1 Introduction

As multimedia, on-line service are getting more and more popular today, problems like how to easily define a good interactive interface to access information at remote servers with reusable and customizable components are becoming very urgent to be solved.

This is where JavaBeans come into the picture, a simple collection of issues which might be helpful to decide whether it's suitable to use the Beans includes: if this component is suitable to customize with graphical builder tool under different circumstances, and if there are a lot of events handling, states changing involved.

Lots of people share the opinion that to construct Graphic User Interface, GUI, from scratch is usually a quite time consuming task. But if we take a close look at them, it's not hard to see that all components of GUI might appear or act differently, but they do have a lot in common.

The purpose of our project is to develop a code generator, which allows to produce the Java source code according to the Interface Definition Language, IDL, file given by the user. The code generated from the GUI generator will be JavaBeans compliant.

IDL is used to define the interface from client to Common Object Request Broker Architecture, CORBA, server. The IDL file is compiled which produces Java code for the CORBA communication code.

As well as the IDL defines the interface to the CORBA server IDL also implicitly defines the user interface of the client, therefore the generation of the client code for the GUI parts should also be possible.

The generator we will develop should produce Java User Interface code for accessing required attributes, such as a user interface to access a person's "id" and "address", so "id" and "address" are required attributes.

So first of all, we need set up a standard Java pattern file for User Interface to decide what kinds of components we need for each attribute, what properties those components have, and what events will happen in User Interface. Then from this pattern file, we summary those common parts of code for each type attribute, and build up a database of pattern file.

After that, we need to define the GUI mapping between Java and IDL file, both in the component part and the event handling part. Based on the GUI mapping, we define how IDL file should look like to be used as our input file to generator.

The code generator should take the IDL file(s) as definition and use the pattern files from the database to produce the JavaBeans code.

The remainder of the report is structured as follows:

Chapter 2 "Related Work", includes all the studyings we did before and during the thesis project's process, which focuses on 4 parts, Java, JavaBeans, CORBA and IDL and compiler knowledge.

Chapter 3 "JavaBeans Design", presents the requirements and the result of JavaBeans Design, the problems we got and the solution we made are also discussed here.

Chapter 4 "IDL And Java Mapping", defines the mapping between IDL and JavaBeans.

Chapter 5 "The Code Generator", details about parser structure and how we create JavaBeans user interface code from IDL.

Chapter 6 "Conclusion", a summary conclusion made from the JavaBeans design and Code Generator design.

Chapter 7 "The Future", a future improvement of this Generator and applications.

2 Related Work

This chapter introduces all the studyings we have done before and during the project's process, which mainly focuses on four parts, Java, JavaBeans, CORBA and IDL and compiler knowledge.

2.1 Overview Of Java

2.1.1 Java

There are several reasons for why we use Java as the implementation language, and some of them are: simple and powerful, object oriented, and architecture neutral [2].

The flexibility of Java is enhanced by its foundation on an object-oriented paradigm. Java supplies all the advantages of object-oriented programming by supporting such features as encapsulation and inheritance. One will be forced to write program in a object-oriented way if he uses Java. Once the basic concepts of object-oriented programming are understood, any one can learn to program in Java very quickly.

Java also supports dynamic downloading of application components in a distributed system. As such, Java overcomes many of the important technical challenges posted by the diversity of underlying hardware and software technologies.

Applets are in fact small applications, which are supported by Internet servers, they can be transported over the net, automatically installed, and run as a part of a web document. Applets can even send messages to other applets, working in groups. At this moment, the most popular browsers which support Java Applets are Netscape Communicator, Microsoft Internet Explore and HotJava.

It's fun to write programs in Java, with the packages Java provides, one can develop very quickly and get satisfying results without a lot of pain. The standard compiler "javac" always gives accurate, sensible error reports to help you detect the bugs and eventually correct them [6].

But the most important strength of Java language might very well be the ability it has to be totally hardware independent. Today, operating systems, processor and core systems are upgraded so quickly, one cannot even be sure that today's program can be run tomorrow, even on the very same machine. But Java's "write once, run anywhere, anytime, forever" philosophy makes sure that you will not have to face this situation, with other words, these applications can run anywhere there a Java Virtual Machine has been set up.

As the requirements on powerful programs continues to grow, Java is uniquely suited to build the next generation of applications. Java provides solutions to a number of tricky issues, such as platform independence, security [7].

It might take a while for one to get used to all the Java packages at the very beginning, but it's worth it, since once you start taking advantage of Java's amazing features, the possibilities are limited only by your own imagination.

2.1.2 JDK 1.1.3

We use Java Development Kit 1.1.3, JDK 1.1.3. which includes improvements in functionality, performance, and quality over JDK 1.0.2, but still there are some bugs in it.

JDK 1.1.3 offers new capabilities, one of them is JavaBeans package. The JavaBeans APIs, Application Programming Interface, define a software component model for Java allowing third party Independent Software Venders, ISVs, to create and ship Java components that can be composed together into applications by end users.

Another new capability is the AWT, Abstract Window Toolkit, enhancements. Among those enhancements, we take advantage of the new delegation-based event model in AWT in our project, which JDK 1.1 introduces in order to provide a more robust framework to support more complex Java programs.

Delegation Model Overview Event types are encapsulated in a class hierarchy rooted at java.util.EventObject. The whole event handling process consists of three parts, Source, Event, and Listener. "Listener" should register itself to "Source", then an "Event" will be propagated from "Source" to "Listener" by invoking a method on the listener, e.g. actionPerformed, and passing in the instance of the event.

A Listener is an object that implements a specific EventListener interface extended from the generic java.util.EventListener. An EventListener interface defines one or more methods which are to be invoked by the event source in response to each specific event type handled by the interface. An Event Source is an object which originates or "fires" events.

In an AWT program, the event source is typically a GUI component, e.g. textField, and the listener is commonly an "adapter" object which implements the appropriate listener (or set of listeners) in order for an application to control the flow/handling of events.

Additionally, the AWT in 1.1 will support the JavaBeans architecture. In fact, all AWT components in 1.1 are "simple" JavaBeans.

2.2 JavaBeans

2.2.1 What Is JavaBeans

JavaBeans is the platform-neutral, component architecture for Java [1]. In fact it is the only component architecture in Java development environment. JavaBeans extends Java's "Write Once, Run Anywhere" capability to reusable, component development. So JavaBeans code can run on every OS and also within any application environment.

A JavaBean is a reusable software component that can be combined with other components to build an application with the aid of the application builder tool (see Fig. 2.1.).

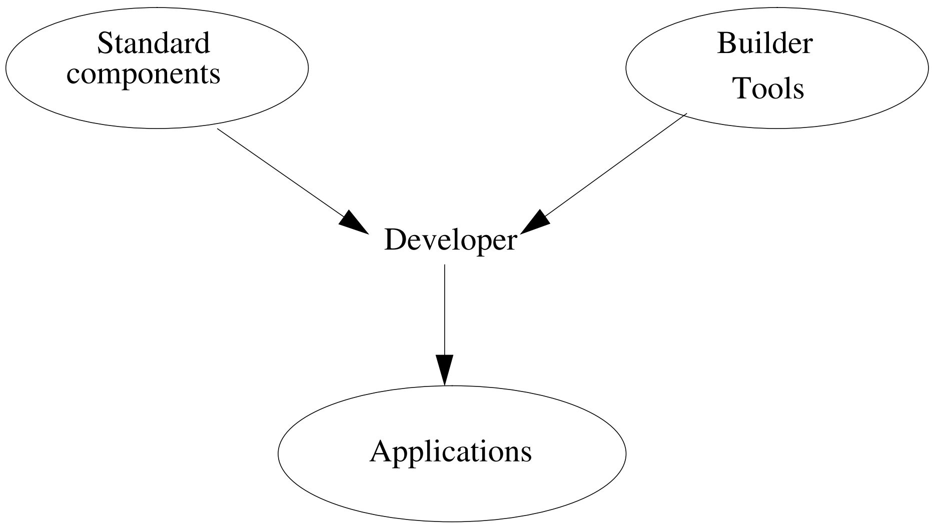


Fig 2.1. *Different Standard software components can be combined together to build an application using Builder Tools.*

And the most important features of a Java Bean are the set of *properties* it exposes, the set of *methods* it allows other components to call, and the set of *events* it fires. Those features are used to connect JavaBeans together [3].

The fundamental JavaBeans parts are Software component, Builder tool, and Visual manipulation. If we take the beans as raw material, the builder tool as a wok, and visually manipulating beans as the cooking process, now, what do we have here then? Voila, a delicious bean dish!

2.2.2 Basic Beans Concepts

Individual JavaBeans will and should vary in functionality, but the typical unifying features of JavaBeans are introspection, customization, events, properties and persistence [3].

Introspection:
Introspection means the process that builder tools figure out which properties, events and methods a JavaBeans supports. The BeanInfo class is used to supply the information about bean's behaviour to builder tool. The Introspector class is provided to allow application builder tool to analyse beans and understand the various design patterns and standard interfaces and provides a uniform way of introspecting on different beans.

