Gaffe:
Graphical Front-ends for Z Animation

Nicholas Daley
Masters Thesis
The University of Waikato
New Zealand
Email: ntdaley@acm.org
Supervisor - Dr Mark Utting

2003

Abstract

This thesis describes Gaffe - a software package that provides GUI front-ends for an animator, a graphical builder for designing these front-ends, and automatic generation of graphical front-ends from a Z specification.

The language Z allows specifications to be reexpressed in a way that is unambiguous, and lends itself to proof and testing. Z animators can be used to test the effects of performing particular actions, before the system is built. However, experience is needed in order to understand the meaning of a Z specification, or to understand the output of an animator.

Gaffe will make it easier to test Z specifications, and to demonstrate them to people unfamiliar with Z.

Acknowledgments

Mark Utting	• Supervisor.
Petra Malik	• Author of (CZT subproject) Corejava.
Rachael Hunt and Judy Bowen	• Helped me come to grips with usability testing.

Contents

List of Figures

Chapter 1

Introduction

> Basic research is what I'm doing when I don't know what I'm doing.
> – Werner Von Braun

This thesis describes the creation of the Gaffe package as part of the CZT (Community Z Tools) project. Gaffe is a graphical front-end for a Z animator.

Z is a language for specifying the behaviour of systems (including computer programs). A Z specification describes a system's possible behaviours. This differs from a program, in that it is not necessarily deterministic, and describes results rather than the steps taken to achieve those results. e.g. The result is sorted vs. the sequence of instructions to perform quick sort.

An **animator** is a type of tool for use with Z specifications. It allows one to try a sequence of operations described by the specification. It can be used for testing properties of a Z specification, finding any contradictions in it, and testing it to see if it matches what is wanted. They generally expect input, and display results in Z or something resembling Z. As a result it takes some experience to be able to use them. This limits their usefulness, because people without that experience (for example a client who commissioned the system being created) will not be able to test a specification.

Gaffe has been written in Java. This was done for a number of reasons:

- Java programs will work across different platforms.
- Most other programs in the CZT project are Java based. This is especially important in the cases of Corejava, and the animator engine[1]; because Gaffe makes use of both of these.
- Java's ability to load classes at run-time provides flexibility useful for creating interfaces. If Gaffe had been made in C or C++, then any new

[1]To distinguish between the Z animator Gaffe attaches to, and the part of Gaffe that runs as its front-end, the Z animator will be referred to from now on as the 'animator engine', and the Gaffe front-end is the 'Gaffe animator'.

classes or code that appeared in a Gaffe interface would either have to be compiled into the Gaffe animator, or be imported in a dynamic library. While it would be possible to do this, it would be much less pleasant, and much harder to make cross-platform.

- JavaBeans are useful when dealing with arbitrary objects, as is necessary to provide flexibility in the interfaces built with Gaffe.

1.1 Goals

- To make it possible to interact with a Z animator using a GUI front-end for a Z animator; enabling users with little or no Z knowledge to use an animator, easing the tasks of debugging specifications, and ensuring that they meet client expectations.
- To make the GUI front-end customisable with little or no programming.
- To make immediately obtainable a default GUI front-end suitable for use with a specific Z specification.
- To be cross-platform.
- Open source, so that others may extend it. For example, to animate other specification languages.

Chapter 2

Related Work

2.1 CZT

The Community Z Tools project (Initially proposed[1] by Andrew Martin), is a project to build a framework and a core set of tools for operating on Z specifications. It now has a web site (http://czt.sourceforge.net/) and CVS repository at Sourceforge.

Other sub-projects in CZT include:

- ZML[11] - an XML format for Z specifications.
- Corejava - Annotated Syntax Trees mirroring the structure of ZML.
- Parser - a parser from other formats of Z.

2.2 Possum

Possum[13] is a Z animator, that can be used graphically. GUI front-ends can be written for it in Tk. These front-ends interact with Possum via call-back functions, called, for example, when a Z variable changes. Values and scripts for the animator are passed back and forth using strings.

Because Possum front-ends are written in Tk, it requires a certain degree of programming skill to create a front-end. Whereas, this should not be as necessary with Gaffe. By communicating with its front-ends via strings, Possum gains some flexibility, but adds to the front-ends the work of parsing these values.

Other differences from Gaffe include:

- Possum is an animator, not just a front-end,

- and Possum can read multiple file formats of Z including the LaTeXand email formats; whereas Gaffe only reads the XML format[1].

2.3 Other Animators

VDMTools[2], a package for working with the VDM specification language, and B-Toolkit[3] for the B language both include GUI based animators. These programs handle input and display of results as text, and do not provide the level of graphical interaction allowed by Gaffe and Possum.

Jaza[4] is a text based Z animator written in Haskell by Mark Utting. He will be writing a new animator in Java for which Gaffe will be the front-end.

2.4 Bean Builder

Bean Builder, made by Sun Microsystems, is a demonstration application builder for JavaBeans.

Using Bean Builder or a similar product instead of writing the Gaffe designer, was briefly considered. However it is only intended as a demonstration of JavaBeans and other technologies; and suffers some annoying limitations, for example in version 0.1, there is no obvious way to put a component inside a panel! Additionally there are some parts of the interface enforced by the Gaffe designer, that would not be enforced by Bean Builder (e.g. The use of the `Form` class, and `BeanContexts`). Without the enforced use of `BeanContexts` (see Section 4.4), beans would not have access to services such as the scripting engine. Though Bean Builder could have been modified, uncertainty over its license made it safer to create a new interface builder.

2.5 Other Interface Builders

GUI builders are available for many platforms and toolkits. e.g. Visual Studio (MS Windows), Qt Designer (Qt), wxDesigner (wxWindows), Glade (Gnome). All tend to have a toolbar (or similar) of components that can be placed by clicking in the design pane; though some will not allow arbitrary placement of components, and require that all containers have associated layouts (e.g.

[1]The CZT project includes utilities to convert between the XML format and the others, so this isn't particularly limiting

[2]Made by IFAD. `http://www.ifad.dk/Products/vdmtools.htm`

[3]Made by B-Core Ltd. `http://www.b-core.com/btoolkit.html`

[4]http://www.cs.waikato.ac.nz/ marku/jaza/

components might be forced into a table-like layout). Several also display a tree hierarchy of the components in the interface (a feature not yet present in Gaffe). Some are restricted to designing the user interface (wxDesigner, Visual Studio). Others like Bean Builder (and, in a less graphical way, Qt Designer, and Glade) mentioned above are application builders, capable of designing behaviour as well as looks, by registering handler objects or methods to respond to events. The mentioned GUI builders could not be used for Gaffe, because they are all made for use with C or C++, not Java, and so would not be as easily used on different platforms, and would not connect easily with other CZT software.

Chapter 3

Gaffe Architecture

3.1 Gaffe Interfaces

Part of the plan for Gaffe interfaces was that they would look much like any application. Because of this, interfaces need to be capable of being quite complex, with components of many types potentially arranged into panels and sub-panels. It was also desired that designers of Gaffe interfaces should not need to write significant amounts of program code to achieve their goals. Thirdly, Gaffe should place as few restrictions as possible on how an interface appears; e.g. whether it appears all in one window or many, whether operations share input space or have their own input forms, etc.

So there are the following goals for interfaces:

1. Interfaces should be capable of having complex arrangements.
2. Many types of visual components should be usable.
3. Interface designers should not have to write much code.
4. Gaffe should place minimal restrictions on the shape of an interface.
5. and to be useful, interfaces must be savable.

Early designs for Gaffe had visual input and output components being instances of special plug-in classes[1]. The idea being that these classes would know how to get the information they needed from, or feed the entered information to the animator engine. This approach however had a major disadvantage, in that a component that was to act differently, or appear differently would mean the creation of a whole new plug-in class; violating goal 3 above and hindering the others, especially goal 2.

[1] See Appendix B.

Fortunately, a better solution was found to the issue of how interfaces were to be structured... JavaBeans[4] (See Chapter 4). JavaBeans is a design pattern for allowing both component and non-component objects to be connected together to interact and react to events triggered by each other. The JavaBeans specification was created with visual interface builders in mind; and not *just* for designing the interface, but also for designing behaviour! Also, as a handy benefit of using JavaBeans, since all of Java's windowing libraries are bean based, we get to use all of their standard components for free.

So, JavaBeans are well suited to Gaffe's purposes. Goal 1 is not affected by using JavaBeans, and is mostly an issue for the designer program. Because Java's libraries use beans, goal 2 is dealt with. Assuming a good library of beans is available, the amount of new code should be minimal. Because the designer can put together an interface from the elemental components, there are almost no restrictions on what the constructed interfaces look like; the only thing affecting this is the underlying control code.

Since Java version 1.4 the `java.beans` library has included classes for handling long-term persistence of JavaBeans (see Section 4.3); this handles the issue of saving interfaces.

How, then, is the control code gluing the interface to the animator engine kept flexible enough that it does not restrict the design? One option would be to use beans that when activated take required actions; however different behaviour would mean new beans to be written.

Though using beans in this way is still possible in Gaffe's, there is a simpler solution which does not require users to write new beans. Instead, the Gaffe animator makes use of an ECMAScript[5] (better known as JavaScript) engine[2]. One of Gaffe's core set of beans is a Script bean; when triggered by an event from another bean, it will run its associated script. For the hopefully small amount of glue code required, this should also be easier than compiling new beans to achieve the same task.

Thanks to JavaBeans, the designer can make interfaces arbitrarily complex, with any appearance desired. Thanks to ECMAScript, the designer can make interface *behaviour* arbitrarily complex, with any behaviours desired.

[2]Specifically the BSF (Bean Scripting Framework) from Apache, which provides a uniform interface for scripting with multiple languages, and uses Rhino from Mozilla to implement ECMAScript.

3.2 Animation

The way the Gaffe animator code handles interfaces has been dealt with above. Other than this, the main decisions with animation were:

1. How to manage the history of animation states.
2. How beans get access to the history, and other global resources (like the scripting engine).
3. What portion of the Gaffe animation code interacts with the animator engine.

Because one of the early goals for Gaffe was that it be very customisable, it was decided that the history object should be swappable (so, for example, it could be replaced with one that kept a tree-like history, that kept a branch for each operation tried from each point in the history). So the basic code interface for all histories has a method for retrieving the current point in the history, but no methods for going back and forth. Because JavaBeans, and ECMAScript are being used, this is not a problem (ECMAScript doesn't keep track of types, and will give access to any members/methods available).

To manage each form's collection of JavaBeans, and the 'services' (e.g. history, scripting engine) available to them, Bean Contexts were used. A bean context is essentially just a collection of beans, and possibly other contexts. A `BeanContextServices` object is a bean context that can have 'services' registered with it, and makes these available to its children. So for example the Script bean gets access to the script engine via the `BeanContextServices` that contains it.

Because all parts of a Gaffe interface that display information from the animator engine are dealing with state information that has already been entered into the history, the connection to the animator engine is through the history. This can seem strange when entering input data into the history, but was deemed worth it because it restricts the connection to the animator engine to one class.

3.3 Gaffe Interface Design

Issues for interface design code are:

1. Bean properties must be editable.
2. An interface being designed should look as similar as possible to its appearance in the Gaffe animator.

3. Without recompiling the designer, it must be able to use new types of beans in a created interface.

The `java.beans` library provides a mechanism for finding the appropriate editor for a property. `PropertyEditors` (these are registered using `PropertyEditorManager`) describe enough information to edit a property. If necessary, they can provide a component to be used as the editor. By using these editors the designer can allow editing of any type of property.

JavaBeans are meant for use at design time as well as run time. So, the panel that makes up the form is placed in the design window with very few special measures taken. In order to stop it from being manipulated a 'glass pane' is placed above it to trap user input (see Section 8.1).

Because the BSF and Rhino libraries were already required for Gaffe, it was an easy call to configure the designer using an ECMAScript script. This way to change the configuration of the designer, the script can be changed, and the program does not have to be recompiled.

3.4 Gaffe Interface Generation

Decisions for interface generation include:

1. How to make different portions of the generation code swappable, so that different styles of interface may be generated.

2. How to provide uniform command-line behaviour despite the different needs of swappable parts of the generation code.

Solving both of these, a plugin architecture was developed. With command-line processing separated into a separate class. This plugin architecture is described in Chapter 7.

Chapter 4

Introduction to JavaBeans

4.1 What are JavaBeans?

JavaBeans[4] is a design pattern that allows compliant objects ('beans') to be connected together, for example, by application builder programs. By following certain rules[1] on how internal state is exposed, and how other objects can interact with it, a bean allows itself to be connected to other beans without the specifics of the bean interfering.

1. A bean must provide a constructor that takes no arguments. This makes creation easy for programs that wish to use the bean.

2. A bean should, rather than having public variables, expose its state ('properties') via getter and setter methods. e.g. A bean class with a property '`point`' of type '`Coordinate`' would have members:

   ```
   public void setPoint(Coordinate p);
   ```

   and:

   ```
   public Coordinate getPoint();
   ```

   This way a property can be made read only, or write only (by having only a getter or a setter), events can be thrown if necessary (e.g. a `PropertyChangeEvent` to notify that the property has changed), and other internal state can be changed to be consistent.

3. A bean should be serializable via Java's object stream classes, so that interfaces containing the bean can be saved.

[1] The java.beans library provides mechanisms to allow objects that don't follow these rules to be used as beans, but doing so requires extra work.

4. If a bean should be able to trigger actions in other objects ('events'), then it should provide methods for registering listeners. When the appropriate conditions happen to trigger such an event, the bean calls the corresponding method on all of the registered listeners of appropriate type, passing an event object. e.g. A bean that fires an event when one of its properties change would have a method for registering the event:

```
public void addPropertyChangeListener
                        (PropertyChangeListener pcl);
```

   and when a property changed it would create a `PropertyChangeEvent` object, and pass it to the appropriate method on all registered `PropertyChangeListeners`:

```
public void propertyChanged(PropertyChangeEvent ev);
```

Names are important here; getter and setter methods *must* start with get and set, property registration methods must follow the form:

`public void add`⟨**listener type**⟩`(`⟨**listener type**⟩`)`

Events must have names corresponding to their listener types. So an event `FooEvent` would get delivered by a bean to listeners of type `FooListener`, which would be registered with the bean's method:

```
public void addFooListener(FooListener)
```

```
class MyBean
{
    ⋮
  public void addFooListener(FooListener listener)
    ⋮
};
interface FooListener extends EventListener
{
  //This method could have any name.
  public void eventMethod(FooEvent ev);
};
```

Some of these rules should look familiar to people who have experience writing GUI applications in Java, because the classes in Java's Swing and AWT libraries are written as beans.

