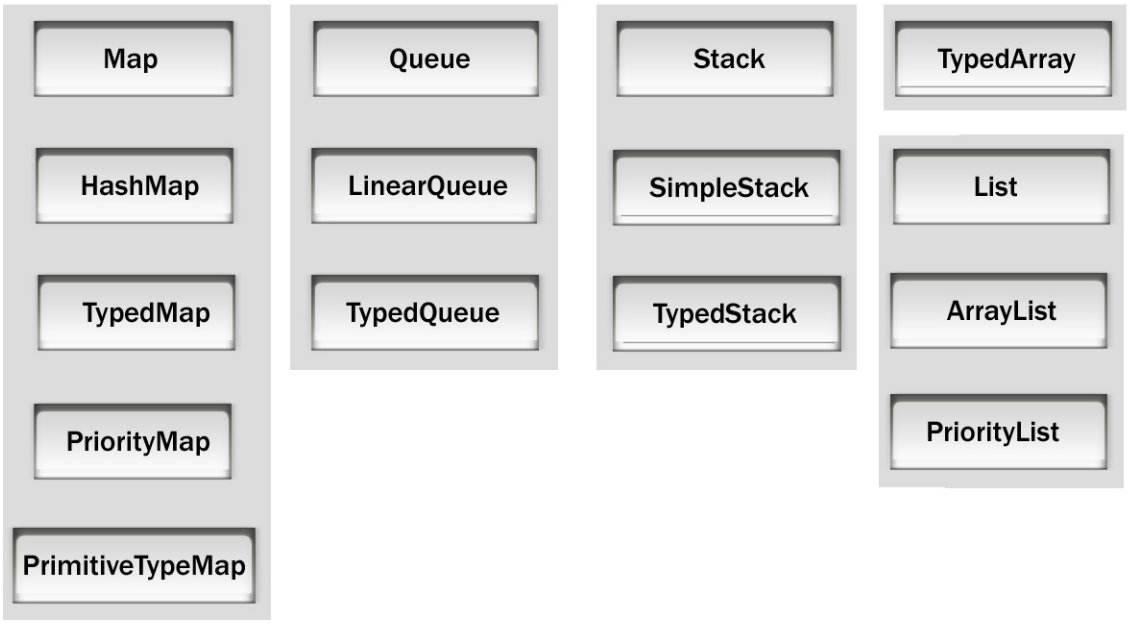

Abbildung 7.1: Datenhalter des *holder* Package

Kapitel 8

Overloading

Beweggrund: Da Überladen von Methoden in Java unterstützt wird können zB.: in einer Klasse zwei Konstruktoren vorhanden sind, die sich nur durch in ihren Übergabeparameter unterscheiden. Versucht man in *Actionscript 2* mehrere Konstruktoren oder Methoden mit dem gleichen Namen und unterschiedlichen Übergabeparametern zu implementieren wird folgende Fehlermeldung ausgegeben:

> „Eine Klasse kann nur einen Konstruktor haben!“
> bzw.
> „Ein Mitgliedsname darf nur einmal vorkommen!“

Lösungsansatz: Die *as2lib* ermöglicht das Overloading in *ActionScript 2* durch das *overload*[1] Package. Benötigt man z.B.: drei Konstruktoren in einer Klasse kann dieses Problem mit der *as2lib* einfach gelöst werden.

```
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],
    setNumber);
    overload.addHandler([String],
    setString);
    overload.forward(arguments);
  }
  public function setNumber(number:Number){
```

[1] org.as2lib.env.overload

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

In diesem Codebeispiel wird in der Klasse *TryOverload* ein Konstruktor erstellt, der keine bestimmten Übergabeparameter definiert. Wird eine Instanz der Klasse *TryOverload* erstellt, wird zusätlich eine Overload Objekt erzeugt, dem für jeden zustätzlichen Konstructor ein eigener Handler(Typ des Übergabeparameters und aufzurufende Methode) übergeben werden muss. In diesem speziellen Fall gibt es einen Konstruktor für eine Number und einen Konstruktor für einen String. Schlussendlich werden die übergebenen Argumente(arguments) dem Overload Objekt übergeben, der die entsprechende Funktion aufruft. Ein Test der TryOverload Klasse würde wie folgt aussehen:

```
var overload1:TryOverload = new TryOverload("Hallo");
var overload2:TryOverload = new TryOverload(6);
```