Customization:
In application builder tool, users are able to customize the appearance and behaviour of the various bean they are using to compose an application. For simple and medium size beans, application builder can use properties of a bean to construct a GUI property sheet and provides a property editor to edit properties of the bean. For large size JavaBeans, the Customizer class controls the customization of the bean.

Events:
Event notification means a component can notify interested parties when something of interest happens to it (see Fig. 2.2.). Typically an event fired by a component represents either its action or change on one of its properties.

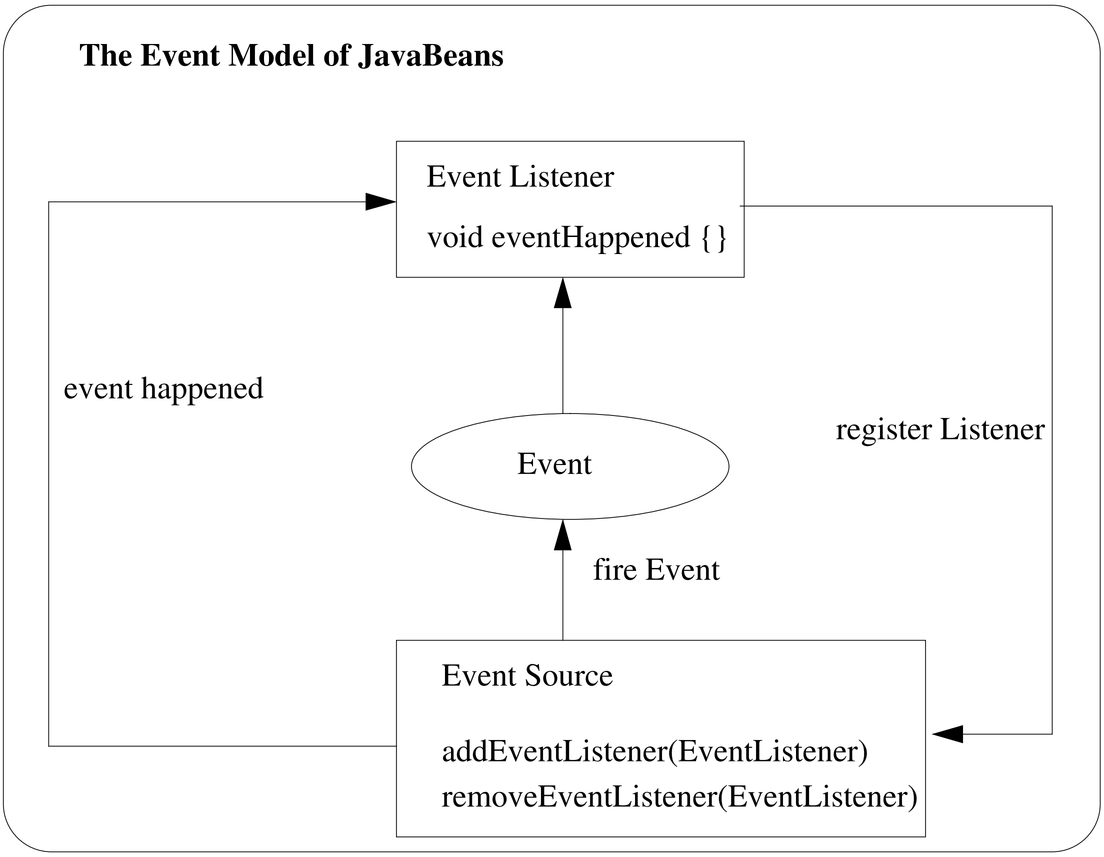


Fig. 2.2. *Event Model of JavaBeans. Event Listeners are registered to Event Source by addEventListener, or removed by removeEventListener. When an Event is fired by Event Source, Event-Listeners will reaction with method "eventHappened".*

Properties:
Properties represents the named attributes of a JavaBean which can affect its appearance or it behaviour. Properties can be accessed by other components calling their “get” and “set” methods. As part of the processes of customizing JavaBeans, certain properties are presented in a property sheet for user.

There are two special kinds of properties which are *bound properties* and *constrained properties.* When a bound property is changed, it fire a change notification to some or all of its property change listeners, in this way, they allow other components to bind special behaviour to property changes. Constrained properties allow other bean to validate the change, and reject it if it is inappropriate.

Persistence:
Beans persistence is supported by Java Object Serialization mechanism which will

automatically store away the internal state of JavaBeans. Also JavaBeans can choose to "externalization" stream mechanism that provides it with full control over the data layout.

2.2.3 Why JavaBeans

As a matter of fact, the concept of beans is not a stranger for us, builder tools like Delphi, Visual Basic, have already somehow given the notion of a bean. The basic idea about component architecture is almost the same, just the programming languages might be different. Here comes the big question, why JavaBeans? The answer is the unique platform independent character of Java Language, with other words, "*write once, use everywhere.*"

Now the question is, why a component architecture is used in Java? With the component development, users are turning to creating reusable components rather than monolithic applications to free themselves from slow, expensive application development, and to build up a portable, reusable code base [4].

In our project, we use JavaBeans to implement GUI component on client side of CORBA IDL, because JavaBeans integrate so well with CORBA IDL with its distributed character. Beans can work with any network model, and CORBA IDL is great for interoperatibility, which provide heterogeneous distributed computing with platform independent components [17].

2.2.4 JavaBeans Builder Tools

An application builder tool is a GUI-based application development environment to layout, customize and connect components in a container [5].

We paid a quick visit to some of the existing and coming development tools for Java Beans, among them we found some quite interesting.

For instance, **Java Workshop**, a product from Sunsoft, is heading to the next-generation which includes features to support editor syntax highlighting, remote debugging, integration of Sun's high performance JIT technology, etc [12].

Another example is **Borland JBuilder**. The unique DataDirector architecture, which consists of powerful JavaBeans database components, delivers everything you need to build scalable data-centric applications [13].

Visual Cafe, from Symantec Co., provides a total integration between visual tools and the underlying source code, including true two-way tools and WYSIWYG form designer [14].

Visual JavaScript, a next-generation Netscape development tool, allows developers to access native component models such as ActiveX through JavaBeans. [15]

In our project, we choose to use **BeanBox** [1] to test our JavaBeans components. As our main task is to create an IDL to JavaBeans generator, we don't need advanced builder tool to build application with JavaBeans. BeanBox is a very simple test container, it satisfies our requirement to test our own Beans. Fig. 2.3. shows our sample Bean in BeanBox.

