Using Plans for Specifying Preconfigured Bean Sets

Dietrich Birngruber, Markus Hof
Johannes Kepler University, Linz
Institute for Practical Computer Science
A-4040, Linz, Austria
(birngruber | hof)@ssw.uni-linz.ac.at

Abstract

Component composition is either done manually in a component assembly tool or with specific applications (wizards). However, from the view of the application programmer this situation is not satisfactory. Wizards are not flexible enough and their creation requires substantial efforts. Doing everything manually is annoying, requires quite a good knowledge of the used components and increases the training period of the application programmer considerably.

This paper describes a new approach for composing a set of JavaBeans using so-called bean plans. *A bean plan is a partial description of a set of beans and their wiring. It describes a semi-finished bean suite with some JavaBeans being already preconfigured and preconnected and others being still isolated. The application programmer can later complete the ensemble with any component assembly tool by defining missing connections and overriding preconfigured properties. The plan is shipped together with the bean suite and it relieves the application programmer from routine configuration tasks.* BeanPlan *is the scripting language for writing bean plans and will be described in this paper.*

Keywords*: Components, Compositional Patterns, Composition Automation, JavaBeans*

1. Introduction

Software components offer solutions to various software engineering problems, e.g. how to build and maintain complex software systems in a changing environment. The idea is to acquire prefabricated and well-tested software components on the market and to compose them to new applications. There are already markets [2] for components as well as some common understanding about the term *software component* [9].

Software composition divides the development process into two parts. First, the component developer that writes new component libraries and second, the application programmer uses them to compose his applications. Different individuals often take these parts. This leads to a knowledge gap, as the application programmer has to determine *how* and in *which* context he can apply the different components.

This paper introduces the idea of bean plans that tries to bridge this gap. A bean plan describes how an application programmer typically glues components of a given library together, i.e. it captures domain knowledge. The application programmer processes these plans with a plan builder. The builder instantiates the plan and consults the programmer if necessary. Our current implementation is tuned toward JavaBeans [8], however one may apply the idea also to other component technologies, e.g. Microsoft's COM [4].

Normally, the application programmer can choose from various ways to glue JavaBeans together: Visual composition (e.g. with VisualAge), scripting languages (e.g. JavaScript [3]) or Java [5] itself. However, before he is able to combine the components, he has to know *which* kinds of components exist and *how* he can combine them. Component libraries (bean suites) often contain some kind of documentation that describes the look of typical composition patterns. However, each time the application programmer wants to use the beans he has to *remember* one of the typical composition patterns, to *hope* that the documentation is accurate and to *instantiate* the chosen composition pattern by hand (component by component). Considering a suite with many components, it is a tedious task to find the right components and to instantiate and glue them together according to the desired composition pattern. Our bean builder and bean plans automate this process by supplying the programmer with insider knowledge of the bean developer.

A bean plan is written in a scripting language (*BeanPlans*) that is instantiated by the plan builder tool. The plan guides application programmers semi-automatically (similarly to a wizard) through the assembly process. In contrast to a wizard, a plan is not fixed but can be modified. It is like a composition template with optionally some degrees of freedom. A plan may contain *interactive decision spots* that offer choices to the application programmer.

The idea is similar to a prefabricated house, where the customer (application programmer) can influence some details of his new house. When a customer wants to have a house built, he asks the architect (the plan builder tool) for the available types of houses (the available plans). After having selected the appropriate type of house, the architect walks through the plan together with the customer and they settle all the design decisions that have not already been fixed by the plan, e.g. the kind of heating, the material for the floor and the windows, etc. Then the architect builds the house (the plan builder creates the bean ensemble and wires them). Finally, the house (the bean ensemble) can be used (by the end user), but this is completely separated from the building process.

An application programmer uses bean plans at design-time, i.e. when he wires the components to a new application. Plans are processed at a different point in time as other scripting languages for component composition as e.g. Piccola [1] or JavaScript. These languages are interpreted at run-time, i.e. at the computer of the end user while the actual application runs.

Currently, our plan builder produces an XML [10] description of the composed beans. Sun Microsystems tries to establish this as a standard just now [6]. It submitted its "Long-Term Persistence for JavaBeans" to the Java Community Process [7]. We hope that - by the time this paper is published - commercial bean assembly tools support this upcoming standard and are able to read design descriptions in XML.

The paper is structured as follows. Chapter 2 describes the context and roles that can work with bean plans and gives an extended example. A detailed description of the BeanPlan scripting language can be found in Chapter 3. Finally, in Chapter 4, we draw some conclusions and outline our plans for the future.

2. Working with Bean Plans

Section 2.1 of this paper describes the context and roles that can work with bean plans. Section 2.2 presents an extended example that explains some key concepts of bean plans.

2.1. Overview

Chapter 1 introduced the ideas underlying the concept of a bean plan. Such a plan serves as the means with which an expert of a given bean suite can pass on his expert knowledge to other persons.