4.2 Introspection

`java.beans.Introspector` allows the `BeanInfo` object for a class to be obtained; this can be done with any type even non-beans. A `BeanInfo` object describes all of the useful information for an application builder's use of a bean:

- Names and descriptions suitable for showing to the user of an application builder.
- Methods involved; e.g. the getter and setter methods for properties.
- Types involved; e.g. the class of a property.
- Attributes for adding extra information about classes, properties, etc. e.g. The transient attribute mentioned in section 8.3.

It gives information for properties, event sets (i.e. which classes of listener can be registered with it), methods, and the bean itself. `BeanInfo`'s can be overridden allowing objects that don't quite meet the rules in the previous section to be used as a bean, or allowing extra information to be added (e.g. an icon to represent the class in application builders).

4.3 Persistence

Although, according to the JavaBeans specification, beans should be serializable via ObjectStreams, this has gone out of vogue for long term storage. The `java.beans` library now provides `Encoder` (also its concrete implementation `XMLEncoder`, and the corresponding `XMLDecoder`) which is used for saving a connected set of beans in terms of the calls necessary to make them via their public interfaces (the methods and properties described by a `BeanInfo`). This way, it doesn't matter if a class is changed to store its internal state differently, as long as it doesn't change how it exposes this state.

As with introspection, it is possible to change how `Encoder`s deal with a bean. A `PersistenceDelegate` class can be written for a bean class to change the way it is saved. This allows objects that don't quite fit the rules for beans to be saved.

An example encoded file ...

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2" class="java.beans.XMLDecoder">
 <object class="EncoderExample$A">
  <void property="integer">
   <int>3</int>
  </void>
  <void property="object">
   <object id="EncoderExample$B0" class="EncoderExample$B">
    <void property="string">
     <string>Hello world</string>
    </void>
   </object>
  </void>
 </object>
 <object idref="EncoderExample$B0"/>
</java>
```

... and the program that created it

```
import java.beans.XMLEncoder;
public class EncoderExample {
  public static void main(String[] args) {
    B b=new B();
    b.setString("Hello world");
    A a=new A();
    a.setInteger(3);
    a.setObject(b);

    XMLEncoder encoder=new XMLEncoder(System.out);
    encoder.writeObject(a);
    encoder.writeObject(b);
    encoder.close();
  };
  static public class A {
    private int integer;
    private B object;
    public int getInteger() {return integer;};
    public void setInteger(int i) {integer=i;};
    public B getObject() {return object;};
    public void setObject(B o) {object=o;};
  };
  static public class B {
    private String string;
    public String getString() {return string;};
    public void setString(String s) {string=s;};
  };
};
```

4.4 Bean Contexts

In order to manage groups of beans, `java.beans.beancontext` provides the class `BeanContext`. `BeanContexts` act as hierarchical containers, which can contain beans, other bean contexts, or any other type of Java object. If a class instance needs to be aware of the `BeanContext` that contains it, then the class can implement the `BeanContextChild` interface; in which case its `beanContext` property will be set when it is added to or removed from a context. `BeanContextMembershipListeners` can be used to track the contents of a context, but this feature is not used in Gaffe.

The next step beyond `BeanContexts` are `BeanContextServices` which, as well as acting as a collection of beans, can also provide services to those beans. In this context a service is an object that can be requested using its class as a key. The example used in the Java^{TM} Tutorial's JavaBeans^{TM} Trail[12] is of a service for counting the number of words in a document; where the document is represented by a bean in the `BeanContextServices`.

A `BeanContextServicesProvider` can be registered with a `BeanContextServices`; it has methods for finding what services it provides, and for acquiring and releasing service objects. If a `BeanContextServices` does not have any service providers that cover a requested service, it will pass the request up to its parent (assuming it has one); as a result different services can be made available to a larger or smaller group of beans.

The Gaffe animator maintains a root `BeanContextServices` providing the scripting engine, and the animation history as services. Below this are `BeanContextServices` for each `Form`, and below these are the beans contained in each of these forms. Through these services, beans can run scripts, or manipulate the history.

Chapter 5

Introduction to BSF, Rhino, and ECMAScript

ECMAScript started life as JavaScript (invented by Brendan Eich), built into Netscape Navigator. It was first adopted as a standard[5] by the industry association 'Ecma'[1] in 1997. The following year it was adopted as an ISO/IEC standard[6]. Though their names are often used interchangably, 'ECMAScript' is the name of the language, 'JavaScript' is the name of Netscape's implementation which adds the objects, methods, etc. used to allow scripts and browser to interact, and 'JScript' is Microsoft's version in Internet Explorer.

Rhino[2] is Mozilla's Java implementation of ECMAScript. Gaffe uses another library that sits on top of this; BSF[3] (Bean Scripting Framework) is an extensible framework for scripting JavaBeans using different languages. It uses Rhino as its back-end for interpreting ECMAScript.

Gaffe uses BSF and Rhino to incorporate scripting into its graphical interfaces. Though only ECMAScript has been used, because BSF can handle many languages and can be extended with more, there is nothing stopping other scripting languages being used in one of Gaffe's interfaces.

When using BSF, most if not all interaction goes through objects of type `BSFManager`. A `BSFManager` keeps track of a collection of scripting engines, and of a collection of beans 'declared' with it (so that they can be used by name in a script, e.g. in Gaffe a script can access the history object through the name `History`), and provides methods for evaluating and compiling scripts, and for calling methods in a script.

For a scripting language to be usable in BSF, an implementation of a

[1] http://www.ecma-international.org/

[2] http://www.mozilla.org/rhino/

[3] Previously from IBM, BSF has been taken over by the Apache Software Foundation. It can be found at http://jakarta.apache.org/bsf/

`BSFEngine` must be available, and registered with the `BSFManager`. So for many of the engines that come with BSF[4], all that would be needed to run scripts in their language is to install the libraries they depend on. For engines not included with BSF, it may be necessary to add calls in Gaffe's start-up code to `BSFManager.registerScriptingEngine(...)` registering them.

Scripts can be evaluated giving a result with `BSFManager.eval(...)`, executed ignoring any result with `BSFManager.exec(...)`, or treated as an anonymous function taking arguments with `BSFManager.apply(...)`. Unfortunately the default behaviour, adopted by the engine for ECMAScript is to ignore the arguments in an apply. This caused difficulties, because this would have been an ideal way to feed data into scripts; but a solution (described in Section 8.4) was found. Any errors parsing or executing a script cause a `BSFException` to be thrown, allowing the host program to recover.

Because Gaffe uses JavaBeans it makes a lot of sense to use BSF for scripting, as it is made to operate on JavaBeans. For example in ECMAScript, any methods covered by the `BeanInfo` of a bean that is available to a script, can be used by that script; and all of the bean's properties can be treated as member variables of the bean.

For example the following is a script that closes the current window, and writes a message to standard error (note `thisForm` is one of the things that ideally would be passed into the script using `apply(...)`):

```
thisForm.setVisible(false);
java.lang.System.err.println(``Window '' + thisForm.name +
                             `` closed.'');
```

[4]Including engines for Python, and XSLT.

Chapter 6

Design

Gaffe is split into three applications:

- The Gaffe Designer: a graphical builder for creating Gaffe interfaces manually.
- The Gaffe Animator: connects to the animator engine, using it to execute operations in the Z specification.
- The Gaffe Generator: takes a Z specification as input, and creates a Gaffe interface suitable for use animating the specification.

The Gaffe generator is separate from the others, because it need not be used via a GUI; and at present is only used via the command-line. The Gaffe animator is separate from the Gaffe designer, because the designer is quite complex already, without worrying about side-effects that animation might have on an interface. This also allows a client to be given access to the Gaffe animator, without any possible confusion that access to the designer might cause.

The code for the Gaffe animator and designer is highly interdependent, as both must be able to load, manipulate, and display Gaffe interfaces. The generator on the other hand depends on code from the others for manipulating and saving interfaces, but need not be installed for the Gaffe animator and designer to work.

This chapter deals only with the Gaffe animator, and the designer. The generator is described in the next chapter.

6.1 File Format

Although it has already been specified that the file is saved using `XMLEncoder`, there is still the issue of what gets saved, and in what order.

`XMLEncoder` and `XMLDecoder` can have an 'owner' object attached. This object can be of any type, and has its properties saved one at a time by the program, rather than automatically by the `XMLEncoder`. The owner can be used to save properties global to the file. Gaffe uses the owner object to store the history object (and as its properties the name of the state and initialisation schema), the initialisation script (for setting up global variables, defining functions, etc.), and the URL for the Z specification to use.

After the owner's properties are saved, three objects are saved for each form in turn; the `Form` object, a vector of `BeanWrapper` objects (used to represent the location in the designer of non-visual beans), and a vector of `BeanLink` objects (to represent the connections between event listeners, and the beans they are registered with).

So the file might look something like this[1]:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2" class="java.beans.XMLDecoder">
 <!-- The owner contains settings for the whole interface. -->
 <void property="owner">
  <void property="history">
   <object class="net.sourceforge.czt.animation.tmp.BBHist">
    <void property="initSchema">
     <string>InitBirthdayBook</string>
    </void>
    <void property="stateSchema">
     <string>BirthdayBook</string>
    </void>
   </object>
  </void>
  <void property="initScript">
     ⋮
  </void>
  <void property="initScriptLanguage">
   <string>javascript</string>
  </void>
  <void property="specificationURL">
   <string>file:/research/birthdaybook_unfolded.xml</string>
  </void>
 </void>
 <!-- The first Form in the interface.  All of this form's -->
 <!-- beans get saved in this element with the calls that -->
 <!-- add them to the form. -->
 <object class="net.sourceforge.czt.animation.gui.Form">
```

[1]Knowing the file format is mostly useful for directly editing the file, to test how the Gaffe animator would react to something, before making corresponding changes in the designer or generator. For example testing animation requires a custom `History`, but the designer does not yet allow a history object to be selected.

```
    ⋮
 </object>
 <!-- The first Form's bean wrappers. These save the -->
 <!-- location of the visual representations of non-visual -->
 <!-- beans in the designer. -->
 <object class="java.util.Vector">
  <void method="add">
   <object
     class="net.sourceforge.czt.animation.gui.design.BeanWrapper">
      ⋮
   </object>
  </void>
    ⋮
 </object>
 <!-- The first Form's bean links. All of the event links -->
 <!-- between beans in the first Form are saved here. -->
 <object class="java.util.Vector">
  <void method="add">
   <object class="net.sourceforge.czt.animation.gui.design.BeanLink">
      ⋮
   </object>
  </void>
    ⋮
 </object>
   ⋮
</java>
```

6.2 Animator

Java Package: `net.sourceforge.czt.animation.gui`

The Gaffe animator can be considered in two parts - the portion that manages history, connects with the animator engine, and represents the data exchanged with the animator engine; and the portion that deals with the user interface. The next two subsections describe the design of each of these parts, and the reasons behind some of the design decisions.

6.2.1 The Processing Portion

This section describes the first of the two portions (shown in Figure 6.1).

Data coming from the animator engine can come in five types: the three complex types - `ZSet` which covers all set variables in Z including relations and functions, `ZTuple` which covers all tuples, and `ZBinding` which covers variables whose type is a schema; and the two simple types - `ZNumber` which represents any integer, and `ZGiven` which represents other simple values like

given types and free types[2]. All of these types are immutable, providing read-only accessor methods to retrieve the value inside; they all also implement Java's `toString()` and `equals(...)` methods, and inherit from the `ZValue` interface[3]. One type not implemented at present are recursive free types e.g.

$$Foo ::= foo\langle\langle S \rangle\rangle$$

Its implementation would only affect the rest of Gaffe if locators (described at the end of this section) for this type were going to be implemented too. If a `ZFreeLocator` was going to be implemented, then it would only affect `ZLocator`'s `fromString(String)` method, and some small ECMAScript methods.

`SolutionSet` represents the set of solutions resulting from an operation. Each solution is represented inside it as a `ZBinding` (because both bindings and solutions are mappings from variable names to values).

Eventually `SolutionSet` and the types will change to be abstract types, so that the animator engine can, for example, choose an implementation of `SolutionSet` that uses lazy evaluation, improving response time by waiting until a solution is asked for before evaluating it.

The `History` object is responsible for passing operation requests on to the animator engine, getting a `SolutionSet` in return, and making the contained results available to the rest of the program.

`SolutionSet` contains an ordered set of solutions, and keeps track of a currently viewed solution. The only way to get access to the solutions inside is via a `getCurrentSolution()` method; in order to access the other solutions `nextSolution()` and `previousSolution()` must be used to step through it. Similar to one of Java's standard `Iterator`s, methods are provided to check if there is a next/previous solution; however unlike `Iterator`s, the position in a `SolutionSet` points at a solution, not between two solutions. The current position points at rather than between solutions, because multiple beans may need to access the current solution; and if it behaved like Java's `Iterator`s instead, then the current solution would only be available to the bean calling `nextSolution()` or `previousSolution()`. To allow beans to discover when the current solution is changed, `PropertyChangeListener`s[4] can be registered with a `SolutionSet` on the `currentSolution` property, although normally beans would do this through the `History` object instead.

[2]The value stored in a `ZGiven` is a string to allow for a broad range of uses.

[3]ZValue is just a 'marker' interface, so that values can be passed around without regard to what specific type they are.