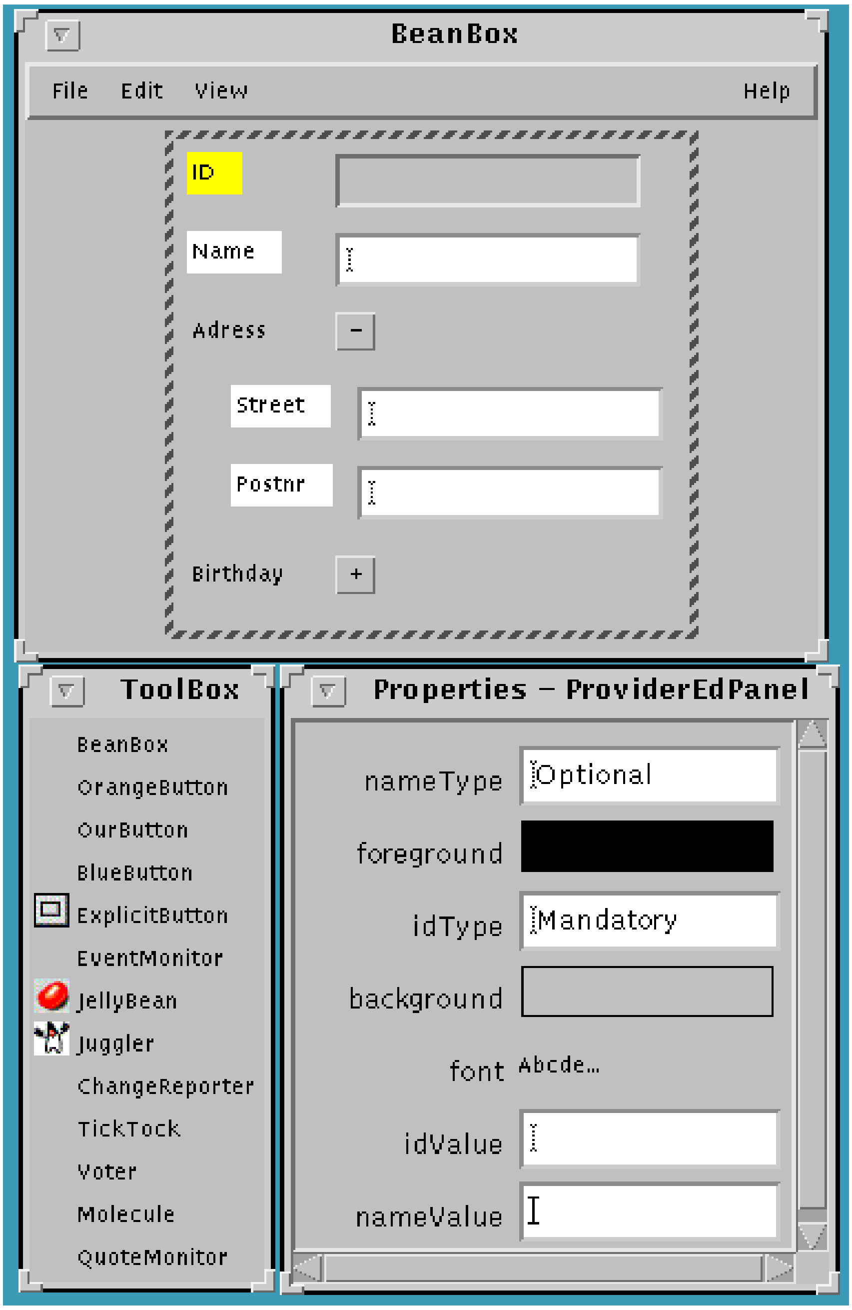


Fig. 2.3. *Our Sample Bean Displayed in BeanBox. There are three windows in BeanBox, the upper one is the main container window where the Bean is visually placed and manipulated, the left one is the ToolBox window which lists all the Beans registered for use with the BeanBox, the right one is the PropertySheet window which provides a means of editing the individual properties of the Bean.*

2.3 The CORBA And IDL

2.3.1 What Is CORBA

Building distributed applications requires a proven infrastructure - one that scales to handle the diversity of the enterprise or the Internet for that matter. Common Object Request Broker Architecture, CORBA, provides that infrastructure - allowing you to build standards-based systems from components implemented in a variety of languages and running on a collection of different platforms.

CORBA also guarantees total interoperability between these components through use of a standardized wire-protocol - the Internet Inter-ORB Protocol (IIOP), (see Fig.2.4.).

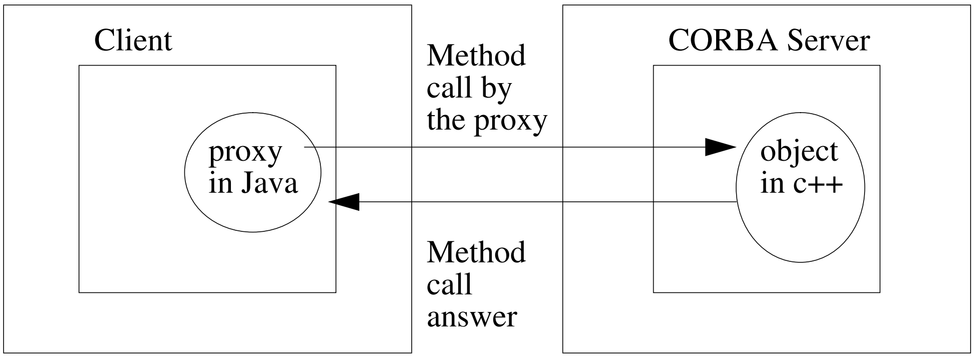


Fig. 2.4. *Based on CORBA, a Java proxy client can communicate with a C++ server.*

2.3.2 CORBA To Java

The CORBA to Java mechanism allows your Java Applets and applications to be plugged into the CORBA infrastructure and thereby allows them to interoperate with both Java and non-Java applications and components which might even allocated over the whole internet. A full and complete development and runtime environment is provided that allows building both client and server Java CORBA components [15].

The outcome code and runtime classes are pure Java which means that any Java development environment can be used to build up the applets and applications.

2.3.3 Available Products

So far, the most widely used CORBA product is called Web ORBIX, from IONA, all it requires to run an application is only a Java-enabled browser on the machine. OrbixWeb is at the leading edge of Java - the first Java ORB, which provides full support for JDK 1.1 and even JDK 1.0.2, Sun's latest Java programming platform [16].

2.3.4 IDL Fundamentals

The purpose of an IDL, Interface Definition Language, is mainly to define an interface between client and server processes so that they can communicate with each other at a level higher than simple byte strings in the computing environments.

The following simple example shows how a IDL file for Date might look like:

```
/*a sample
*/
        struct Date {
        //attribute
        string year, month, day;
        };
```

Another thing which also goes along with the IDL file is a code generator or so called a simple compiler. The compiler or the generator takes as its input an IDL file and generates code. The output are usually Java, C/C++ code. The generator could also implement the stubs and one or more mechanisms to set up the connection between client and server at the same time.

One good example is, AccordSTAR Generator uses C Language itself as its IDL, known as CIDL. Given that C language is a standard, then it follows that CIDL, which is a pure subset of is C.

2.4 The Phases Of A Compiler

What a normal compiler does is to translate high level language programs into executable code. In order to achieve this, a modern compiler often passes through many phases, from breaking down the sentences to finally assembling the machine code.

Obviously, this kind of compiler structure shows the strength of being flexible, in our case, the task is to break down and analyse a IDL program and map it into a JavaBeans like structure later on, so only the first couple of phases of a complete compiler would be needed here.

Compilers might be used for different purposes, but they do have a lot of things in common [18].

- Lex, break the source code into individual words, or tokens.
- Parse, analyse the phrase structure of the program.
- Abstract Syntax, build a piece of *abstract syntax tree* corresponding to each phrase.
- Semantic Analysis, determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase.
- Frame Layout, place variables, function parameters, etc, into activation records (stack frames) in a machine-dependant way.
- Translate, produce *intermediate language trees* (IR trees) [19][20].

3. JavaBeans Design

Before we started with the code generator, we build up a model JavaBeans which gave us a brief understanding about how the result should look like. This chapter presents the requirements, the final result of our design of the model JavaBean User Interface. The problems we got and the solutions we made are also discussed here. We use BeanBox as our development tool.

3.1 Requirements

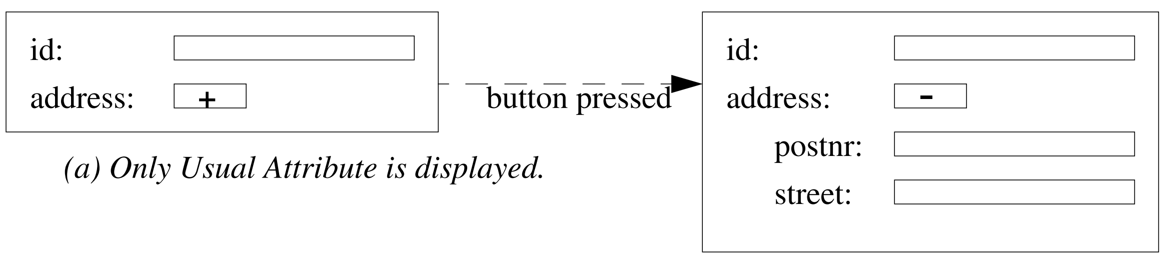
3.1.1 Attributes

In our project, attributes are categorized into two major kinds, one is usual attribute, and the other is extensible attribute.

Usual attribute is just a normal attribute which is presented directly on user interface. It can be read-only or normal (readable and writable), mandatory or optional.

Extensible attribute is actually an abstract presentation for its further detail sub-attributes. Extensible attribute is default represented on user interface without any detail attributes. Those sub-attributes of extensible attribute, which in turn have the same property as usual attributes can be displayed on demand from user. This facility can save space on user interface.

For example, "id" is a usual attribute, and "address" is an extensible attribute which has sub-attributes "postnr" and "street" (see Fig. 3.1.).



(a) Only Usual Attribute is displayed.

(b) Usual Attribute and Extensible Attribute are displayed.

Fig. 3.1. *The Layout of Usual Attribute and Extensible Attribute in User Interface. "id" is a usual attribute, so it is displayed on user interface initially, "address" is an extensible attribute, so its sub-attributes, "postnr" and "street" are not initially displayed, only when button is pressed, those sub-attributes are displayed.*

3.1.2 Events

The key feature in GUI is event handling [3]. In our project, there are two types of events, *internal events* and *propagated events.* Internal event is bounded internally and is not propated outside, e.g. for those extensible attributes, when button of "address" is pressed, "postnr" and "street" will show up. Propagated event is to fire event out and notify outside event listeners to make certain reaction, e.g. start or stop doing something.

3.1.3 Methods

Methods exported from JavaBeans are normal Java methods which can be called from other components [4]. Those methods we require in our JavaBeans design are methods to "get" and "set" the properties and values of attribute, e.g. read-only/normal, mandatory/optional. Also we need methods which can be called to register other components as event listeners.

3.2 Result

3.2.1 Package Structure

In order to supply Beans information to builder tools and let them be aware of which properties, methods and events our JavaBeans support, we have BeanInfo class for introspection [5]. For example, (see Fig. 3.2.), BeanBox knows that our only event notification to other components is "keyReleased".