Kapitel 9

Test Cases

Ein *Test Case*(Testfall) ist eine Klasse/Methode, die eine bestimmte Klasse/Methode auf ihre korrekte Funktionsweise überprüft. Jeder Entwickler muss für seine Klassen Testfälle erstellen und wird von einem Test Case System unterstützt, das Funktionalitäten für ein automatisches Testen zur Verfügung stellt.

Beweggrund: Obwohl bereits zwei Testsysteme für *Actionscript 2* Klassen existieren, as2unit[1] und asunit[2], wurde aus folgenden Gründen ein eigenes Testsystem erstellt:

- *as2unit* unterstützt nur wenige Funktionalitäten(*as2unit* **7** Testmethoden – *as2lib* **17** Testmethoden).
- *as2unit* ist trotz verkündeter Offenlegung des Quellcodes immer noch nicht Open Source.
- Es gibt immer noch keine offizielle Dokumentation der *as2unit*.
- *as2unit* ist nur als Komponente vorhanden.
- *as2unit* kann immer nur eine Klasse testen.
- *as2unit* erlaubt eine Ausgabe nur über trace.
- *as2unit* unterstützt keine Funktionalitäten zum Unterbrechen(pause()) bzw. Weiterführen (resume()) der Abarbeitung von einem Testfall. (Dies ist mitunter erforderlich für Methoden die auf einen response warten müssen. Beispielsweise wenn eine datei geladen wurde.)

[1] www.as2unit.org
[2] asunit.sourceforge.org

Lösungsansatz: Um Testfälle durchführen zu können, sollen nicht zwei unterschiedliche Aktionen notwendig sein(Einbinden der Flashkomponente, setzen der Parameter), wie es momentan bei der *as2unit* der Fall ist. Es soll dem Entwickler möglich sein einen Testfall durch einen einfachen Methodenaufruf durchführen zu können. Sowohl ein direkter Aufruf einer Klasse, als auch ein Aufruf eines gesamten Packages, falls mehrere Test Fälle durchzuführen sind, ist möglich.

```
import org.as2lib.test.unit.Test;

// Hier äTestflle üeinfgen.
test.org.as2lib.core.TReflections;
test.org.as2lib.util.TReflectUtil;
test.org.as2lib.util.TStringUtil;

Test.run("test.org");
```

Die zu testenden Fälle müssen vor dem Testaufruf angeführt werden, damit sie zur Laufzeit verfügbar sind. Ähnlich wie bei Test Case Systemen zB.: JUnit[3] stehen dem Entwickler in der *as2lib* bereits eine Vielzahl von Methoden zu Verfügung(optionale Parameter sind durch [] gekennzeichnet):

- **assertTrue**([*message:String*], *testVar1:Boolean*) - Ist testVar1 false wird eine Fehlermeldung ausgegeben.
- **assertFalse**([*message:String*], *testVar1:Boolean*) - Ist testVar1 true wird eine Fehlermeldung ausgegeben.
- **assertEquals**([*message:String*], *testVar1*, *testVar2*) - Sind die übergebenen Parameter nicht gleich(!=) wird eine Fehlermeldung ausgegeben.
- **assertNotEquals**([*message:String*], *testVar1*, *testVar2*) - Sind die übergebenen Parameter gleich(==) wird eine Fehlermeldung ausgegeben.
- **assertSame**([*message:String*], *testVar1*, *testVar2*) - Sind die übergebenen Parameter nicht die selbe Objektreferenz (!==) wird eine Fehlermeldung ausgegeben.
- **assertNotSame**([*message:String*], *testVar1*, *testVar2*) - Sind die übergebenen Parameter die selbe Objektreferenz (===) wird eine Fehlermeldung ausgegeben.
- **assertNull**([*message:String*], *testVar1*) - Ist testVar1 nicht null wird eine Fehlermeldung ausgegeben.