[4]See Chapter 4

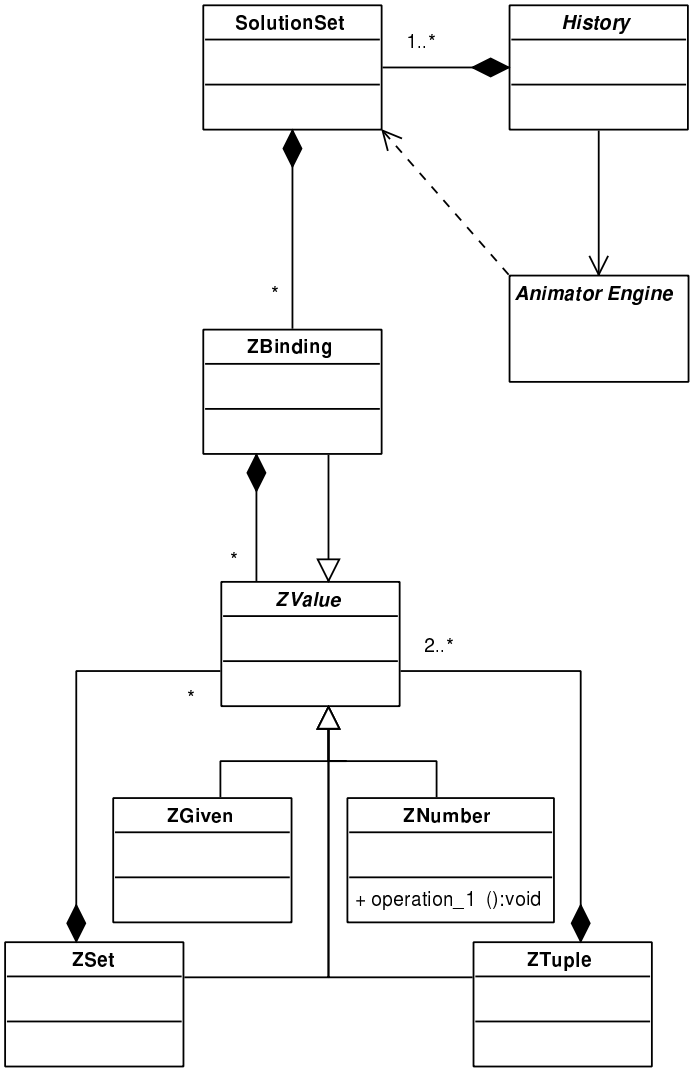

Figure 6.1: UML class diagram of the Processing Portion of the Gaffe Animator.

It should be noted, that because the animator engine is not yet written, testing has been done with custom implementations of `History` that do the work of the animator engine for a particular specification.

Not shown in Figure 6.1 are `ZLocators`. Because part of a user interface will be interested in a certain part of the current solution (e.g. the 'fred' variable in the third binding in the tuple called 'bob'), but a particular solution is not always the *current* solution; `ZLocators` can be used to recursively describe a location independently of the solution involved. At present there are only locators for `ZBindings` and `ZTuples`, because these were deemed most likely to be useful. A bean that wants to display a particular value from the current solution, rather than finding it itself, can pass the current solution into the

`apply(...)` method of a `ZLocator` to retrieve the value it wants.

6.2.2 The User Interface Portion

This section describes the second of the two portions (shown in Figure 6.3).

Most of the work in the user interface portion is done by the beans in the interface loaded from the file (labeled as 'Bean' in the UML diagram in Figure 6.3). Other than this, the most interesting parts are `Form`, `AnimatorCore`, and `History`.

`Form` represents one window in the interface. It also tracks all of its descendant beans using a `BeanContextServices`; to which the `Form` itself is added as a service (See section 4.4), so that beans can easily access the form they belong to and the other children of the form. It provides methods for adding and removing beans (used by the designer, and by the file format[5]), methods for accessing its beans, and for looking them up by their `name` property.

`AnimatorCore` (the main class of the animator) has three responsibilities; loading and setting up the interface, managing global settings and services, and managing the set of `Forms` making up the interface. The code that starts the Gaffe animator finds a file to load either from the command line, or from a file-chooser dialog. It then creates the `AnimatorCore` object which loads the file, hooking up event links between beans, and creating a window to put each form in.

`AnimatorCore` then declares beans with the scripting engine (See chapter 5); though two of these have the same name as a Java class, in ECMAScript they are the sole instances of these classes present during an execution of the Gaffe animator:

- `History` - the `History` object.
- `AnimatorCore` - the `AnimatorCore` object.
- `Forms` - the list collecting all of the `Forms`, with the addition of a method `lookup(...)` for finding a form by its `name` property.

The `AnimatorCore` collects all of the `Forms`' `BeanContextServices` in its own `BeanContextServices` (See Figure 6.2). This context provides the scripting engine, and the `History` object as services; so that beans in the interface can make use of them.

[5] See section 4.3.

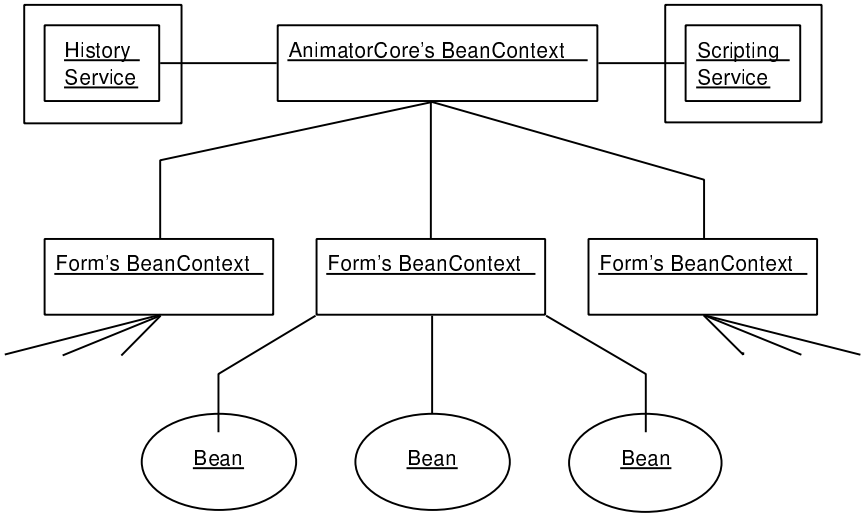

Figure 6.2: The hierarchy of `BeanContext`s.

Once all of this is done, the interface is in a state suitable for being used; at least one form window is open, and the user interacts directly with the interface loaded from the file. After this the only responsibility of the `AnimatorCore` is to exit when all windows have been closed.

In this portion of the Gaffe animator `History`, has two responsibilities. One is to present the data from the animator engine in a structured, navigable way; the other is to receive inputs to pass on to the animator engine.

To give access to data from the animator engine, `History` provides accessor methods to get the current solution, and the current solution set. It also reproduces via delegation, the methods for navigating through the `SolutionSet`. However methods for navigating through the history only appear in implementations of the `History` interface. Contrary to what users of strongly typed languages might expect, this does not present any problems, because ECMAScript variables are untyped, and will give access to all of their methods. So that beans know when the current solution has changed, `History` repeats `SolutionSet`'s provision for notifying `PropertyChangeListener`s when `currentSolution` has changed.

To enable input and schema activation, `History` provides a property called '`inputs`' containing a mapping from `ZLocator` to `ZValue`. To ease the task of scripts in the interface, a number of variations of a method '`addInput(...)`' are given. Once an interface has filled the `inputs` property with appropriate values, it will call `activateSchema(...)` to run a given schema on those inputs. Gaffe does not check that inputs are correctly formatted, or all present;

if there is a problem with the input, then the animator engine will throw an exception.

The classes `BSFServiceProvider` and `HistoryServiceProvider` are used with `AnimatorCore`'s bean context, to register the scripting engine and the history as services. Another class `FormServiceProvider`, which does not appear on the UML diagram below, registers each `Form` as a service to its beans.

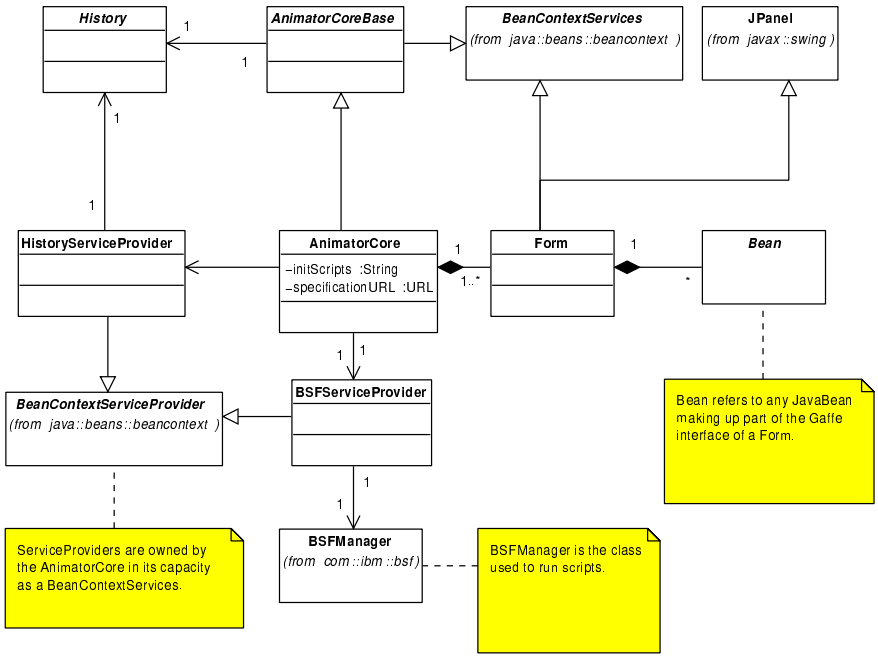

Figure 6.3: UML class diagram of the User Interface Portion of the Gaffe Animator.

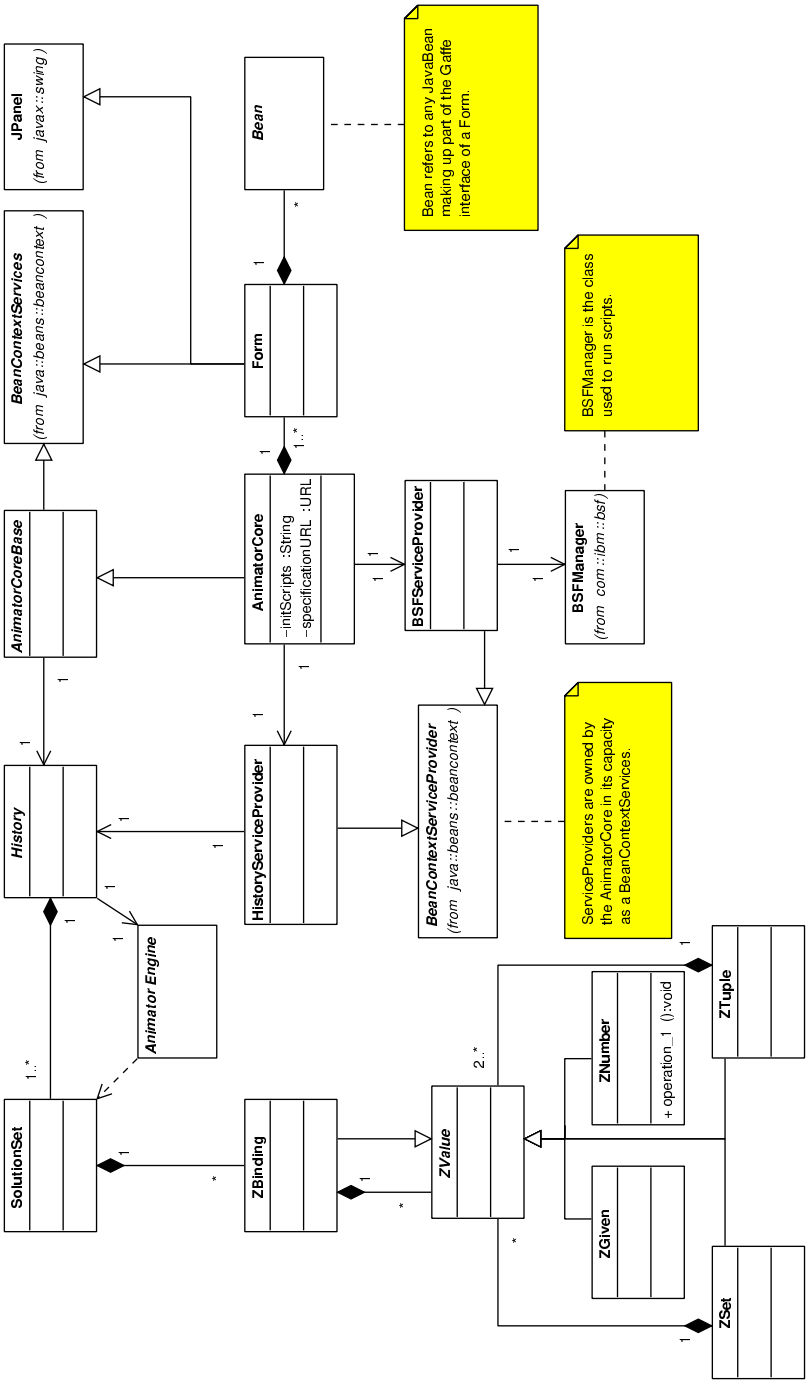

Figure 6.4: UML class diagram of the whole of the Gaffe Animator.

6.3 Designer

Java Package: `net.sourceforge.czt.animation.gui.design`
Other than those related to tools, and the properties window, the important classes in the designer (shown in Figure 6.5) are `DesignerCore` and `FormDesign`.

`DesignerCore` tracks all windows, the majority of keyboard shortcuts, and user actions (including loading and saving, creating and deleting forms, etc.) When it is started, the `DesignerCore` runs the configuration script, sets up the tool and properties windows, and either loads an interface or starts a new one with a single `FormDesign`. The configuration script is located in the .jar library containing Gaffe; this script can be overridden, or supplemented on a system-wide, or a per-user basis using Java's `java.util.prefs` library. It includes calls to register `PersistenceDelegates` (See Section 4.3), `PropertyEditors` (See Section 6.3.2), and `Tools` (See Section 6.3.1).

Each `FormDesign` (See Figure 6.6) manages a window for designing the interface related to one `Form`. Except for its menu and status bar, the `FormDesign` is made up of two layers; the first is a 'bean pane' which contains the `Form` and `BeanWrappers` representing all of the non-visual beans, and the second is a 'glass pane' on which are placed handles for manipulating the currently selected bean, and highlighting is drawn to indicate the event links between beans.

The glass pane also traps any user input events, such as mouse clicks. This prevents the Gaffe interface from being used at design-time, which could cause unwanted side-effects. A custom traversal policy is also applied to stop components on the bean pane from being accessed via the keyboard.