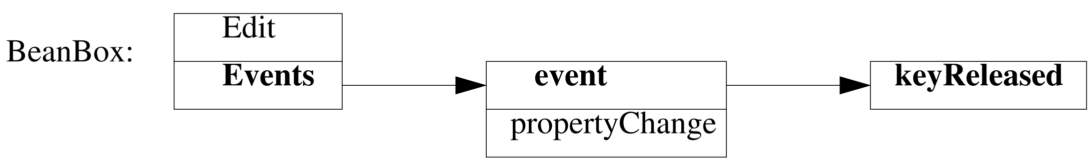
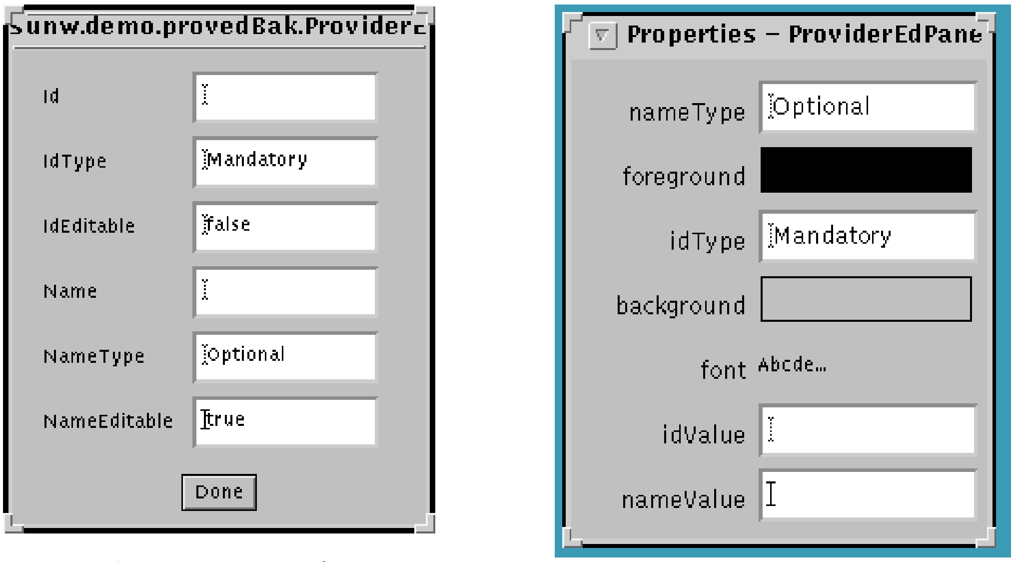


Fig. 3.2. *BeanBox's Event Handling. On BeanBox's "Edit" menu, there are two kinds of "Events", "event" and "propertyChange". The only "event" is "keyReleased".*

For the purpose of Customization, our JavaBeans provide a Customizer class by which user can customize the appearance and behaviour of our Bean in builder tools to compose application [5]. So builder tools can use properties of a Bean to construct a property editor and provide property sheet to edit the Bean properties (see Fig. 3.3.).



(a). Customizer Window

(b). Property Sheet Window

Fig. 3.3. *Customizer Window(left) and Property Sheet Window(right). The Customizer window is used for customizing the appearance and behaviour of the Bean to compose application. The Property Sheet window is used for editing the properties of the Bean.*

The whole layout and events handling are defined in the main class. In addition, we have also created a PanelListener class which is an abstract interface extends EventListener for all listeners of our Bean. Fig. 3.4. shows the whole structure of our Bean Package.

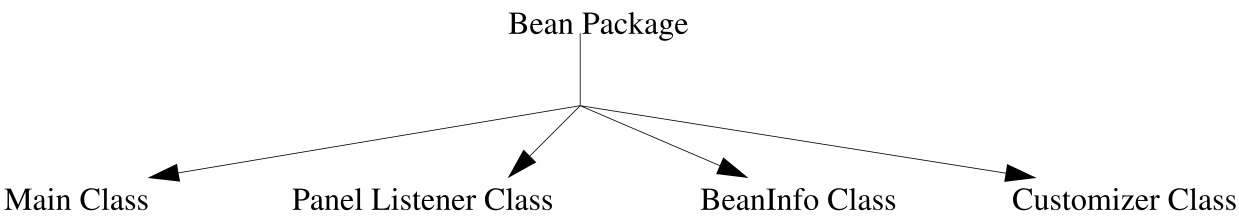


Fig. 3.4. *Bean Package Structure. There are four classes in our Bean Package. Main Class defines the whole layout and events handling of our Bean, Panel Listener Class is an abstract interface for all listeners of our Bean, BeanInfo Class provides the information that is needed to integrate a Bean into a large system, Customizer Class offers GUI-based support for customizing the facilities of a Bean.*

3.2.2 GUI Layout

Our GUI design for JavaBean is quite simple. A usual attribute is represented with Label and TextField. For extensible attribute, it is built up with Label and Button, when control button is pressed, sub-attributes will show up.

For read-only attribute, its textfield is non-editable, for normal attribute, we set textField editable, mandatory attribute's label has yellow colour as background, and for optional attributes we set background colour to white.

In the whole view of user interface, we use GridBagLayout as Layout Manager to constrain every component to certain position. To construct extensible attribute, we use our own modified CardLayout to manage the appearance of one empty panel and one detail panel, therefore it saves the space when sub-attributes are not required. When the button of extensible attribute is pressed, the detail panel is shown up with all sub-attributes.

Fig. 3.5. shows the GUI Layout of our Bean.

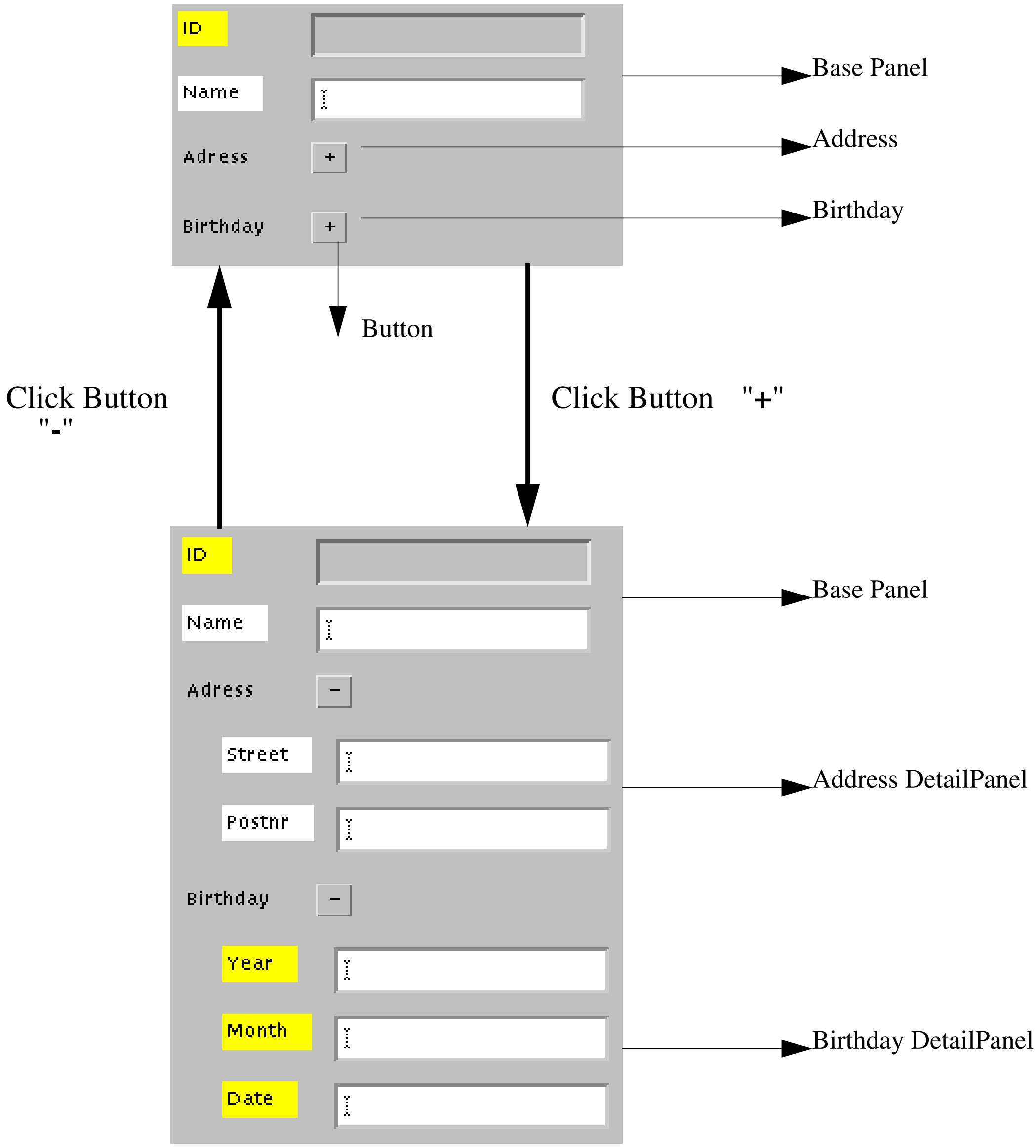


***Fig. 3.5.** GUI Layout of Our Bean. The upper window is the default layout when the Bean is initialized, buttons' status are "+", and all sub-attributes of extensible attributes are not displayed. The below window is the layout after buttons are pressed, after that, buttons' status are "-", and those detail panels with all sub-attributes is displayed by the CardLayout.*

3.2.3 Properties

Each usual attribute has two properties and one value, which correspond to the appearance of those graphic components.