[3] http://www.junit.org

- **assertNotNull**([*message:String*], *testVar1*) - Ist testVar1 gleich null wird eine Fehlermeldung ausgegeben.
- **assertUndefined**([*message:String*], *testVar1*) - Ist testVar1 nicht undefined wird eine Fehlermeldung ausgegeben.
- **assertNotUndefined**([*message:String*], *testVar1*) - Ist testVar1 gleich undefined wird eine Fehlermeldung ausgegeben.
- **assertEmpty**([*message:String*], *testVar1*) - Ist testVar1 weder undefined noch null wird eine Fehlermeldung ausgegeben.
- **assertNotEmpty**([*message:String*], *testVar1*) - Ist testVar1 undefined oder null wird eine Fehlermeldung ausgegeben.
- **assertInifinity**([*message:String*], *testVar1*) - Ist testVar1 nicht unendlich(d.h. != Infinity) wird eine Fehlermeldung ausgegeben.
- **assertNotInfinity**([*message:String*], *testVar1*) - Ist testVar1 unendlich(d.h. == Infinity) wird eine Fehlermeldung ausgegeben.
- **fail**(*message:String*) - Fügt eine selbst definierte Fehlermeldung zu allen Error-Ausgaben hinzu.
- **assertThrows**(*exception:Function, atObject, theFunction:String, parameter:Array*) - Wird beim Ausführen der übergebenen Funktion(theFunction) des Objektes(atObjekt) mit den Übergabeparametern(parameter) keine Exception geworfen wird eine Fehlermeldung ausgegeben.
- **assertNotThrows**(*exception:Function, atObject, theFunction:String, parameter:Array*) - Wird beim Ausführen der übergebenen Funktion(theFunction) des Objektes(atObjekt) mit den Übergabeparametern(parameter) eine Exception geworfen wird eine Fehlermeldung ausgegeben.

Ein Testfall muss von der *TestCase* Klasse[4] abgeleitet werden. Jede Methode des Testfalles, die mit „test“ beginnt, wird von dem Testsystem aufgerufen. Wie genau ein erstellter Testfall aussehen könnte wir in folgendem Codebeispiel gezeigt: Vor jedem Methodenaufruf wird eine neue Instanz des Testcases erstellt um einen unmodifizierten Ausgangspunkt für die Operation zu ermöglichen. Um die Instanz dezidiert auf das Ausführen einer Methode vorzubereiten gibt es den Befehl setUp(). Dieser wird vor jeder Methode ausgeführt. Über die Methode tearDown(), die jeweils nach jeder methode ausgeführt wird, kann die initialisierung Rückgängig gemacht werden(z.B. um offene Serververbindungen wieder zu schliessen).

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

```
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;

  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;
  }
```

Kapitel 10

Speed Tests

Beweggrund: Das Testen von Applikationen auf dessen Performanz ist unumgänglich, da diese meistens direkt mit dem Erfolg der Applikation zusammenhängt.

Lösungsansatz: Die Durchführung eines Speed Testes lässt sich nach folgendem Schema durchführen:

- Erstellen der einzelnen Testfälle. Beispiel für MyAddSpeedTest:

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

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

- Importieren der SpeedTest und der Config Klasse.

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

- Erstellen einer Instanz der Test Klasse und Übergabe des Output Handlers.

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

- Setzen der Anzahl der durchzuführenden Test Fälle.

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

- Nachdem die durchzuführenden Testfälle hinzugefügt worden sind kann der Test durchgeführt werden. Der Übergabeparameter *true* in *test.run()* bewirkt eine sofortig Ausgabe des Testergebnisses.

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

Ausgabe:

```
** InfoLevel **

 * Testresult [1000 calls] *
 116\% MyAddSpeedTest: total time:29ms;
calls/second:34482;
average time:0.029ms; (+0.004ms)
[fastest] 100\% MyMinusSpeedTest: total time:25ms;
calls/second:40000;
average time:0.025ms;
```

Die Ausgabe des SpeedTests zeigt die Anzahl der Aufrufe, die verstrichene Gesamtzeit und den durchschnittlichen Zeitbedarf. Sie findet den schnellsten Testfall und gibt das prozentuelle Verhalten der anderen Testfälle im Verhältnis zum Schnellsten aus.