`FormDesign` provides methods for use by the event link tools for adding and removing event links. It also provides methods for use by the bean creation and deletion tools for adding or removing a bean; visual beans (those that are instances of `Component`) will not be allowed outside the `Form`, non-visual beans will not be allowed inside it. Finally it provides methods to deal with converting a location on the glass pane to a location in one of the bean pane's descendants, and for dealing with the related complexities of nested panels.

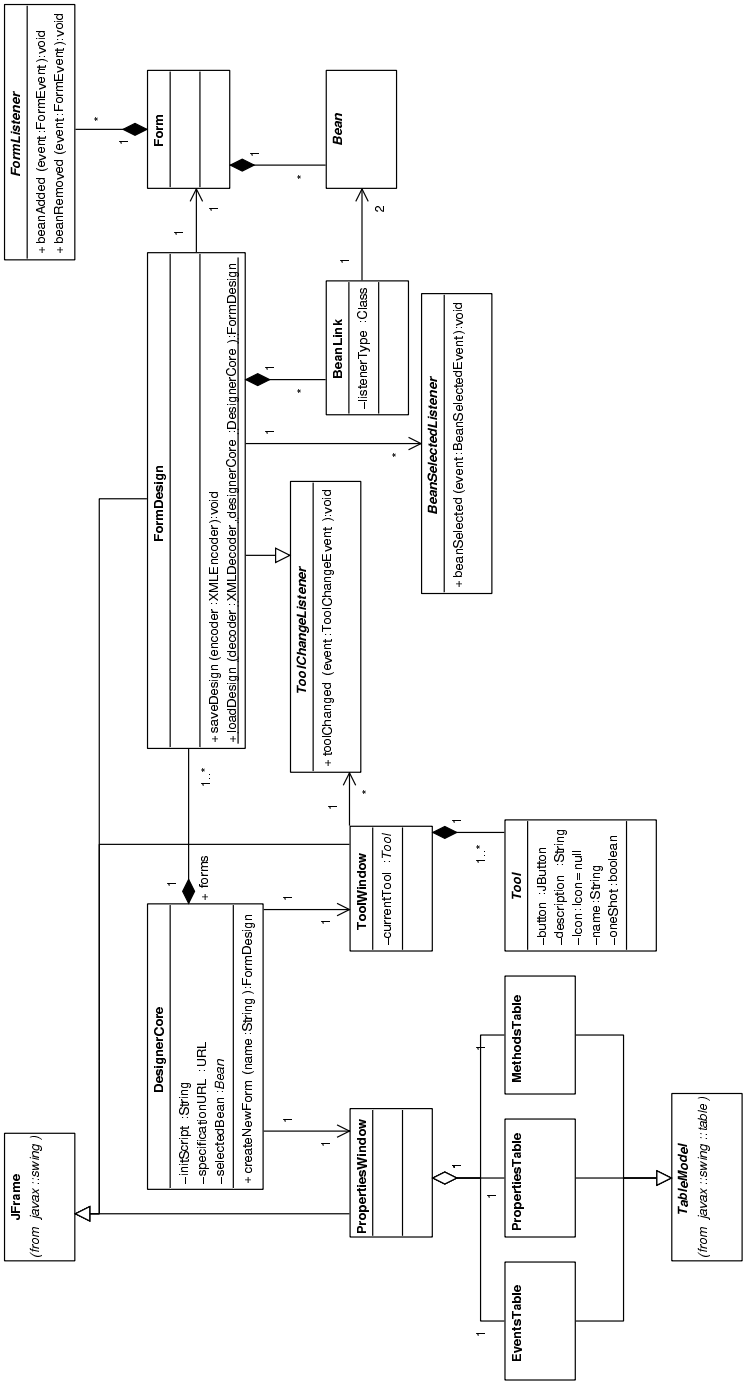

Figure 6.5: UML class diagram of the Designer.

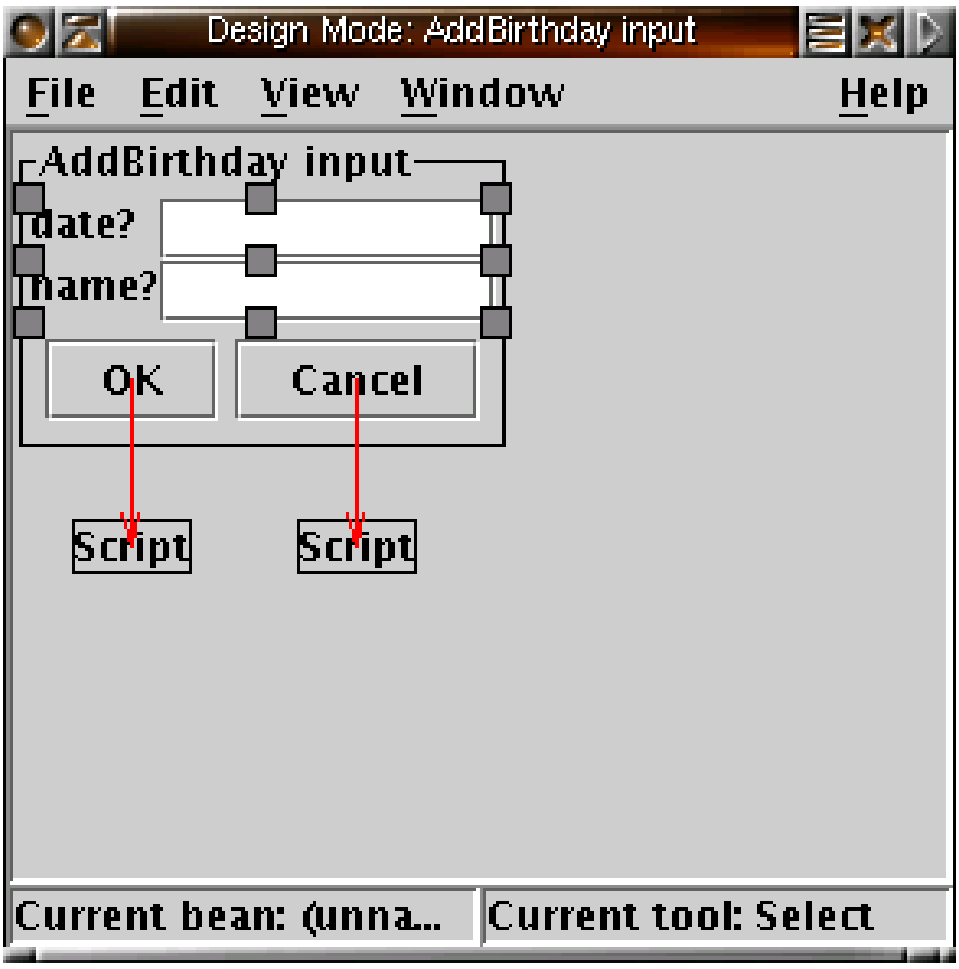

Figure 6.6: BirthdayBook's Add input form being edited.

6.3.1 Tools

Every tool provides:

- (optionally) an icon to display in its button in the tool window.
- A name to display if there is no icon.
- A short description to appear in the button's tool-tip.
- A flag indicating whether it's an instant operation (e.g. the delete tool deletes the current bean), or requires some further interaction (e.g. the create bean tools need to know where to place the bean).

At present there are four special tools; the selection tool (which is reselected after any other tool has finished its task), the delete bean tool, the make event

link tool, and the delete event link tool. All other tools are for creating different types of bean (The types used are set in the designer's configuration script).

`Tool` has methods that are called when the tool is selected/unselected; also methods that mimic the mouse listener methods, except that they take an extra parameter - a reference to the `FormDesign` the mouse is working on. Whenever the glass pane of the `FormDesign` receives mouse events, it passes them to these mouse methods of the current tool.

The `ToolWindow` is just a window, split into one panel for special tools, and one for bean creation tools. It also manages tracking the current tool. The `FormDesign` knows what the current tool is, because it is registered with the tool window as a `ToolChangeListener`.

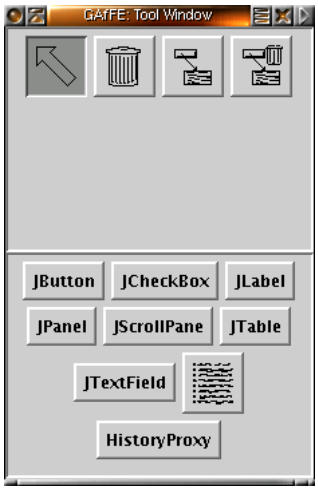

Figure 6.7: The Tools Window.

6.3.2 Properties Window

The largest part of the properties window is the tabbed pane giving access to three tables; the properties table, the events table, and the methods table. The events table displays all listeners registered with the current bean, the methods table displays all of the methods provided by this bean. The more useful table is the properties table.

The properties table gives an entry for each property in the bean. It also allows the properties to be edited by making use of `PropertyEditors` from the `java.beans`' library. A `PropertyEditor` defines how to edit properties of a particular type, it can have methods for translating to and from strings, or it can provide a component that will edit the property. When a property isn't being edited, the component used to render it is set via `TableCellRenderer`.

With this combination, a colour property can appear in the table as a block of the appropriate colour, but when clicked on, it opens a colour chooser panel.

The properties list can be quite long on some beans, so a menu on the properties window allows the displayed properties to be filtered by criteria obtained from the bean's `BeanInfo`. In a `BeanInfo` properties can be marked [6]:

- Hidden - to recommend that the property not be shown in an application builder.
- Expert - to suggest that only expert users edit the property.
- Preferred - to indicate that a property is likely to be useful.
- Transient - to indicate that it won't be saved in the interface.

Additionally, they can be filtered on whether it is possible to edit the property or not (this depends on whether the property has a setter method, and whether a `PropertyEditor` is registered for the property's type).

To allow for new property types, the designer's configuration script can be used to register new `PropertyEditor`s, and `TableCellRenderer`s.

`Customizer`s can also be registered in the configuration script. These are custom GUI components for editing a bean. Clicking the "Customiser" button, in the properties window, will open the customiser for the current bean.

[6] Though these are a standard part of JavaBeans, they do not, as one might hope, come set by default for any of the standard Java classes.

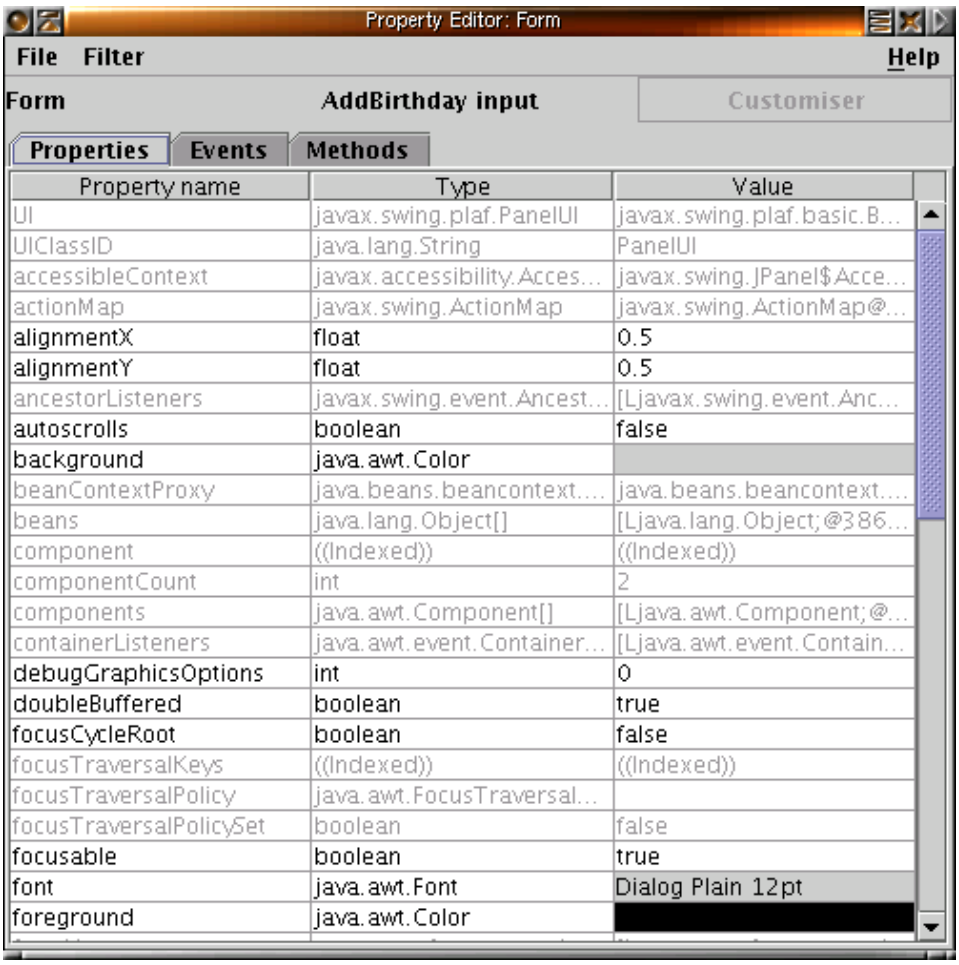

Figure 6.8: The Properties Window.

Chapter 7

Design of the Generator

Java Package: `net.sourceforge.czt.animation.gui.generator`
The process of generating was separated into tasks.

1. Obtaining the Z specification, and parsing it.
2. Finding all of the schemas in the specification.
3. Determining which schemas are relevant, and which fill the roles of state, initialisation, and operations.
4. Identifying the relevant variables in a schema.
5. Identifying the appropriate bean to use to display a variable, or get input for a variable.
6. Generating the interface. (deciding layout of forms, scripts, creating scripts, etc.)
7. Saving the interface.

Because Z has so few restrictions on how a specification can be organised, and because there are so many ways users may want their Gaffe interfaces to be arranged; it was decided that each of these tasks should be replaceable as plug-ins; especially interface generation. Because most of these plug-ins would need options to be set by the user (e.g. via command line), the bulk of the work processing these options was separated out. All tasks inherit from the interface type `Plugin`, which provides method headers for obtaining help text, and a list of handled options. Each task has an interface type (which inherits from `Plugin`), that provides the method headers used by the main program to do the actual work; this interface type also acts as a key to identify that task.

The main program only accesses the plug-ins through a `PluginList`, which keeps track of plug-ins, instantiates plug-in implementations, and handles most of the option processing.