If we have two attributes, one is read-only, mandatory "id", with value "7094", and the other is normal, optional "name", with value "Johanson", (see Table. 3.1.).

attribute	value	readonly /normal	TextField (Editable /non)	mandatory /optional	Label Background Colour
id	7094	readonly	non	mandatory	yellow
name	Johanson	normal	Editable	optional	white

Table. 3.1. *Properties of Attributes, "id" and "name". "id" is a read-only attribute, so its TextField is non-editable. "id" is also mandatory, so its Label Background colour is "yellow". "id" has value "7094", so the TextField shows the value "7094". "name" is a normal attribute, so its TextField is editable. "name" is also optional, so its Label Background colour is "white". "name" has value "Johanson", so its TextField shows the value "Johanson".*

Properties of attributes can be edited in Customize Window of BeanBox (see Fig. 3.3.a). They can be accessed and updated by "get" and "set" method. The values and properties of attributes are listed in Property Sheet window. (see Fig. 3.3.b).

3.2.4 Event Handling

There are two kinds of events in our Bean. One is inside event, for example in Fig. 3.6., the extensible attribute's button is pressed, to control the detail Panel to show up. The other is (see Fig. 3.7.) keyReleased on main Panel, Label or TextField, our Bean will notify other components that are listeners to our Bean, and cause appropriate reaction in these components.

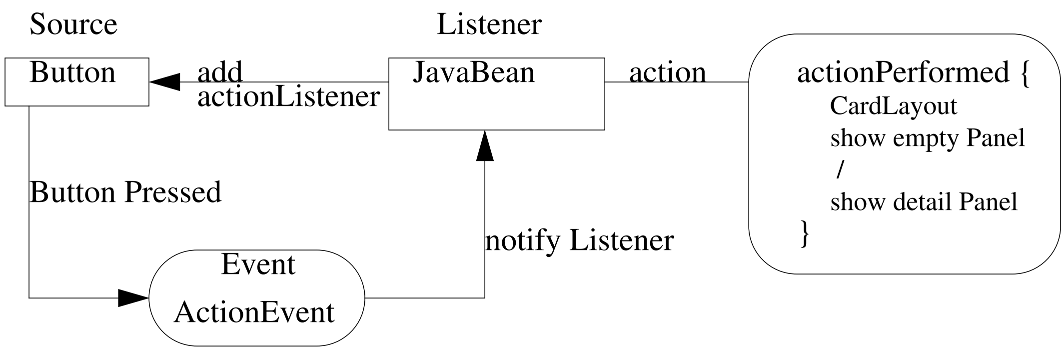


Fig. 3.6. *Inside Event to Control CardLayout. When the Button of the extensible attribute is pressed, ActionEvent is fired and the JavaBean which is Button's actionListener will react with actionPerformed method.*

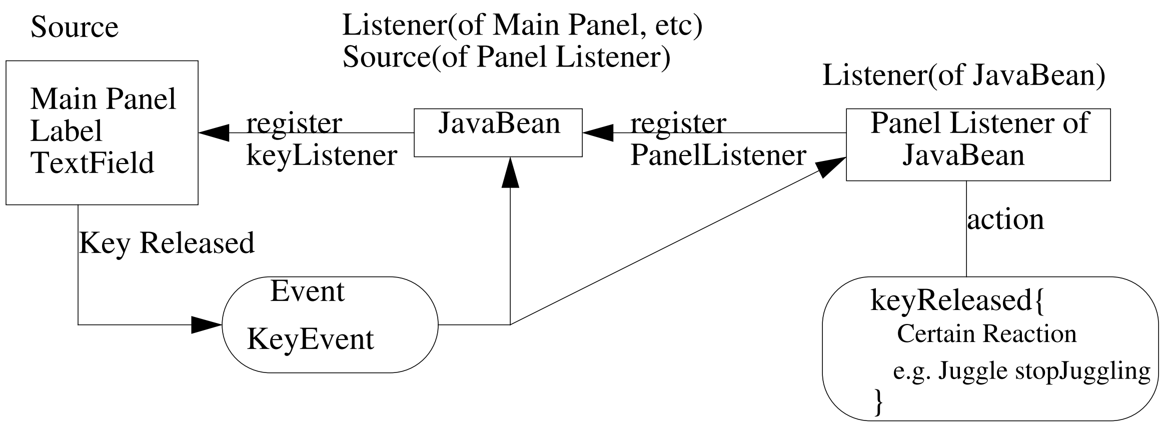


Fig. 3.7. *Event Notification to Our Bean's Listener. When any key is released on Main Panel, Label, or TextField, a KeyEvent will be fired, our Bean, which is the KeyListener of Panel, Label or TextField, will send this KeyEvent to all the listeners of our Bean.*

3.3 Problems And Solutions

In the process of programming JavaBeans, we met two problems. One is how to let Panel generate the Key Released event, and the other is how to justify the size of CardLayout's sub Panel. And we have solved those problems in some efficient ways.

3.3.1 Panel Notify

The problem is that JavaBeans can hear all key Event on Label, TextField, but not Panel. What causes this is that Canvases and Panel, by default, do not generate key Events because they do not request Focus. So we need to call requestFocus() [10] on Panel, then it will generate key events to its listener.

To solve this problem, we bring in mouseListener to JavaBeans, when mouse is clicked first in Panel, we call requestFocus() to get Focus on Panel, so keyReleased event can be generated by Panel when key is released on Panel.

Below is the section of code which requestFocus on Panel:

```
public ProviderEdPanel() {
        super();
        addKeyListener(this);
        addMouseListener(new MyMouseAdapter());
            ................
  }

// Get Focus on Panel through mouseClicked
class MyMouseAdapter extends MouseAdapter
{
    public void mouseClicked(MouseEvent evt){
            requestFocus();        // get Focus on Panel
    }
}
```

3.3.2 CardLayout

CardLayout in JDK 1.1.3 manages cards of the same size [10]. But what we need in project is to handle the cards with different sizes. There are two cards in CardLayout, one is an empty Panel, the other is our detail Panel, which includes several subcomponents. So the size of the empty Panel should be height 0, and the detail Panel should have its own height > 0.

We modified CardLayout and create our own ResizingCardLayout, which changes the size of cards while triggering them (see Fig. 3.8.).

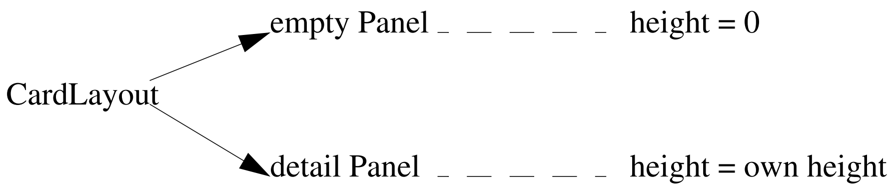


Fig. 3.8. *CardLayout Control Card's Height. When the empty Panel should be displayed, the height of the card is set to 0. When the detail Panel should be displayed, the height of the card is set to the detail Panel's own height.*

Here is the source code section.

```
 // For extensible attribute address, when the button is pressed
public void actionPerformed(ActionEvent evt) {
        if(evt.getSource() == adButton) {
              if((getAdValue()).equals(EMPTY)) {
                    setAdValue(DETAIL);
               ResizingCardLayout layout = (ResizingCardLayout) adDetail.getLayout();
                    layout.show(adDetail, DETAIL, false);
                    getParent().validate();
              } else {
                    setAdValue(EMPTY);
               ResizingCardLayout layout = (ResizingCardLayout) adDetail.getLayout();
                    layout.show(adDetail, EMPTY, true);
                    getParent().validate();
              }
        }
        ..........................
 }

// Modified CardLayout
class ResizingCardLayout extends CardLayout
{
        private boolean height0 = true;

        public void show(Container parent, String name) {
                    this.show(parent, name, false);
        }

        public void show(Container parent, String name, boolean height0) {
                    this.height0 = height0;
                    super.show(parent, name);
        }
```

```
    public Dimension preferredLayoutSize(Container parent) {
            Dimension di = super.preferredLayoutSize(parent);
            if (height0 == true)
                return new Dimension(di.width, 0);
            return di;
    }

    public Dimension minimumLayoutSize(Container parent) {
            Dimension di = super.minimumLayoutSize(parent);
            if (height0 == true)
                return new Dimension(di.width, 0);
            return di;
    }
}
```

4 IDL And Java Mapping

This chapter is mainly focused on how we set up the mapping between IDL interface specification and Java GUI. This mapping is used to design the code generator.