Kapitel 11

Ausblick

11.1 Connection Handling

Beweggrund: Verbindungen mit externen Datenquellen kann in *Flash MX 2004* auf verschiedene Art und Weise erstellt werden. Externe Datenquellen können zB.: über *Flash Remoting*, *Web Services*, *XML Socket Connection* angesprochen werden. Aber auch XML- oder Text-Dateien, Anfragen über die URL(zB.: CGI, PHP,...) oder auch Verbindungen zwischen unterschiedlichen SWF-Dateien(LocalConnection) können durchgeführt werden. Die Implementierung dieser unterschiedlichen Datenschnittstellen unterscheidet sich jedoch teilweise massiv und muss für jede Datenquelle speziell durchgeführt werden. Zwar bestehen in *Flash MX 2004 Professional* für bestimmte Datenquellen Verbindungskomponenten(Web Services und XML Dateien), jedoch können sie nur von Flash MX Professional Besitzern verwendet werden und unterscheiden sich in ihren Übergabeparametern. Reine *Actionscript 2* Lösungen bestehen nicht für alle Datenschnittstellen und es kann auch keine standardmäßige Implementierung für alle Datenquellen durchgeführt werden.

Lösungsansatz: Die *as2lib* stellt eine standardisierte Datenschnittstelle zur Verfügung, die alle Datenschnittstellen beinhaltet. Jeder Verbindung liegt ein Proxy zugrunde, dass im Gegensatz zum normalen Flash Remoting Proxy der Verbindung entsprechend typisiert ist und so Überprüfung zur Compile-Time ermöglicht.

11.2 as2lib Debug Konsole

11.2.1 Anforderungen

Bei der Entwicklung von Flash Applikationen steht dem Entwickler nur die Konsolenausgabe zur Verfügung. Wird jedoch die Applikation auf einem

Webserver veröffentlicht und es treten in dieser Phase Probleme auf ist es nur sehr schwer möglich schnell die Fehlerquelle zu ermitteln. Dieser Umstand kostet Zeit und Geld. Es sollte eine zusätzliche Ausgabe von Fehler- und Statusmeldungen außerhalb der Enwicklungsumgebung möglich sein. Aus diesem Grund stellt die *as2lib* eine externe Konsole zur Verfügung die diese Funktionalitäten bietet. Die Anforderungen an die *as2lib console* sind:

- Ausgabe des *as2lib* Output Handling, siehe Kapitel 3 auf Seite 5 soll sowohl in der Entwicklungsumgebung als auch in online Flashapplikationen möglich sein.
- Anzeige und Debugging von Verbindungen zu externen Datenquellen einer Flashapplikation.
- Anzeige und Debugging von Events in einer Flashapplikation
- Darstellung aller Objekte einer Flashapplikation in Baumform
- Informationen zu einzelnen MovieClips
- Speicherverbrauch einzelner Elemente

11.2.2 Die as2lib console

Im ersten Prototypen der *as2lib console* wird vor allem das *as2lib* Output Handling, siehe Kapitel 3 auf Seite 5, und Connection Handling, siehe Abschnitt **??** auf Seite **??** verwendet.

Die grundlegende Funktionalität der *as2lib* Debug Konsole wird in Abbildung **??** auf Seite **??** dargestellt. Im Prototypen werden Debugausgaben(zB.: *Out.debug()*, *Out.info()*,...) mehrerer Flashapplikationen die sich zur Debug Konsole verbinden können in der Konsole ausgegeben. Die Flashapplikationen können sich im Browser im Flash Player oder auch in der Flash Entwicklungsumgebung befinden.

Ausgehend von diesen Anforderungen wurde ein erster grafischer Entwurf erstellt, der die erste geforderte Funktionalität beinhaltet(siehe Abb. **??**, S. **??**). Die Konsole besteht aus mehreren Karteireiter(Tabs) in denen jeweils nur die Informationen ausgegeben werden die das jeweilige OutLevel besitzen. Nur im Karteireiter All werden alle Informationen dargestellt.