After option processing, the main program goes through each plug-in in turn, feeding it the data it may need from previous plug-ins, and extracting the data later plug-ins may need.

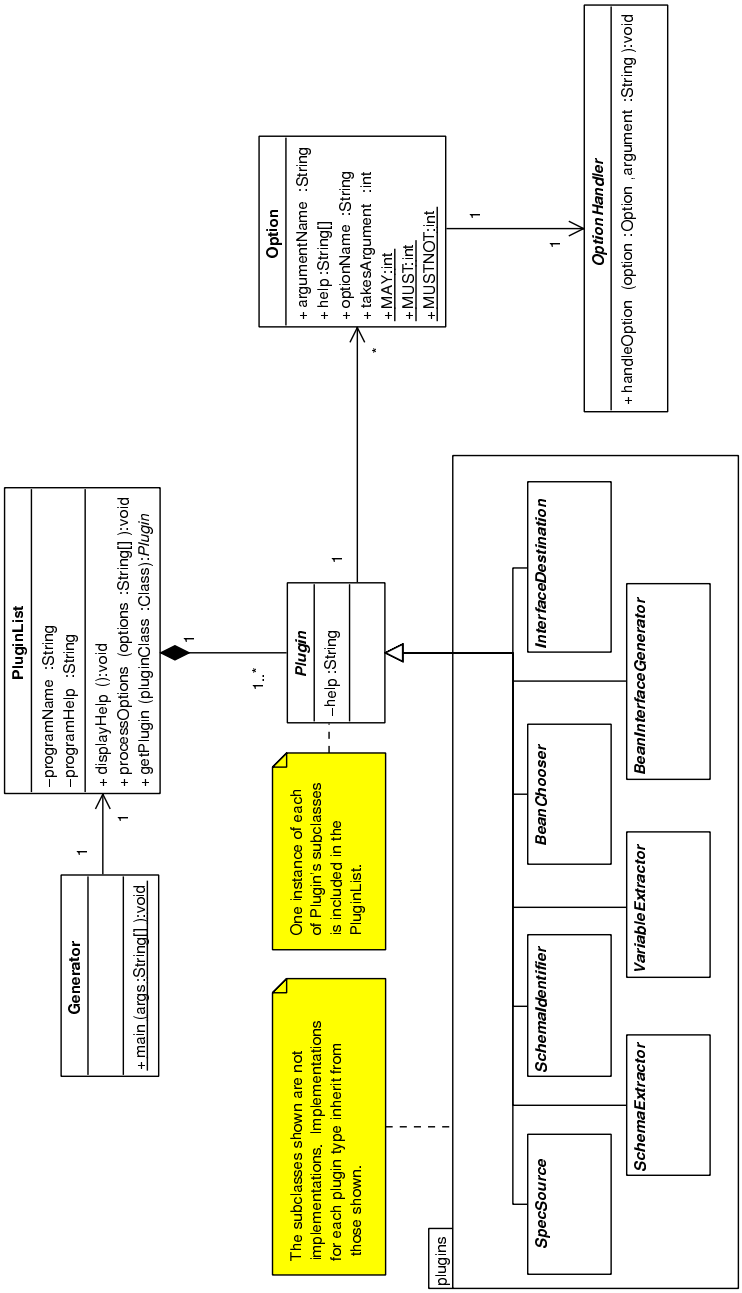

Figure 7.1: UML class diagram of the Generator.

7.1 Option Processing

At present the generator's options can only come from the command line. However there is nothing preventing this coming from a file, and it would be a simple matter for it to come from a GUI without any changes being needed to plug-ins.

Every plug-in must be able to provide a list of `Option`s that it can handle. Each `Option` object contains the relevant data for one option:

- a name suitable for use on the command line (e.g. "help" for the '-help' option),
- help text,
- whether it can, may, or must take an argument,
- the argument's name suitable for use in help text (e.g. "plugin name" for the help option, which can be told to give help for only one plugin),
- and a reference to an `OptionHandler`.

An `OptionHandler` is to an `Option` what a listener is to an event; it performs whatever action is required by this `Option`; normally this would involve configuring a plug-in. By separating this from the `Option`, it becomes possible to have two options which do the same thing (i.e. one is an alias for the other).

When created, `PluginList` takes an array of interface types, an array of their default implementation types, and some information used in help text. Using a private plug-in it can allow a user (at run time) to select a different implementation to use instead of the default[1]. Another private plug-in handles display of help text. `PluginList` will go through each string in its input, decide which are option names, which are arguments, which `Option` object they map to, and run the corresponding `OptionHandler`.

Any strings in the array given to `PluginList` which starts with a '-' is identified as an option. If the option can take an argument, and the next string in the array doesn't start with a '-', then it is the argument for this option. Arguments that don't get associated with an option, become associated with the `null` option, which can be handled by plug-ins the same as any other option; this way plug-ins can handle arguments not associated with an option, as is the case for filenames on most command-line programs. The correct option to use is found by sorting through all of the plug-ins starting with

[1]Because of this, the main program accesses the plug-ins through the PluginList, using its interface type as a key. Also because a plug-in needs to be instantiated to handle its options, the plug-in selector's options must appear first on the command line.

PluginList's private plug-ins, then going through the array of plug-ins in order. If two plug-ins provide an option by the same name, then the first plug-in's option will be used[2]. All of the strings must be handled, and all options marked as MUST on their takesArgument property must be given an argument; otherwise an exception is thrown, and the program displays an error and exits. Also OptionHandlers may throw a BadOptionException, which will similarly cause the generator to exit with an error.

As well as changing implementations at run time, another advantage to this design is that it is general enough for these classes to be used by other applications. At present it is being used (with SpecSource, SchemaExtractor, SchemaIdentifier, and VariableExtractor) by z2b - another package in the CZT project which translates Z specifications into the B language.

It was later discovered that parts of this API for option handling were similar to Apache's Jakarta Commons CLI library[3], though this similarity was not intentional. Though Jakarta's CLI library is a more capable library for command-line processing, it probably would not handle choosing plug-ins at run-time as easily, or so easily change to use a GUI.

7.2 Plug-ins

All plug-in interfaces are in the

`net.sourceforge.czt.animation.gui.generator.plugin`

package. All of the default implementations are in the

`net.sourceforge.czt.animation.gui.generator.plugin.impl`

package.

The following subsections describe the interface of each kind of plug-in, the behaviour of the associated default plug-in, and possible alternative plug-ins.

7.2.1 Specification Source

Interface:

`net...plugins.SpecSource`

```
public interface SpecSource extends Plugin {
  public static final String optionName="source";
  public static final String name="Specification Source";
  public Term obtainSpec() throws IllegalStateException;
  public URL getURL();
};
```

[2]This is an unlikely occurrence, however Section 10.5 describes a solution.
[3]http://jakarta.apache.org/commons/cli/

The method `obtainSpec` is used to get the parsed specification, throwing an exception if it didn't get the information it needed from the option processing. The method `getURL` gets a URL that represents the location of the specification; this is used by the interface generator to store a reference to it in the `.gaffe` file.

Default Implementation:

`net...plugins.impl.SpecReaderSource`

The default specification source takes a file name or a URL through its options, and uses Corejava to load the Z specification in the ZML[11] file at that location.

Possible Variations:

If, in the future, the generator got combined into an IDE for Z, then another SpecSource plug-in could get the specification from one already loaded by the IDE.

7.2.2 Schema Extractor

Interface:

`net...plugins.SchemaExtractor`

```
public interface SchemaExtractor extends Plugin {
  public static final String optionName="extractor";
  public static final String name="Schema Extractor";
  public List getSchemas(Term spec);
};
```

The `getSchemas` method takes the specification acquired from `SpecSource`, and returns a list of Z schemas obtained from the specification.

Default Implementation:

`net...plugins.impl.VisitorSchemaExtractor`

The default schema extractor uses a visitor[2] to extract all schemas, regardless of what section of the Z specification they are in (and using a rather primitive test, which will not find schemas constructed with schema operators).

Possible Variations:

A smarter schema extractor might account for the scope rules of sections.

7.2.3 Schema Identifier

Interface:

`net...plugins.SchemaIdentifier`

```
public interface SchemaIdentifier extends Plugin {
  public static final String optionName="identifier";
  public static final String name="Schema Identifier";
  public void identifySchemas(Term specification,
                              List schemas)
    throws IllegalStateException;
  public ConstDecl getStateSchema();
  public ConstDecl getInitSchema();
  public List getOperationSchemas();
};
```

The method `identifySchemas` takes the specification from `SpecSource`, and the list of schemas from `SchemaExtractor`. After calling `identifySchemas` the other methods can be called to obtain the state schema, the initialisation schema, and the operation schemas respectively. Though there may be advantages to combining the schema identifier with the schema extractor (e.g. access to information on which sections schemas were in), they are performing separate tasks, and so are separate plug-ins. If there proves to be a need for section information, then that should be added to the collection of information passed between plug-ins by the main program.

Default Implementation:

`net...plugins.impl.CommandLineSchemaIdentifier`

Using its options, the default schema identifier is given the name of the state, initialisation, and operation schemas, and uses the schemas it finds by those names.

Possible Variations:

A smarter schema identifier might guess the function of each schema, based on whether it had primed variables, inputs, etc. Another possibility would be to use an unfolded schema; identifying, for example, operation schemas as those that included the delta of the state schema.

7.2.4 Interface Destination

Interface:

`net...plugins.InterfaceDestination`

```
public interface InterfaceDestination extends Plugin {
  public static final String optionName="destination";
  public static final String name="Interface Destination";
  public OutputStream obtainOutputStream(URL inputURL)
    throws IllegalStateException;
};
```

The `obtainOutputStream` method is given the URL of the specification from `SpecSource` (so that it can guess an output location, if desired). It returns an output stream to write the interface to.

Default Implementation:

`net...plugins.impl.FileInterfaceDestination`

The default interface destination plug-in takes a file name through its options, or tries to guess it based on the input file/URL. This guess is based on replacing `.zml` and `.xml` file suffixes with `.gaffe`.

7.2.5 Interface Generator

Interface:

`net...plugins.BeanInterfaceGenerator`

```
public interface BeanInterfaceGenerator extends Plugin {
  public static final String optionName="interface";
  public static final String name="Interface Generator";
  public void generateInterface(Term specification,
                                URL specURL,
                                List schemas,
                                ConstDecl stateSchema,
                                ConstDecl initSchema,
                                List operationSchemas,
                                VariableExtractor varExt,
                                BeanChooser beanChooser,
                                OutputStream os);
};
```

The `generateInterface` method takes the specification and URL obtained from `SpecSource`, the list of schemas from `SchemaExtractor`, the identified schemas from `SchemaIdentifier`, the `VariableExtractor` plug-in, the `BeanChooser` plug-in, and the stream to write the generated interface. It uses the variable extractor to find the variables it wants, uses the bean chooser to select the proper bean to display each variable, creates the interface (creating forms and beans, setting layouts and properties, etc.), and writes it to the output stream.

Default Implementation:

`net...plugins.impl.BasicBeanInterfaceGenerator`

The default interface generator produces:

- one main window for state, with buttons for operation schemas, and history buttons at the bottom.
- one window per operation schema for inputs.
- one window per operation schema for outputs.

It creates the actual interface, and outputs it in the same way that the designer does. Screenshots of the interface generated for the BirthdayBook example can be found on the following pages.

Possible Variations:

There is an almost unlimited number of variations on how the generator could lay out interfaces; merge input and output window, merge input, output and state windows, combine input fields used by different operations, ...

Alternative Implementations:

With little modification to other plug-ins (except the bean chooser), it should be possible to make an interface generator that produced XML documents rather than creating all of the interface objects. Doing this might improve the generator's performance, but experimentation has shown that creating the interface the same way as the designer does not cause any problems, and doing it via an XML document leads to the generator plug-in's source code becoming long, repetitive, and confusing.

Example Interface Screenshots - BirthdayBook

These screenshots are from the Gaffe animator, using an interface generated from the BirthdayBook example Z specification in Appendix A.

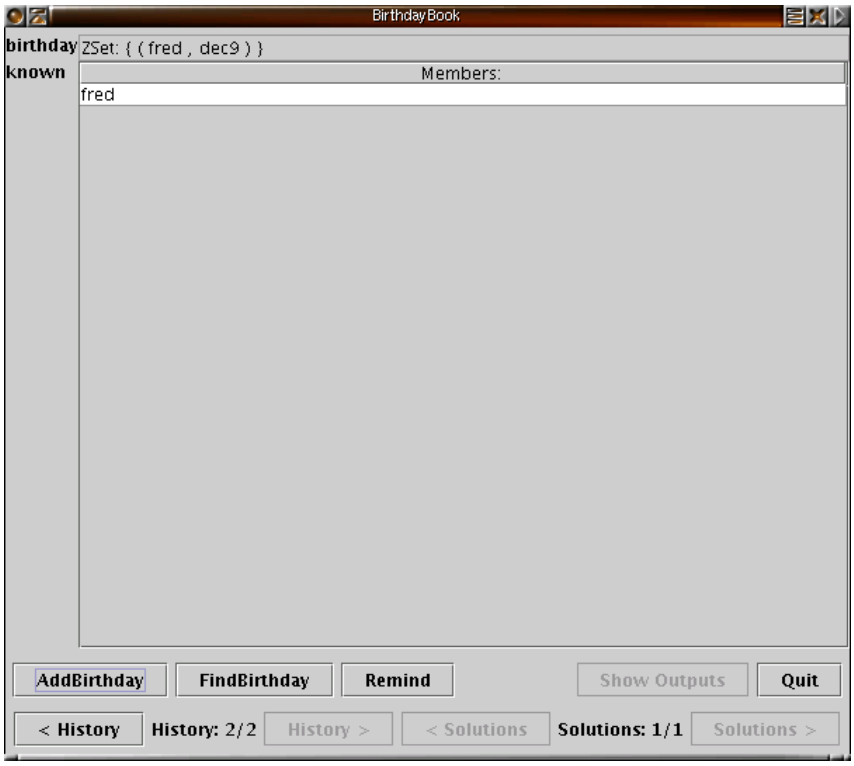

Figure 7.2: The state window of the generated interface.