4.1 The Integration

Now it's clear that what we really need here is a standard which should be clean, tightly specified interfaces for objects (services, components, etc...) available, and these interfaces can be used between different types of software, or client and server.

Here is where IDL comes into the picture, IDL describes interfaces only, not implementations; in other words, it is not a programming language. It is a programming-language-independent way of describing interfaces to objects, and is similar in syntax to Java and C/C++.

Following are some reasons for why the integration between IDL and Java should work out in a proper way:

IDL modules are like Java packages.
IDL interfaces are like Java interfaces.
IDL operations are like Java methods.
IDL contains several built-in data types and facilities for creating your own.

IDL is mapped into each programming language to provide natural access to object interfaces. The IDL definitions can be either compiled by using a standard IDL compiler or used as data source for the Java code generator [21].

4.2 The Mapping

The IDL to Java mapping might be quite different, depending on the purpose. In our project, only the GUI part of mapping concerns us. The GUI mapping consists of two parts: **Component mapping** and **Event mapping**.

4.2.1 Component Mapping

For Component mapping, we have two types of data in IDL, they are String and Struct. String is mapped to one Label and one TextField.

```
IDL:     string street;
Java:    Label street;
         TextField streetField;
```

Also we have two kinds of comments for String in IDL which distinguish two kinds of attributes, one is comment "readonly attribute", the other is "attribute". Fig. 4.1. shows the mapping between a single attribute called "street" and it's corresponding Java GUI. According to this difference, we have different properties for our Java components.

For "readonly attribute" comments in IDL file, we define the mapping TextField's not editable. For "attribute" comments in IDL file, we define the mapping TextField editable.

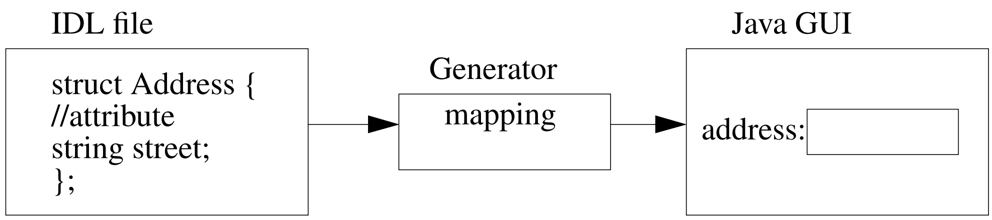


Fig. 4.1. *The Code Generator performs the mapping between IDL attribute and Java GUI.*

There are two kinds of Structs, one is the independent Struct, which is not included in any other Structs, the other is Substruct, which is used as a part in other Structs.

For the independent Struct, we map it to one Panel in Java.

```
IDL:     struct Address {
             //attribute
             string street;
         };

Java:    Panel p;
         Label street;
         TextField streetField;
         p.add(street);
         p.add(streetField);
```

For the Substruct, we map it to one Label, one Button and one Panel with CardLayout which includes empty Panel or its detail Panel.

```
IDL:     struct Person{
```

```
    //attribute
    string id;
    //structure
    Address address;
};
```

Java:

```
Panel person;
Label street;
TextField streetField;

Label address;
Button addressButton;
Panel addressPanel;
addressPanel.setLayout(CardLayout c);
Panel emptyPanel;
Panel addressDetailPanel;
c.add(emptyPanel);
c.add(addressDetailPanel);

person.add(street);
person.add(streetField);
person.add(address);
person.add(addressButton);
person.add(addressPanel);
```

4.2.2 Event Mapping

The other part of mapping is the Event mapping between IDL and Java. In IDL, if it is a hidden Struct included, its Java Event mapping is to allow the mapping Button for the hidden Struct to be pressed, and the mapping CardLayout Panel will show up one of the included Panels according to the Button's status (Fig. 4.2.).

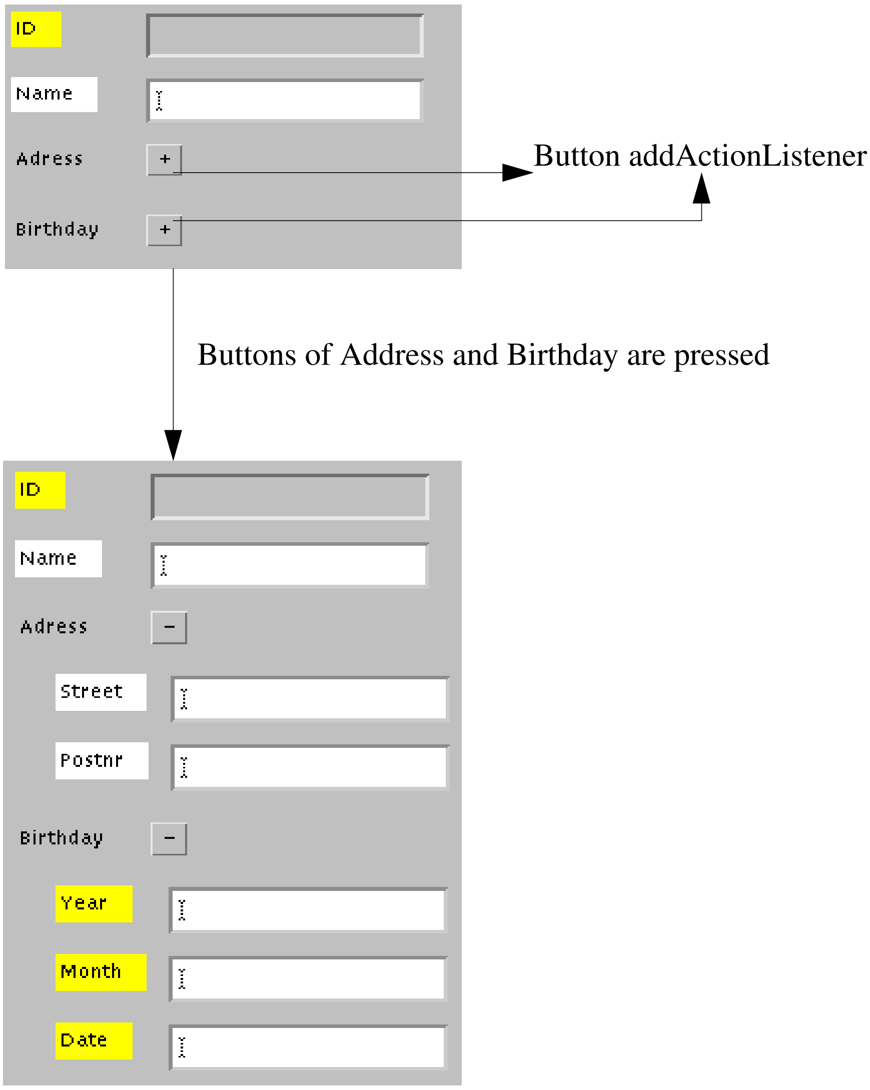


Fig.4.2. *Address and Birthday are Substructs, their buttons are added ActionListener, So when they are pressed from "+" to "-", the Detail-Panel of them are shown up in the CardLayout.*

4.3 What We Have Learned

There are several things we have learned during the mapping process, first of all, before we dive into implementation, it's always a nice and necessary thing to have a sketch over the objects, (or components) we are going to use [23].

Secondary, one should try to discover and use the common parts of the components as much as possible, to try to deal with each single component in a different way is a waste of both time and resource.

One thing we have to make clear here is, only those IDL/Java parts related to our project are discussed in this project, we make no warranties about the suitability of this software.

5 The Code Generator

This chapter explains the structure of compilers and gives the detail description of our code generator.

5.1 Introduction To Compiler And Parser

Over the past decade, there have been several shifts in the way compilers are built. A lot of new programming languages are being used, and among them, object-oriented languages with dynamic methods might very well be the most popular ones. New machines with large register sets and high penalty for memory access along with compiler assistance also contribute to better performance [19].

Fig. 5.1. shows the phases and interfaces of a complete compiler:

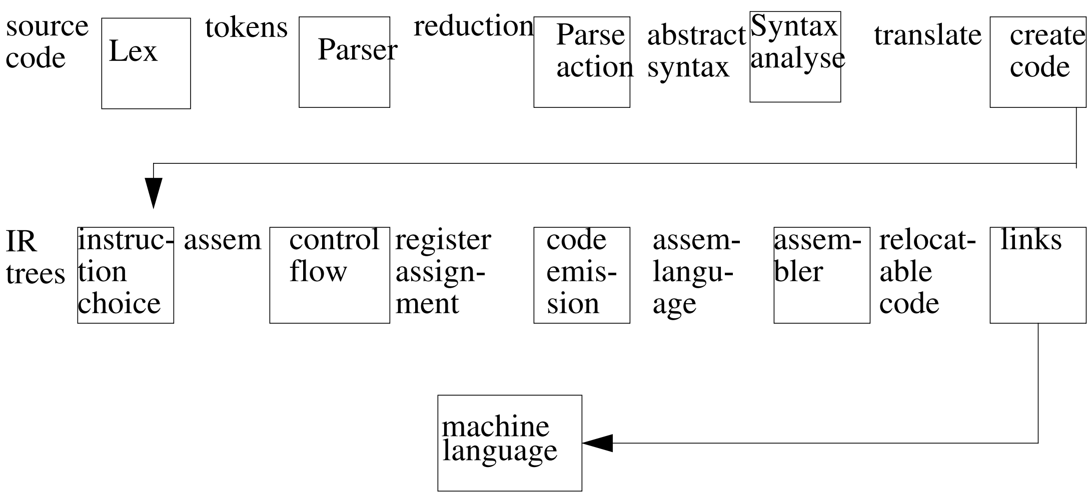


Fig. 5.1. *The steps of a compiler. Starting from Lex, each phase of a compiler processes the source code and then passes it further to the next phase through the interfaces.*

As a matter of fact, this so called code generator should work like a parser, and it operates on an abstract language, IDL, the Interface Definition Language. In this chapter, we will focus on describing the techniques, data structures and Java packages used for the translating between IDL and Java.