11.3 Roadmap

siehe www.as2lib.org/roadmap.php

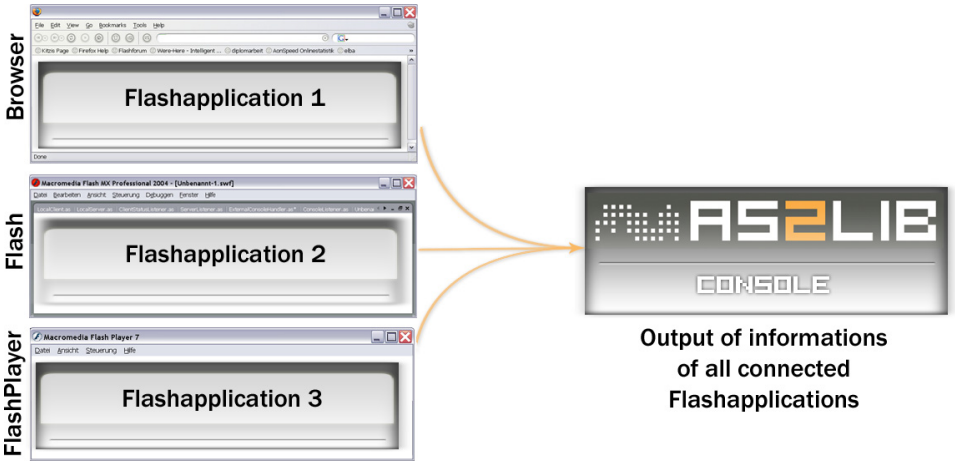

Abbildung 11.1: Verwendung der *as2lib console*

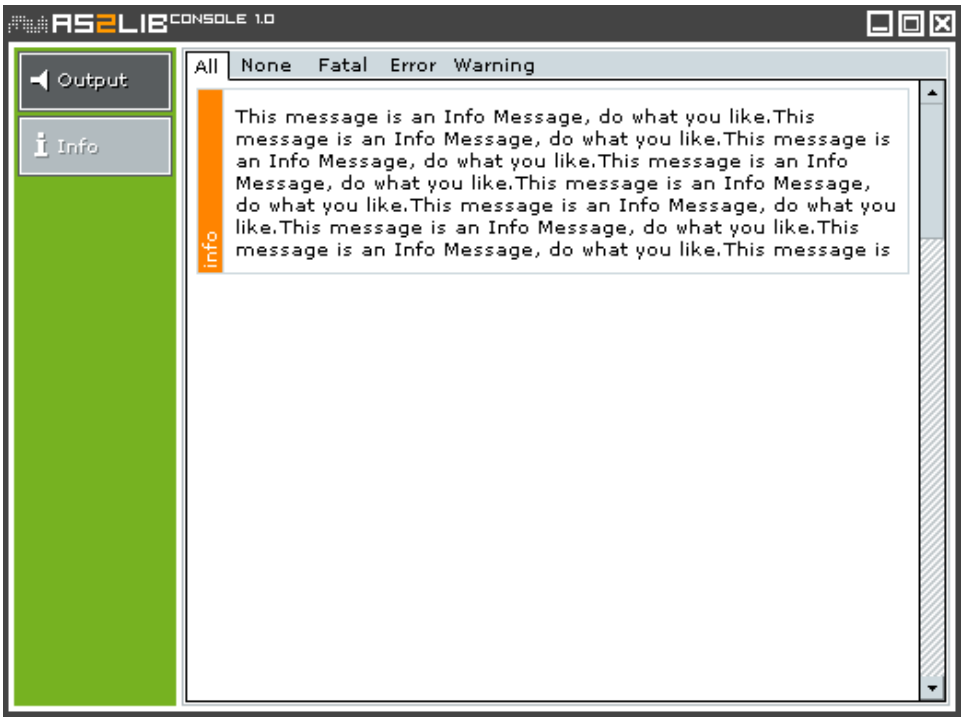

Abbildung 11.2: Grafischer Entwurf der *as2lib* Debug Konsole