Note that, although the birthday variable is a relation, it has appeared as a text field rather than as a table. This is because, the Gaffe generator is not yet very smart about determining some variable types. Once the CZT type-checker is written, Gaffe will be able to use it to determine variable types, and the generator will correctly produce a two column table.

The variables are updated by a script function (`fillBeans`) that is called by a script which is triggered by a `HistoryProxy`[4]; this function matches Z variables to components based on the name property of the component, then does what is needed, depending on the component's type, to display the variable through the component. The first row of buttons contains one for each operation, each of which trigger scripts that open the appropriate input window (Finding the appropriate form based on its `name` variable.). The top row also contains a button to redisplay the output window of the previous operation in the history (the window is tracked with an ECMAScript variable), and a button to quit from the Gaffe animator.

[4] `HistoryProxy`s forward events from the history, indicating a change in the current solution, so that beans within a form can respond to them.

The bottom row of buttons contains buttons to step back and forth through the history, and back and forth through the current set of solutions [5]; also it contains labels to display the current position in the history. The scripts associated with these buttons just call the appropriate method on the `History` object, and the labels are updated by a `HistoryProxy`.

Figure 7.3: The AddBirthday input window.

Input windows display their variables in much the same way that the state window does, except that the fields are editable. The 'OK' button will activate the operation schema, and display the output window if appropriate. The 'Cancel' button will close the window without activating the schema. Instead of using `fillBeans`, another script function `fillHistory` is used by the 'OK' button's script to get the data from the entry fields, after which the fields are cleared, the window closed, the schema activated via the `History` object, and the output window (if there is one) opened.

Figure 7.4: The FindBirthday input window.

Figure 7.5: The FindBirthday output window.

Output windows are similar again, containing non-editable fields filled by the `fillBeans` function. The 'Close' button will close the outputs window; however it can be redisplayed using the 'Show Outputs' button in the state window.

[5] Though because BirthdayBook is deterministic, there will never be more than one, so these buttons are disabled.

Figure 7.6: The Remind input window.

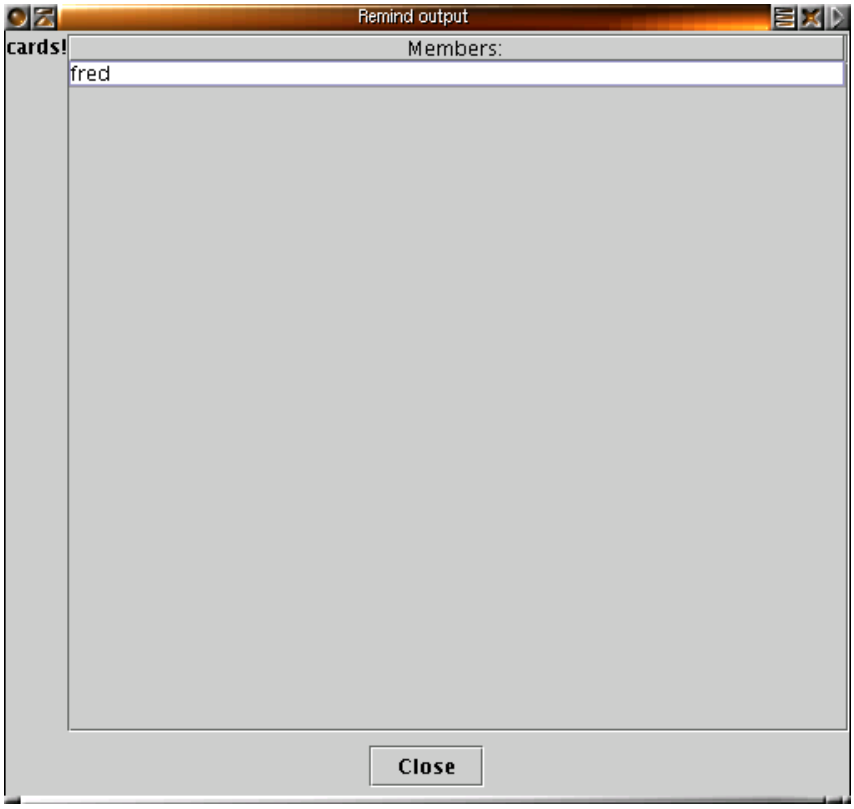

Figure 7.7: The Remind output window.

7.2.6 Variable Extractor

Interface:

`net...plugins.VariableExtractor`

```
public interface VariableExtractor extends Plugin {
  public static final String optionName="variable";
  public static final String name="Variable Extractor";
  public Map getInputVariables(ConstDecl schema);
  public Map getOutputVariables(ConstDecl schema);
  public Map getStateVariables(ConstDecl schema);
  public Map getPrimedVariables(ConstDecl schema);
  public Map getNumberedVariables(ConstDecl schema);
};
```

Each method gives a collection (a mapping from variable name to variable declaration) of variables of a specific role, to appear in the generated interface. The roles being input variables which end with a ?, output variables which end with a !, state variables which have no decoration, primed variables which end with a $'$, and numbered variables which end with a subscript number.

Default Implementation:

`net...plugins.impl.DefaultVariableExtractor`

The default variable extractor gets *all* variables of a requested type from a schema. At present this implementation requires that the schema be unfolded, i.e. that it not include another schema (as it does not look for variables inherited from inclusions).

Possible Variations:

Later versions of the variable extractor may handle folded schemas. Other extractors might be used, to block some variables from appearing in the Gaffe interface.

7.2.7 Bean Chooser

Interface:

`net...plugins.BeanChooser`

```
public interface BeanChooser extends Plugin {
  public static final String optionName="bean";
  public static final String name="Bean Chooser";
  public Component chooseBean(Term specification,
                              ConstDecl schema,
                              String variableName,
```

```
                                VarDecl variableDeclaration,
                                boolean editable);
};
```

The `chooseBean` method returns a `Component` suitable for displaying/entering a variable. It takes the specification from `SpecSource`, the containing schema, the variable's name, the variable's declaration, and a flag to set whether it is for input.

Default Implementation:

`net...plugins.impl.BasicBeanChooser` The default bean chooser produces a text field for most variables, but will produce a table for sets, tuples, bindings, and relations (though it isn't yet very good at identifying relations).

Possible Variations:

When the CZT type-checker is written, this could be used to decide what type of variable is being looked at, and so what kind of bean to use. Another bean chooser might allow the user to specify a bean to use for certain types.

Chapter 8

Problems Encountered

This chapter describes some of the problems encountered while developing Gaffe; the intention being to show the reasons for design decisions made, and to recount solutions that may be useful to other projects.

8.1 Showing components in a form design while blocking their use

The designer needed to be able to show the components as they would appear in the form, however those components must not be usable. For example a button that was part of a form's design should not be usable from the designer, as it might trigger unwanted side-effects, possibly changing the interface. One early idea was to have a component used to stand in for all components in the designer, which would delegate its `paint` call to the correct component's `paint` method, trying to fool that component into thinking it was painting itself when it was painting the stand-in. Because this would break accepted assumptions about the use of `paint`, it was decided that there was too much risk of this causing unintended side-effects, and the actual components should be used in the designer.

Experiments were made with overloading the event-handling methods of the main panel of the form design window. This was done in the hope that events could be blocked from being delivered to its children, however this did not work; because events bubble up from child to parent, they do not filter down from parent to child.

The final solution that worked was `JLayeredPane` from Java's Swing library. `JLayeredPane` allows layered containers ('panes'); including 'glass' panes that will trap user input, but are transparent so users can see the interface on a pane 'beneath' it. By using a `JLayeredPane` with the same form used

in the Gaffe animator on its bottom pane, and a glass pane that handles user interaction (and displays handles for resizing, event link highlighting, etc.) on its top pane, the designer can show forms just as they will appear in the Gaffe animator, while preventing the forms' controls from being used.

8.2 Nesting panels in a form

Early versions of Gaffe didn't allow panels to be nested inside a form (nested panels are useful for improving layout), and it was expected that making the transition to allow nested forms would be difficult. These difficulties were expected to occur in the designer, where it would be necessary to identify components being pointed at by a mouse, and to determine whether, for example, a button was the object of interest, or the panel containing it, or the form containing that. These problems essentially come down to translating between the coordinate space of the design window, and the coordinate space of the form and the various panels[1]. Because these issues were anticipated from the start of writing `FormDesign`, when the time came to make the transition only minor fixes were needed to allow for the extra levels of depth.

8.3 Unwanted items being saved by the Encoder

When an object is saved using the `java.beans` library's `Encoder` class, all of its properties (with exceptions mentioned in the next paragraph) are saved too, and all of its listeners. The problem comes when saving of some of this information isn't wanted.

`Encoder` allows any property of any class to be blocked from being encoded. So for example, the name property for the `JButton` class can be marked as transient, and when encoded no `JButtons` will have their name recorded. However there are places where this doesn't quite work 'as advertised'; also listeners aren't blockable via this mechanism.

For the majority of cases blocking unwanted properties was just a matter of marking that property transient, however in the case of 'location' for `Components`, this does not work due to the default `PersistenceDelegate` for `Component`; it treats setting 'size' and 'location' differently, because they can be integrated into one call to `setBounds`, however the special handling doesn't repeat the check for transient status that would normally be done.

[1]Utility methods for which are in `FormDesign`.

Because the location property for `Form`s is only used by the designer, saving it wasn't wanted. The work-around that was chosen in the end was a `PersistenceDelegate` for `Form` that tricked the `Encoder` into thinking that the default location was its actual location, so it needn't be set. Though this was an unpleasant solution, it saved repeating all of the work done by the delegate for `Component`s.

The other major place at which unwanted items were being saved (or trying to be saved) was the listeners attached to objects. The Gaffe designer does not attach listeners to the objects in its interface, instead it keeps a separate record of the event links which is saved separately in the file. The only things that do get registered as listeners on beans in an interface during design, are parts of the designer (e.g. The handles used for resizing beans register listeners on their associated component, to know when to move themselves). `Encoder` however will try to save these listeners anyway; and will often fail because the listeners do not follow the design pattern for beans. This failure will produce unwanted error output, and if the `Encoder` does manage to save one of these listeners, then it could produce unwanted effects when animating. Unfortunately allowances are not made for blocking listeners, as can be done with properties. Experiments were tried with stripping listeners from objects before saving them, but this became rather complicated. The solution finally used was one that filtered the statements written to file, blocking those that matched the pattern for registering a listener.

Even though the solutions needed to work around shortcomings in `Encoder` are unpleasant, the alternative - implementing its equivalent - would not be worthwhile, because of the development time involved, and the inability to take advantage of `PersistenceDelegate`s written for products that do use `Encoder`.

8.4 Getting parameters into the Script bean's script

It is desirable that a script take certain arguments, so that a script could more easily identify its containing bean, the form that contained that, and the cause for the script being fired. BSF provides a method for evaluating an anonymous function[2], giving it named arguments. However, the default behaviour for this method is to run the script while ignoring the arguments,

[2]Having scripts run as functions is also desirable, so that their variables have restricted scope.

and this is the behaviour used by the current JavaScript engine! The only other way to get information into the script from outside is by 'declaring'[3] a bean, this makes it available to *all* scripts; this is not desirable, because if Gaffe were to become multi-threaded, there might be two scripts running at once but only one name for an argument to go under. The solution chosen was to wrap extra code around the script, turning it into an anonymous function, and calling it with arguments that find the form and script beans (using their name property, via the list of forms that is registered with BSF as a globally available bean). This solution is applied only if the script is marked as being in ECMAScript. Hopefully a later version of BSF will make use of the named parameters, and this work-around can be removed.

So, for example, a script:

```
clearBeans(thisForm);thisForm.setVisible(false);
```

might become:

```
(function (thisForm, thisScript) {
  clearBeans(thisForm);thisForm.setVisible(false);
})(Forms.lookup("AddBirthday input"),
   Forms.lookup("AddBirthday input").beans[1]);
```

8.5 Slow performance finding property editors

The `java.beans` library's `PropertyEditorManager` is in charge of finding `PropertyEditors`, used by the properties window to edit the properties of a bean. Unfortunately it is capable of being quite slow.

If a class does not have an editor registered with the manager, then it will search through the entire class path trying to find one that matches the class name followed by '`Editor`'. This is done repeatedly when the properties window is displayed. At one stage this would produce a very noticeable pause, because `PropertyEditorManager` does not cache missing editors; so a few common classes without editors would result in repeated searches through the class path. The rather straight-forward solution to this was to add a cache which cached both hits and misses from the manager. Now there can still be a slight pause on the first viewing of the properties window, but not as severe and it occurs only once.

[3] See Chapter 5.

8.6 Flexible configuration

Some way was needed to make addition of tools, bean types, and property editors achievable without recompiling. It was a simple matter to add an initialisation script to the designer, allowing these and other settings to be configured.

8.7 No back-end

Because the animator engine that Gaffe attaches to isn't written yet, all testing has been done with custom `History` implementations that fake the back-end for a particular specification.

Chapter 9

Usability Study

9.1 Purpose

A usability study was performed to gauge the quality of the Gaffe Designer's user interface, identify usability problems, identify unclear or ambiguous aspects of the interface, and to study how people use the Designer in order to find ways to make it easier to use.

9.2 Procedure

Participants were selected from Computer Science students at the University of Waikato. All sessions were conducted in the Software Engineering Lab in room G.2.06 of the university.

After signing consent forms, participants were asked to:

- complete a questionnaire to determine their previous experience with animators and interface builders,
- perform a series of exercises using the Gaffe Designer, and
- complete a questionnaire to gauge their experience during the exercises, including rating from 1 to 7 how easy they found certain tasks.

Participants were encouraged to think aloud, and ask questions during the exercises. While they performed the exercises, notes were taken on their comments, questions, and observed actions.

As well as the booklet containing the questionnaires and exercises (See Appendix C), they were given access to a simple user manual describing how to use the designer (Appendix D). Only one participant took more than a brief look at this manual, the remainder only skimming through to it when they got stuck.

9.2.1 Exercises

The exercises involved modifying an interface based on one generated for the BirthdayBook specification, and were as follows:

1. Go to the 'BirthdayBook' form, and highlight all event links.

2. Choose the panel containing just the 'AddBirthday', 'FindBirthday', and 'Remind' buttons, and change its background colour to something more festive.
   If you like, change the colour of some other panels.