The purpose of our project is to provide a parser between IDL files and JavaBeans GUI code, with other words, it should work as a simple compiler, a translate mechanism, (see Fig. 5.2. and Fig. 5.3.):

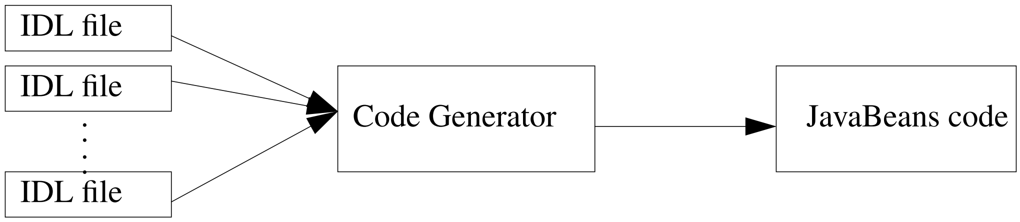


Fig. 5.2. *The Code Generator takes IDL file(s) as input and then produces JavaBeans code.*

Bellow is how the steps involved are organized:

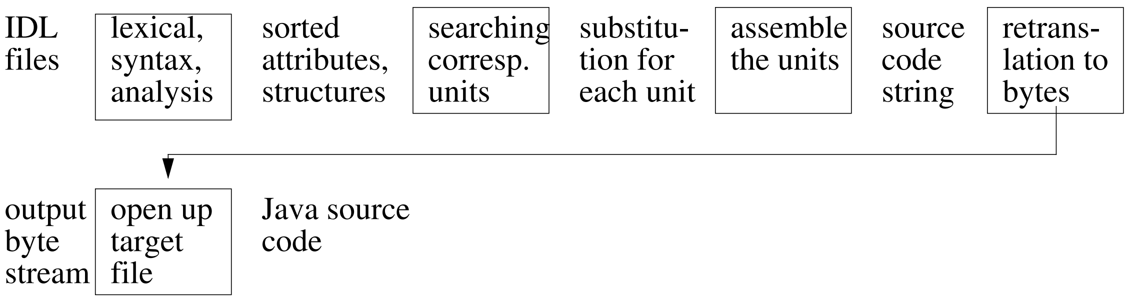


Fig. 5.3. *Phases and interfaces of our parser.*

5.2 Lexical And Syntax Analysis

in order to translate a program from one language into another, in our case, from IDL to Java, what a parser first has to do is to break the program down into small pieces, understand its structure and meaning, save those useful information, then put them together in different ways.

In our project the analysis part is broken up into **Lexical Analysis** and **Syntax Analysis**:

Lexical Analysis: breaking the input IDL code into individual words, “tokens”. The analyser class (or function) takes a stream of characters and produces a stream of tokens.

By using the Java class StringTokenizer [10], the lexical analysis process is not very complicated. Here is a simple example to demonstrate this handy tool:

```
public static void handle_sub_struct(String sub_name) throws Exception {
        InputStream sub_struct_idl = new FileInputStream(sub_name);
        String          sub_struct_idl_ex = to_mystrings(sub_struct_idl);
        StringTokenizer sub_struct_idl_st;
        sub_struct_idl_st = new StringTokenizer(sub_struct_idl_ex, " ,{ }\n\t/");
        analys(sub_struct_idl_st);
}
```

Fig. 5.4. *a StringTokenizer takes sub_string_idl as input, filters away- the characters ",{}\n\t" and has sub_struct_idl_st, a string of tokens, separated with a space as output.*

Fig. 5.4. shows how a text from IDL file is accumulated into a string object which is passed to a StringTokenizer instance to be broken into tokens.

Syntax analysis: detecting the structures of the program, calculating the program’s meaning, retrieving and sorting the information we need for further usage. This is a crucial step, since all the following processing totally rely on this step.

There is an example for sorting different data types. The tokens are classified into three vectors: readonly, attribute and structure.

```
/* using StringTokenizer to create the data flows
*/
public static void analys(StringTokenizer st) throws Exception {
        while(st.hasMoreTokens()) {
                        String k = st.nextToken();
                        int my_op;

                if (k.equals("readonly")){
                insert into readonly vector;
                }else if (k.equals("attribute")){
                insert into attribute vector;
                }else if (k.equals("structure")){
                insert into structure vector;
                }else {
                default processing...;}
        }
}
```

Fig. 5.5. shows how to update the vectors.

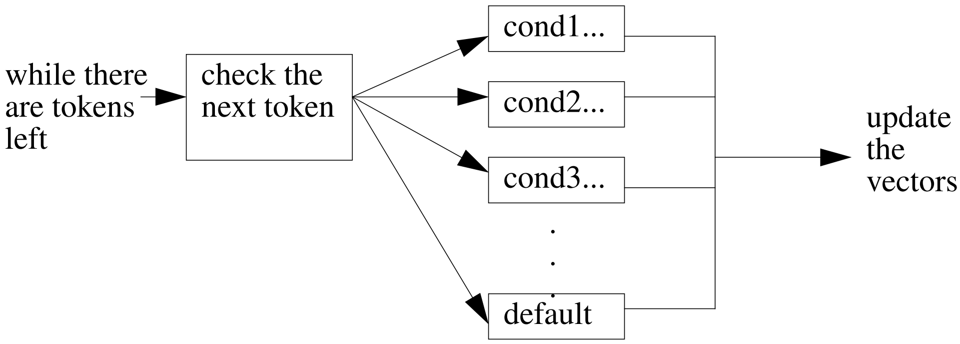


Fig. 5.5. *Each token is checked and classified into different vectors according to the conditions.*

5.3 Searching Database And Producing The Corresponding Code Units

We have set up a database of structure modules, these modules are the common parts of different types of graphical components. These modules will be used to carry out the substitutions. With slight changes, they can be applied on most of the Java GUIs, "write once, run anywhere". The entries to this database are the names of the modules.

When we have got all the data flows, like vectors updated correctly, the next step is to find the corresponding module in the database. The following two pieces of code are responsible for finding the proper module and performing the substitutions, (see Fig. 5.6.).

```
/* responsible for finding patterns and replace them if needed
*/
        public static String my_any_match(String src_string,
        String my_old_val, String my_new_val) throws Exception {
        finding the matching parts...;
        performing the substitutions...;
        return the result;
        }


/* the new code producing function
*/
        public static void my_new_code(String  my_new_string,
        String the_new_pattern, String the_arg_array[]) throws Exception{
```

```
call the my_any_match function...;
append the result to the final string;
}
```

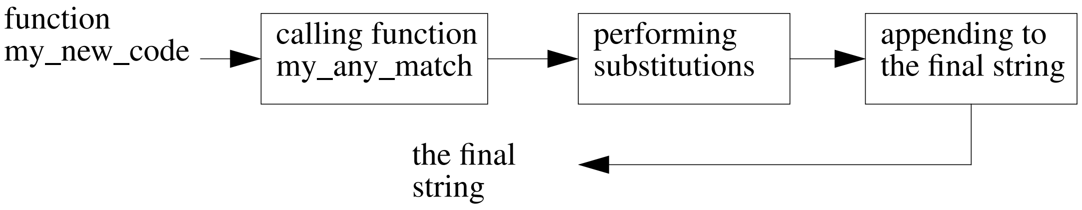


Fig. 5.6. The Code Generator performs the substitution and produces the final string.

5.4 Creating The Target File

Now, with the final string we obtained, we have reached the final step, to create the output stream and target file, (see Fig. 5.7.). In order to achieve that in Java, we'll have to once again use byte stream:

```
/* converting outputStream to byte and write to the output file.
*/
        String final_str;
        byte my_buffer_byte[] = new byte[final_str.length()];
        OutputStream bean_out;
        bean_out = new FileOutput_Stream(output_name_beans);
        my_buffer_byte= final_str.getBytes();
        bean_out.write(my_buffer_byte);
        bean_out.close();
```

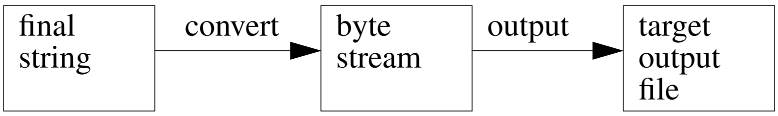


Fig. 5.7. The final string is converted into byte stream and passed to the target file.

5.5 Putting It All Together

In 5.4.1-5.4.4 we have described the structures and functions of the important components of our code generator. What lessons have we learned? On one hand, it should be flexible, with the ability to handle different types of input files, but on the other side, it also should be, to use Einstein's phrase, "as simple as possible -- but no simpler". More information about how to use the code generator can be found in Appendix B: a complete user manual for JavaBeans code Generator.

6 Conclusion And Future Works

The age of standing-alone program is over, today networking is involved in most of the softwares development one way or the other.

Recently Sun launched its latest Java Development Kit version, also known as JDK 1.1.4. Most of what is new and exciting about this new version are the potential for new kinds of dynamic, networked applications. Some of the newly approved classes are sockets, URLs, InetAddress, UDP, TCP/IP, multicasting, Applets, Servlets, Remote Method Invocation, RMI, and more.

Sun Microsystems has produced Java IDL which can create distributed Web-enabled Java applications that can transparently invoke operations on remote network services using the industry standard OMG IDL (Object Management Group Interface Definition Language) and IIOP (Internet Inter-ORB Protocol) defined by the Object Management Group.

In our thesis project, we have presented a JavaBeans code generator which is used to generate JavaBeans compartible source code from a IDL definition of graphical user interface. Thanks to the unique syntax structure of IDL, which is very similar to Java, we successfully accomplished the mapping from IDL definition file to GUI JavaBeans file.

The latest version of code generator is mostly developed for the local intranet users, and the next version is very likely to be modified in the following ways:

- provide interactive functions, possibly a applet;
- provide more standard functions, like help and etc;
- provide the possibility to handle input files other than IDL;
- provide a more friendly user interface;

We believe that the future is networking.

7 References

[1] *JavaBeans,* http://splash.javasoft.com/beans/

[2] *Java,* http://java.sun.com/

[3] Greg Voss, edited by Andy Quinn, *JavaBeans Tutorial, JavaSoft,* available at http://java.sun.com/docs/books/tutorial/beans/

[4] Peter Coffee, Michael Morrison, Randy Weems, and Jack Leong, *How to Program JavaBeans,* Ziff-Davis Press

[5] Alden DeSoto, *Using the Beans Development Kit 1.0, Feb 1997, a tutorial,* JavaSoft, download available at http://splash.javasoft.com/beans/bdk_download.html

[6] Sun Microsystems, *The Java Tutorial Object-Oriented Programming for the Internet,* available at http://java.sun.com/docs/books/tutorial/index.html

[7] Tim Ritchey, *Programming with Java, Beta 2.0*

[8] Patrick Naughton, *The Java Handbook,* Osborne McGraw-Hill

[9] Dan Brookshier, *JavaBeans Developer's reference,* New Riders Publishing

[10] Sun Microsystems, *Java Platform 1.1.3 core API,* available at file:/home/qtxyin/jdk1.1.3/docs/api/packages.html

[11] Sun Microsystems, *Java Platform 1.1.4 core API,* available at http://java.sun.com/products/jdk/1.1/docs/api/packages.html

[12] Sun Microsystems, *WorkShop Developer Products,* http://www.sun.com/workshop/java/

[13] Borland, *JBuiler,* http://www.borland.com/jbuilder

[14] Symantec, *Visual Cafe for Windows 95/NT,* http://www.symantec.com/vcafeprowin/

[15] From IDL to Java, http://www-koba.ces.kyutech.ac.jp/java/

[16] OrbixWeb for Java, http://www-usa.iona.com/Products/Orbix/OrbixWeb/index.html

[17] Sun Microsystems Inc, *Mapping IDL to Java,*

[18] Andrew W. Appel, *Modern Compiler Implementation in Java: Basic Techniques.*

[19] Elliot Berk, *JLex: A Lexical Analyzer Generator for Java(TM),* available at http://www.cs.princeton.edu/~appel/modern/java/JLex/

[20] Java(tm) Based Constructor of Useful Parsers (CUP) Home Page, http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/home.html

[21] IDL/Java integration, http://www-usa.iona.com/ Products/ Orbix/OrbixWeb/ doc/pguide/pgind.html

[22] A Sun product, http://java.sun.com/products/jdk/ preview/docs/ext/idl/docs/jidlCompiler.html

[23] IDL/Java code mapping http://splash.javasoft.com/JavaIDL-alpha2.0/pages/release-notes-win32-x86.html

[24] Code generator, http://www.accord.com/products/doc/stargen.html

8 Appendix

Appendix A: Abbreviations And Acronyms

JDK Version 1.1.3

We use the Java Development Kit 1.1.3, or JDK1.1.3, which is the standard suite of development tools for Java. It includes improvements in functionality, performance, and quality over JDK 1.0.2, but still there are some bugs in it, which now has been fixed in JDK 1.1.4 [11]. JDK 1.1.3 offers new capabilities: Internationalization, signed Applets, Java ARchive files(JAR) file format, AWT (window toolkit) enhancements, JavaBeans(tm) component model, networking enhancements, Math package for large numbers, Remote Method Invocation, Reflection, Java DataBase Connectivity, JDBC, database connectivity, new Java Native Interface, Object Serialization, Inner Classes, and performance enhancements [10].

Beans Development Kit, BDK, Version 1.0

The JavaBeans Development Kit, or BDK, is designed to be used in conjunction with the JDK [5]. It provides tools necessary to create and test JavaBeans components. The most important tool in BDK is the BeanBox, which we used to test our sample JavaBeans in a visual environment and connect sample JavaBeans with other beans.

Abbreviations and Acronyms

API	Application Program Interface
AWT	Abstract Windowing Toolkit
BDK	JavaBeans Development Kit
CORBA	Common Object Request Broker Architecture
GUI	Graphical User Interface
IDL	Interface Definition Language
JAR	Java ARchive files

JDBC	Java DataBase Connectivity
JDK	the Java Development Kit
ISV	Independent Software Vender
RMI	Remote Method Invocation

Appendix B: User Manual For The Code Generator

A complete user manual for JavaBeans code Generator, Version 1.0.1

Written By Hao Yin and Yu Zhuang
Last updated: 18-Oct-97 10:12:35

This instruction page contains the information you need to use our JavaBeans code generator. There are a lot of reasons for why we wrote this generator, and the most important one is saving time. Even though most of the Java graphic pieces appear to be totally different from each other, still, they have so much in common, and it's not hard to find them if we take a close look at them. And this is where our generator comes into the picture. It simply takes one or several IDL files as input and produces the JavaBeans file for you.

CONTENTS

BEFORE YOU GET STARTED

The Java Development Kit (JDK) contains all the basic things you need to start programming in Java, and therefore has to be installed properly. You can find out more information about download and on-line help on this website:

http://java.sun.com/products/JDK/

and the latest version is JDK 1.1.4, which includes the following:

- Frequently asked questions
- Applet examples (some of which are in this release, as well)
- API documentation
- Documentation for the Java programming tools (including Java, javac, and jdb)
- The latest Java Language Specification

- Release specific information

SETTING UP THE ENVIRONMENT

- Pick your working directory.
- Download the cool BeansGenerator here:
  http://www.kk.etx.ericsson.se/~qtxyin/user.html
- You can get on-line help here:
  http://www.kk.etx.ericsson.se/~qtxyin/Readme.html
- Here you can find the source code:
  http://www.kk.etx.ericsson.se/~qtxyin/beansgenerator.java

CONSTRUCTING THE IDL FILES

```
//a sample IDL file
struct Person {
     //readonly attribute
    string name;
    //structure
    Address address;
};
```

Comments are used here to define the character of a certain type, it might be attribute, structure and so on, for instance, name is a read-only attribute in this case, and address is a structure called Address.

An instance of a certain structure is defined by using the corresponding IDL file, in this case, address is defined as a Address structure.

```
//a IDL file called Address
struct Address {
      //readonly attribute
    string Street, Postnr;
};
```

UNPACKING THE DOCUMENTATION

After downloading BeansGenerator.tar, you can unpack the documentation anywhere you like with the appropriate command below,

machine> tar xvf BeansGenerator.tar

After that, you will get two new directories, called "**prog/**" and "**jb/**". In "**jb/**" there are all pattern files which our beansgenerator will use, and in "**prog/**" there are three example "**.idl**" files, the beansgenerator source code and executing code.

USING THE GENERATOR

Make sure all the IDL files are located under the same directory as beansgenerator.class.

It's very easy to use this generator, for instance, if we want to **Person.idl** as the input to produce the Java code, simply type this command:

machine > java beansgenerator Person.idl

The new file will have the name **Person.java,** and is located under the current directory. **Person.java** can be compiled just as a usual Java file.

FEEDBACK

To submit comments or suggestions about the generator, please send mail to:
qtxyin@kk.ericsson.se, qtxyzh@kk.ericsson.se