3. Delete the Solutions part of the history navigation panel. The BirthdayBook specification is non-deterministic, so it will not be needed. Don't forget to delete the scripts associated with the buttons you deleted.

4. Make a new form called 'Test Window'. Put a label in it with some text ("Happy Birthday!"), and a button labelled 'Close'.

5. Make a new button labelled 'Test' on the 'BirthdayBook' form in the panel containing the operation buttons. Make the new button open the new form you created. Make the close button in your new form close the form. (Hint: Any scripts you make will be almost identical to scripts attached to similar buttons).

Participants had the option of testing the changes they made using the Gaffe animator. This option was used by all participants at the very least, once they'd finished the exercises.

9.3 Participants

Six computer science students from the University of Waikato were selected to be participants. Only two of these participants claimed a significant (answered with a rating of four out of seven or higher in the initial questionnaire) amount of experience with Z. The remainder of the participants claimed little or no experience with Z, that experience generally being limited to the University of Waikato's COMP-424 paper - Topics in Software Engineering. Only one participant claimed significant experience with Z animators. Two participants claimed significant experience with application builders; participants' previous experience tended to be with Microsoft's Visual Studio, or Visual Basic.

9.4 Results

9.4.1 Issues Found

Most participants said that they found the properties and tool windows easy to use; however several had difficulty finding them, because they weren't where they expected. The Gaffe designer has since been modified to amend this; the properties window is now also available through the edit menu, where most participants first looked, also the toolbox window now appears when the designer starts up. Other things participants expected in the edit menu include, bean deletion, clipboard functionality, and in one case undo. "Delete Current Bean" has since been added to the edit menu. Clipboard and undo functionality are somewhat more complicated to implement; but when undo is implemented, it should probably use the `javax.swing.undo` library available for that purpose.

Some found it hard to determine what "Highlight All Event Links" had done, because the form design window wasn't sized large enough to show the parts that included event links. One participant's suggestion, which should be implemented, was that scroll-bars would make it clearer that the window had more content. In addition to this solution, it would be sensible for `FormDesign` windows to start at a size large enough (within reasonable limits) to show the entire design without scrolling. Several participants were also surprised that it was necessary to highlight links for each window, rather than the setting applying to all form design windows. This has since been partially corrected (work is needed to make menus in all windows reflect the current settings); in addition to which "Highlight All Event Links" is now the default, not "Don't Highlight Event Links".

About half of the participants tried to edit label or button text, or the form's name from the form design window. The same participants also tended to try double-clicking on a bean to open the properties window for that bean. Allowing text to be edited straight from the form design window would probably be impractical, because the actual beans are being used. Opening the properties window or the customiser for a bean should however be achievable; though it may be difficult to tell a click to select a different bean, from the first click of a double-click. When an attempt was made to implement this double-click behaviour, without taking the effort to distinguish click from first click, the result was that a child of the current bean would be selected, and the properties window would open on the second.

Once in the properties window, users had little trouble editing properties,

though one unpleasant bug was found. When changing a text property, if a different property is selected afterwards or the Enter key is hit, then it will be stored properly. However, if the properties window is closed, or another bean is selected in a form design window, then the changes will be lost! Though a small thing, it is certainly an unacceptable behaviour, and tripped up most of the participants. The easiest way to correct this would be to add checks for loss of focus on the property window, and change of current bean.

The other problem participants were observed to have with the properties window (though it was not commented on in the questionnaire), was an attempt to use the events and methods tables. This generally happened either when they were trying to set up a script or an event link, but had not yet found these tools. It may be desirable to grey-out the entries in these tables, as is done with the non-editable entries in the properties table, so that users will not presume they can be used or modified.

One participant found that they were not editing the properties of the bean they thought they were. This turned out to be because they assumed that the currently selected bean would change when they changed the current window. It would be a minor matter to make this the case, but it should be considered whether this will complicate use when clipboard functionality is added. Another problem that a different participant had with selection, was that they expected to be able to deselect the current bean so that there would be no currently selected bean. This is not possible at present, would require changes in several parts of the designer, and would be unlikely to enhance usability much if at all; so making this possible would be at most a low priority change.

Other than problems relating to lack of familiarity, the main problem experienced, that has not been already mentioned, is that the "Test" button would change back after they resized or repositioned it. This was because the panel it was in had a `LayoutManager` associated with it by the Gaffe generator. There is not much that can be done to stop this happening, but perhaps the bean's resize and movement handles could be disabled when their use would have no effect.

One tool that at least half of the participants had problems with at first was the `Script` bean tool. These problems were due to them not realising that non-visual beans could not be placed inside a form. One participant suggested that if the cross cursor (used to indicate where beans can not be placed) was a different colour, then it would be more noticeably distinct from the normal cross-hairs cursor. Another possibility is to report an error in a dialog box, to inform users as to why this will not work. Other problems participants had

with tools are determining how to use the event link tool (drag from source to listener), and identifying the function of some tools from their icons; both of these were quickly worked out and provoked little or no comment from participants.

Despite all of the issues mentioned that participants experienced, they were generally positive about the designer. Several commented that once they found out how, they found the designer easy to use. Participants typically took half an hour (with one exception, at one hour, who took time to read the manual); this was well below the estimated maximum time of $1\frac{1}{2}$ hours. All participants successfully performed the required changes, though in some cases there were minor divergances (for example putting the "Test" button in a different panel, or adding an unnecessary call to `clearBeans(...)` when they copied the body of the close script). So, in spite of the usability issues that came up, the designer is evidently quite usable for the types of tasks tested in this study.

9.4.2 Participants' Rating of Their Experience

Question	Beans	New Form	Scripts	Events	Understanding	Overall
	1	2	3	4	5	6
Average	$5\frac{4}{6}$	$5\frac{3}{6}$	$4\frac{3}{6}$	$5\frac{1}{6}$	$5\frac{4}{6}$	$5\frac{1}{6}$
Minimum	4	4	2	3	5	4
Maximum	7	7	7	7	7	6

Participants rated all parts of their experience with the designer quite highly. The main areas that caused issue were creating event-links and scripts, with ratings given as low as two and three respectively. Opposed to which they found creating beans and forms easy.

Chapter 10

Future Work

10.1 Link to the animator engine

The most important item on the 'to do' list, is to make whatever changes are necessary to connect Gaffe to the animator engine (once that is written). This will be a small change, affecting only the `activateSchema(...)` method in `History`.

10.2 Smarter generator plug-ins

At present the plug-ins to the generator are quite simple; they don't try to do anything clever, but do need several command-line inputs. Some possibilities for smarter/more flexible plug-ins are mentioned in Section 7.2.

10.3 Make use of other CZT libraries

At present the Gaffe generator plug-ins require that a specification's schemas be unfolded, so that they don't have to worry about included schemas. At some stage methods will be written for CZT that unfold the syntax trees that specifications are read into. Once these methods are available, they should be used by Gaffe so that specifications that haven't been unfolded may be used.

At present the Gaffe generator has to figure out schema variable types by itself. Once the type-checker is written, it could be used to properly handle this task.

10.4 GUI for generator

In the current version of Gaffe, both the Gaffe animator and the designer are GUI applications, but the generator is solely console based. As has been noted in Section 7.1, it would not be difficult to make option entry come from a GUI. As a bonus, other programs that use `PluginList` for their option processing could use the GUI as well!

10.5 Namespacing of generator options

At present it is possible for two generator plug-ins to provide options with the same name, in which case the option belonging to the first plug-in in the `PluginList` is used, and the second one is not. By optionally allowing the plug-in's name to prepend the option, this situation could be disambiguated. So for example `SpecSource`'s '-spec' option could alternately be entered as '-SpecSource.spec'.

10.6 Allow scripts to fire or handle any type of event

For scripts to have an equal footing with normal beans, they need to be able to handle any type of event, and to trigger any type of event. Allowing this would enable scripts to interact with a larger range of beans and listeners (possibly created by others), which may make use of new types of events.

The former issue is fairly easy to handle. The `java.beans` library provides `EventHandler` which acts as a back-end, doing all of the processing, for a class created at run-time from a listener interface. This allows any listener type to be chosen at run-time, and the same object will be able to pretend it is that type. A few changes to the code for creating event links in the designer, and for loading them in the Gaffe animator, would allow `EventHandler`s to be used in place of listeners. Then it would just be a matter of making `Script` inherit from `EventHandler` instead of `ActionListener`.

Allowing scripts to trigger any type of event is harder; there is no similar class provided for this purpose. One possibility though, would be to use `EventListenerList` (a utility class for tracking listeners). Any beans that took advantage of this however would not be similarly advantaged in other bean-builders. Because the `Script` bean is unlikely to be used with other builders, this is unlikely to be a problem.

10.7 Allow forms to have menus

At present the designer does not allow for menus and menu bars. Because menus are significantly different from normal components, this would probably mean a separate editor in the Gaffe designer for handling menus. This could be handled as a property editor in the properties window, because a menu bar is a property of its window; if this route is chosen however, the menu editor should also be made quickly accessible from the form design window.

10.8 More of everything

More tools, beans, property editors, history implementations, and generator plug-ins will increase the flexibility and choice for designers of Gaffe interfaces. One possibility, suggested by Greg Reeve, is described in the next section.

10.9 Scrolling history

A different style of interface that displays multiple steps in the history at once using a scroll-pane, would provide another way to interact with the animator, and an alternative to previous/next history buttons. So a user could scroll up in that pane to see previous operations, possibly with the results of more than one operation visible at a time.

Early in development, it was thought that doing this would require a separate implementation of the Gaffe animator. It is for this reason that the UML diagram in Appendix B splits `AnimatorCore` into `AnimatorCore` and `AnimatorCoreBase`. However with the flexibility possible with Gaffe interfaces, thanks to JavaBeans and ECMAScript, it should be possible to make a JavaBean that implements the scrolling history. This bean would make a copy of an appropriate form for each state in the history. The one restriction with this is that it may require a special implementation of the history class, to allow multiple points in the history to be visible at once.

Chapter 11

Conclusions

Split into three parts for manual design of user interfaces, automatic generation of those interfaces, and Z animation; Gaffe provides a flexible, modifiable GUI front-end for a Z animator. It places no restrictions on the structure, content, or behaviour of an interface. Most of Gaffe's flexibility can be attributed to its use of JavaBeans. JavaBeans proved to be straight-forward to work with; though it did have some frustrating quirks, especially with regards to persistence (See Section 8.3). Adding to Gaffe's flexibility is support for ECMAScript, implemented using the BSF library (See Chapter 5). Using BSF was extremely easy, though it too has its minor failings (See Chapter 8.4). Using the Java language also helped to provide flexibility; because classes can be loaded dynamically, the selection of beans that can be added to an interface is not restricted by what was available at compile-time.

Even in parts of Gaffe not immediately dealing with user interface design, flexibility and customisability were significant watchwords. The generator was split into several stages implemented as plug-ins, so that the implementation of each stage can be changed. With option processing separated out, it will be a small matter to allow the generator to be configured from places other than the command line (e.g. a GUI). This modularity has also proven beneficial to other projects; as CZT's z2b project uses the option processing, and several of the plug-ins provided by the generator.

Despite some usability issues (See Chapter 9), the Gaffe designer proved to be usable by new users with computer science experience. These new users (participants in the usability study) rated its usability quite highly.

Though most parts of the designer were shown to be usable, there is still work to be done. Apart from correcting issues found in the usability study, there are several improvements (See Chapter 10) that will improve the usability and capabilities of Gaffe. Most important of these is to attach it to the animator engine once that has been written.

Bibliography

[1] Andrew P. Martin. "Proposal: Community Z Tools Project (CZT)." http://web.comlab.ox.ac.uk/oucl/work/andrew.martin/CZT/proposal.html, September 2001.

[2] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software." Addison Wesley, USA, 1995.

[3] J.M. Spivey. "The Z Notation: A Reference Manual." International Series in Computer Science. Prentice-Hall International (UK) Ltd, second edition, 1992.

[4] Graham Hamilton (Editor) "JavaBeans^{TM} API Specification." Sun Microsystems, version 1.01, 1997.

[5] ECMA International "ECMAScript Language Specification." third edition. Available at http://www.ecma-international.org/publications/standards/Ecma-262.htm

[6] "ECMAScript language specification." ISO/IEC Standard 16262. ISO/IEC JTC1/SC22.

[7] Philip Milne and Kathy Walrath "Long-Term Persistence for JavaBeans." The Swing Connection. Sun Microsystems. Available at http://java.sun.com/products/jfc/tsc/articles/persistence/

[8] Philip Milne "Long-Term Persistence: An Update." The Swing Connection. Sun Microsystems. Available at http://java.sun.com/products/jfc/tsc/articles/persistence2/

[9] "Long Term Persistence of JavaBeans Components: XML Schema." The Swing Connection. Sun Microsystems. Available at http://java.sun.com/products/jfc/tsc/articles/persistence3/

[10] Philip Milne "Using XMLEncoder." The Swing Connection. Sun Microsystems. Available at http://java.sun.com/products/jfc/tsc/articles/persistence4/

[11] Mark Utting, Ian Toyn, Jing Sun, Andrew Martin, Jin Song Dong, Nicholas Daley, David Currie "ZML: XML Support for Standard Z." ZB2003 Conference Proceedings.

[12] Andy Quinn "Trail: JavaBeans^{TM}" The Java^{TM} Tutorial. Sun Microsystems. Available at http://java.sun.com/docs/books/tutorial/javabeans/

[13] Dan Hazel "Possum Users Guide" Software Verification Research Center, University of Queensland. Available at http://www.itee.uq.edu.au/ comp4600/software/user_guide_draft.ps.gz

Appendix A

Birthday Book Example

A.1 The Z Specification

This Z specification is adapted from that given in Spivey's "The Z Notation" [3]. Narrative text has been cut down or removed, and the schemas have been unfolded. This has been done, because at present Gaffe and the CZT libraries don't have code to unfold schemas automatically.

In the next section, is the source code for the history object used while testing with this specification, to emulate the behaviour of the animator engine.

$[NAME]$
$[DATE]$

The state schema:

$$
\begin{array}{|l}
\hline
\mathit{BirthdayBook} \\
\mathit{known} : \mathbb{P}\, \mathit{NAME} \\
\mathit{birthday} : \mathit{NAME} \pfun \mathit{DATE} \\
\hline
\mathit{known} = \mathrm{dom}\, \mathit{birthday} \\
\hline
\end{array}
$$

The initialisation schema:

$$
\begin{array}{|l}
\hline
\mathit{InitBirthdayBook} \\
\mathit{known}' : \mathbb{P}\, \mathit{NAME} \\
\mathit{birthday}' : \mathit{NAME} \pfun \mathit{DATE} \\
\hline
\mathit{known}' = \mathrm{dom}\, \mathit{birthday}' \\
\mathit{known}' = \{\} \\
\hline
\end{array}
$$

The operation schemas:

AddBirthday

$$known, known' : \mathbb{P}\, NAME$$
$$birthday, birthday' : NAME \pfun DATE$$
$$name? : NAME$$
$$date? : DATE$$

$$known = \mathrm{dom}\, birthday$$
$$known' = \mathrm{dom}\, birthday'$$
$$name? \notin known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

FindBirthday

$$known, known' : \mathbb{P}\, NAME$$
$$birthday, birthday' : NAME \pfun DATE$$
$$name? : NAME$$
$$date! : DATE$$

$$known = \mathrm{dom}\, birthday$$
$$known' = \mathrm{dom}\, birthday'$$
$$known' = known$$
$$birthday' = birthday$$
$$name? \in known$$
$$date! = birthday(name?)$$

Remind

$$known, known' : \mathbb{P}\, NAME$$
$$birthday, birthday' : NAME \pfun DATE$$
$$today? : DATE$$
$$cards! : \mathbb{P}\, NAME$$

$$known = \mathrm{dom}\, birthday$$
$$known' = \mathrm{dom}\, birthday'$$
$$known' = known$$
$$birthday' = birthday$$
$$cards! = \{n : known \mid birthday(n) = today?\}$$

Robust versions of the operations:

$$REPORT ::= ok \mid already\_known \mid not\_known$$

Success

$$result! : REPORT$$

$$result! = ok$$

AlreadyKnown

$$known, known' : \mathbb{P}\, NAME$$
$$birthday, birthday' : NAME \pfun DATE$$
$$name? : NAME$$
$$result! : REPORT$$

$$known = \mathrm{dom}\, birthday$$
$$known' = \mathrm{dom}\, birthday'$$
$$known' = known$$
$$birthday' = birthday$$
$$name? \in known$$
$$result! = already\_known$$

NotKnown

$$known, known' : \mathbb{P}\, NAME$$
$$birthday, birthday' : NAME \pfun DATE$$
$$name? : NAME$$
$$result! : REPORT$$

$$known = \mathrm{dom}\, birthday$$
$$known' = \mathrm{dom}\, birthday'$$
$$known' = known$$
$$birthday' = birthday$$
$$name? \notin known$$
$$result! = not\_known$$

$$RAddBirthday \mathrel{\widehat=} (AddBirthday \land Success) \lor AlreadyKnown$$
$$RRemind \mathrel{\widehat=} Remind \land Success$$
$$RFindBirthday \mathrel{\widehat=} (FindBirthday \land Success) \lor NotKnown$$

A.2 The Fake Back-End History

```
/*
  GAfFE - A (G)raphical (A)nimator (F)ront(E)nd for Z
        - Part of the CZT Project.
  Copyright 2003 Nicholas Daley

  This program is free software; you can redistribute it and/or
  modify it under the terms of the GNU General Public License
  as published by the Free Software Foundation; either version 2
  of the License, or (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program; if not, write to the Free Software
  Foundation, Inc., 59 Temple Place - Suite 330,
  Boston, MA  02111-1307, USA.
*/
package net.sourceforge.czt.animation.gui.temp;

import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.Vector;

import net.sourceforge.czt.animation.ZLocator;
import net.sourceforge.czt.animation.gui.history.BasicHistory;

/**
 * History backend for interface generated from
 * <code>birthdaybook_unfolded.xml</code>.
 */
public class BirthdayBookFullHistory extends BasicHistory
{
  private static ZGiven ok = new ZGiven("ok");
  private static ZGiven already_known = new ZGiven("already_known");
  private static ZGiven not_known = new ZGiven("not_known");

  /**
   * Constructs a new birthday book history.
   * With no birthdays recorded in it.
   */
  public BirthdayBookFullHistory()
```

```
{
  super();
  Map newResultsM = new HashMap();
  newResultsM.put("birthday'", new ZSet());
  newResultsM.put("known'", new ZSet());
  solutionSets.add(new SolutionSet("InitBirthdayBook",
                                   new ZBinding(newResultsM)));
  currentSolution = solutionSets.listIterator();
  System.err.println("History initialised: ");
  System.err.println("Current SolutionSet: "
                     + getCurrentSolutionSet());
  System.err.println("Current Solution: " + getCurrentSolution());
};
/**
 * Calculates the next set of solutions based on the inputs it has
 * received.
 * The following schemas are handled:
 * <ul>
 *  <li><code>AddBirthday</code></li>
 *  <li><code>RAddBirthday</code></li>
 *  <li><code>FindBirthday</code></li>
 *  <li><code>RFindBirthday</code></li>
 *  <li><code>Remind</code></li>
 *  <li><code>RRemind</code></li>
 * </ul>
 * @param schemaName The name of the oepration schema to activate.
 */
public synchronized void activateSchema(String schemaName)
{
  System.err.println("Schema activated: " + schemaName);
  System.err.println("Current SolutionSet: "
                     + getCurrentSolutionSet());
  System.err.println("Current Solution: " + getCurrentSolution());
  System.err.println("Inputs:");
  for (Iterator i = inputs_.keySet().iterator(); i.hasNext();) {
    Object a = i.next();
    System.err.println("   " + a + "\t" + inputs_.get(a));
  };

  ZBinding currentResults = getCurrentSolution();
  if (currentResults == null) return;
  ZBinding newResults;
  Map newResultsM = new HashMap();
  final ZSet newBirthdays;
  final ZSet newKnown;
  final ZGiven resultOutput;
  final ZSet currentBirthdays
    = (ZSet) currentResults.get("birthday'");
  final ZSet currentKnown = (ZSet) currentResults.get("known'");
  if ("AddBirthday".equals(schemaName)) {
    final ZGiven nameInput
```

```
    = (ZGiven) inputs_.get(ZLocator.fromString("name?"));
  final ZGiven dateInput
    = (ZGiven) inputs_.get(ZLocator.fromString("date?"));
  System.err.println("+++++" + nameInput + "\t" + dateInput);
  if (currentKnown.contains(nameInput)) {
    solutionSets
      = new Vector(solutionSets.subList(0,
                                currentSolution.nextIndex() + 1));
    solutionSets.add(new SolutionSet(schemaName,
                                     Collections.EMPTY_SET));
    currentSolution
      = solutionSets.listIterator(solutionSets.size() - 1);
    System.err.println("Schema completed: " + schemaName);
    propertyChangeSupport.firePropertyChange(
                                     "currentSolutionSet",
                                     null, null);
    propertyChangeSupport.firePropertyChange("currentSolution",
                                             null, null);
    return;
  } else {
    Set s = currentKnown.getSet();
    s.add(nameInput);
    newKnown = new ZSet(s);
    s = currentBirthdays.getSet();
    s.add(new ZTuple(nameInput, dateInput));
    newBirthdays = new ZSet(s);
    resultOutput = null;
  }
} else if ("RAddBirthday".equals(schemaName)) {
  final ZGiven nameInput
    = (ZGiven) inputs_.get(ZLocator.fromString("name?"));
  final ZGiven dateInput
    = (ZGiven) inputs_.get(ZLocator.fromString("date?"));
  System.err.println("+++++" + nameInput + "\t" + dateInput);
  if (currentKnown.contains(nameInput)) {
    newBirthdays = currentBirthdays;
    newKnown = currentKnown;
    resultOutput = already_known;
  } else {
    Set s = currentKnown.getSet();
    s.add(nameInput);
    newKnown = new ZSet(s);
    s = currentBirthdays.getSet();
    s.add(new ZTuple(nameInput, dateInput));
    newBirthdays = new ZSet(s);
    resultOutput = ok;
  }
} else if ("FindBirthday".equals(schemaName)) {
  final ZGiven nameInput
    = (ZGiven) inputs_.get(ZLocator.fromString("name?"));
  ZGiven dateOutput
```

```
    = (ZGiven) inputs_.get(ZLocator.fromString("name?"));
  newBirthdays = currentBirthdays;
  newKnown = currentKnown;
  if (currentKnown.contains(nameInput)) {
    for (Iterator iter = currentBirthdays.iterator();
         iter.hasNext();) {
      ZTuple t = (ZTuple) iter.next();
      if (t.get(0).equals(nameInput)) {
        dateOutput = (ZGiven) t.get(1);
        break;
      }
    }
    resultOutput = null;
    newResultsM.put("date!", dateOutput);
  } else {
    solutionSets
      = new Vector(solutionSets.subList(0,
                                currentSolution.nextIndex() + 1));
    solutionSets.add(new SolutionSet(schemaName,
                                     Collections.EMPTY_SET));
    currentSolution
      = solutionSets.listIterator(solutionSets.size() - 1);
    System.err.println("Schema completed: " + schemaName);
    propertyChangeSupport.firePropertyChange(
                                     "currentSolutionSet",
                                     null, null);
    propertyChangeSupport.firePropertyChange("currentSolution",
                                             null, null);
    return;
  };
} else if ("RFindBirthday".equals(schemaName)) {
  final ZGiven nameInput
    = (ZGiven) inputs_.get(ZLocator.fromString("name?"));
  ZGiven dateOutput
    = (ZGiven) inputs_.get(ZLocator.fromString("name?"));
  newBirthdays = currentBirthdays;
  newKnown = currentKnown;
  if (currentKnown.contains(nameInput)) {
    for (Iterator iter = currentBirthdays.iterator();
         iter.hasNext();) {
      ZTuple t = (ZTuple) iter.next();
      if (t.get(0).equals(nameInput)) {
        dateOutput = (ZGiven) t.get(1);
        break;
      }
    }
    resultOutput = ok;
  } else {
    dateOutput = null;
    resultOutput = not_known;
  };
```

```
      newResultsM.put("date!", dateOutput);
    } else if ("Remind".equals(schemaName) ||
               "RRemind".equals(schemaName)) {
      final ZGiven dateInput
        = (ZGiven) inputs_.get(ZLocator.fromString("today?"));
      final ZSet namesOutput;
      newBirthdays = currentBirthdays;
      newKnown = currentKnown;
      Set s = new HashSet();
      for (Iterator iter = currentBirthdays.iterator();
           iter.hasNext();) {
        ZTuple t = (ZTuple) iter.next();
        if (t.get(1).equals(dateInput))
          s.add(t.get(0));
      };
      namesOutput = new ZSet(s);
      newResultsM.put("cards!", namesOutput);
      if ("RRemind".equals(schemaName)) resultOutput = ok;
      else resultOutput = null;
    } else {
      throw new Error("Error: Tried to run schema that isn't in "
                      +"birthday book!");
    };
    newResultsM.put("birthday", currentBirthdays);
    newResultsM.put("birthday'", newBirthdays);
    newResultsM.put("known", currentKnown);
    newResultsM.put("known'", newKnown);
    if (resultOutput != null)
      newResultsM.put("result!", resultOutput);
    newResults = new ZBinding(newResultsM);
    SolutionSet newSolutionSet = new SolutionSet(schemaName,
                                                 newResults);
    solutionSets
      = new Vector(solutionSets.subList(0,
                                        currentSolution.nextIndex()
                                        + 1));
    solutionSets.add(newSolutionSet);
    currentSolution = solutionSets.listIterator(solutionSets.size()
                                                - 1);
    System.err.println("Schema completed: " + schemaName);
    propertyChangeSupport.firePropertyChange("currentSolutionSet",
                                             null, null);
    propertyChangeSupport.firePropertyChange("currentSolution", null,
                                             null);
  };
};
```

Appendix B

An Early UML Diagram

This is one of the first attempts at a design for Gaffe. Differences to note include: Using custom 'plug-in' classes instead of beans, the animator core is split into two classes (See Section 10.9), the animator core is responsible for presenting animation methods (delegated to the history), the history can be saved (this is still going to be added, but in a future version of Gaffe), combined animator and designer instead of separate programs; later versions introduced `Form`, though before JavaBeans were decided on it was accompanied by '`FormDesign`'. All of the subclasses of `ZValue` were present in this early design, but they have been left out of this diagram, as they do not differ from the current design.

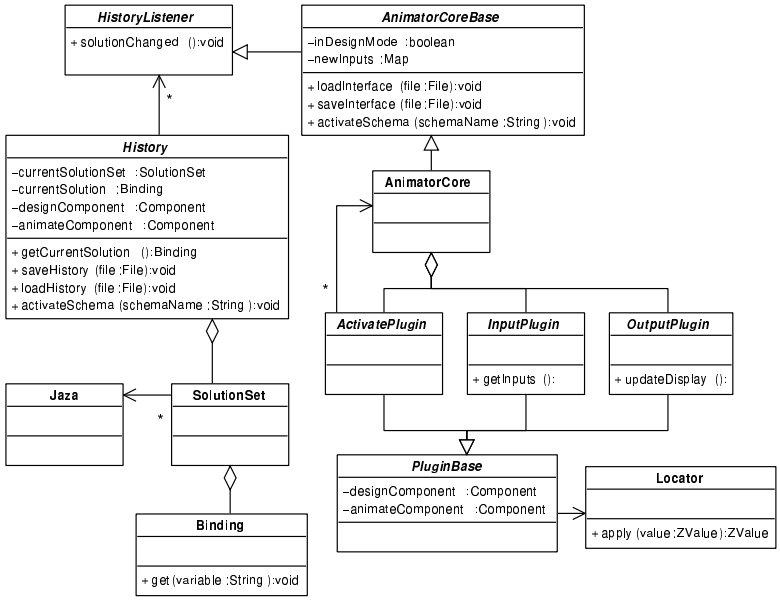

Figure B.1: An Early Design for Gaffe.

Appendix C

Usability Study Booklet

The following pages are a copy of the booklet given to participants in the usability study (Chapter 9). It contains a brief introduction to the purpose and procedure of the study, a set of exercises, and questionnaires for before and after the exercises.

Appendix D

User Manual from the Usability Study

The following pages are a copy of the user manual, that participants in the usability study (Chapter 9) had access to during sessions.