Let us assume we have a bean suite that offers statistical functionality. A typical set-up consists of three beans: A data collector that supplies the input data, an evaluator that performs some statistical operation on the data, and a visualizer that presents the result in a diagram. One or several events interconnect these beans with each other. For example, the data collector fires an event when it can deliver a new data element, the evaluator reacts on it and fires an event which notifies the visualizer that the results have changed. In order to get this set-up, our mythical application programmer has to create all beans manually within his component assembly environment. Afterwards, he has to interconnect them to enable the desired information flow. To do this, he has to know not only the information about where he has to feed the statistical data and which bean displays the statistic output, but also the way in which the three statistic beans interact with each other.

This is the point where bean plans come into the picture. They offer a way, in which the creator of the bean suite can supply the application programmer with enough information to avoid the need for this knowledge. The plan guides the application programmer semi automatically (similar to a wizard) through the assembly process. In contrast to a wizard, a plan is not a fixed application but a script that the plan builder processes (see Figure 1).

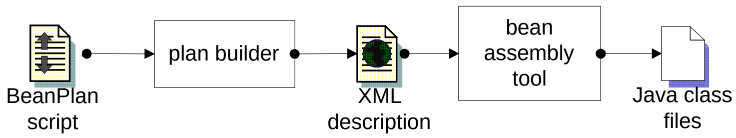

Figure 1: Deployment sequence of beans assembly using bean plans

The application programmer starts by choosing a suitable bean plan. The plan builder processes the chosen plan and instantiates the declared beans, sets their properties and registers them for events of other beans. The plan builder does this semi-automatically, because a bean plan can contain spots where the application programmer can select among a variety of alternatives for beans to be plugged in, i.e. similar to multiple-choice questions. We call them *interactive decision spots*. The writer of the plan decides which and how many of these spots exist, i.e. how many decisions an application programmer has to supply whenever this plan is processed. Using more of them increases the plan’s degrees of freedom but reduces the degrees of automation. Examples for such decisions may be the actual value of a property (e.g. the foreground colour used in one or several buttons), whether or not there shall be a visualisation bean or about the actual type of the bean (bar chart or pie chart).

The XML output of the plan builder is a concrete instantiation of the general concept of the plan (similar to an object that is a concrete instantiation of a class). However, currently there are discussions about different models for long-term persistence of JavaBeans, e.g. BeanScript [6]. We hope, that by the time this paper is published, the producers of bean assembly tools agreed on a common persistence model. This would allow application programmers to exchange their composed set-ups between assembly environments as well as to import the instantiated plans into their own assembly environments. This would greatly ease the use of a new bean suite, as the plan considerably reduces the initial effort necessary to understand the suite's concept. Finally, the output of bean assembly tools is a collection of Java class files that make up the assembled application.

2.2. An Extended Example

This section illustrates some of the key concepts of bean plans using an extended example. We use a bean plan for the composition of a GUI from JavaBeans. The plan *JTableView* describes a typical wiring of *JTable*, *TableModel*, *JButton* and visual containers. Figure 2 shows the containment hierarchy of the used JavaBeans.

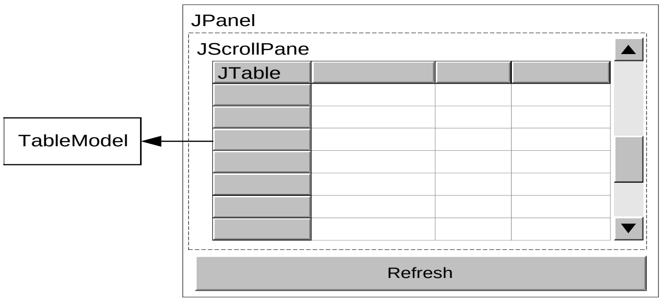

Figure 2: Containment Hierarchy

We divide the example code into several code snippets for presentation reasons. This code shows how several Beans are customized by setting fixed properties, by asking the application programmer to fill in missing values and by gluing them together with event listeners and hierarchical dependencies.

```
import javax.swing.*;
import java.awt.event.* ;
import java.awt.Color;
import java.awt.BorderLayout;
import javax.swing.table.TableModel;

plan JTableView {
   Bean refreshButton = new JButton {
      text = “Refresh”; // set property text
   }
```

This code defines the plan *JTableView* that contains (up to now) one bean declaration. The declared bean *refreshButton* is of class *javax.swing.JButton*. Within its initialiser (the curly brackets), we assign the value “Refresh” to the property *text*.

```
   // ask the programmer to pick a TableModel and show
   // the appropriate customizer or property sheet of the model
   Bean tableModel = new <TableModel> {  }
```

```
Bean tableView = new JTable {
    autoCreateColumnsFromModel = true;
    model = tableModel;
}
```

This snippet declares two new beans: *tableModel* and *tableView*. The visual bean *tableView* of class *JTable* displays the data provided by the non-visual bean *tableModel* of class *TableModel*. Typically, the component palette of the component assembly environment contains several classes of table models, e.g. a model that queries a database, or a model that fetches its data from a regular file. Some of these models may even be unknown to the writer of the plan. This variety is expressed in the plan using interactive decision spots. Angle brackets surround decision spots, which enable user interaction during the execution of the plan. In our example, the decision spot allows the application programmer to exchange the class of the *TableModel* with one of its subclasses. The builder checks for all known types that are compatible with *TableModel* and it creates a user dialog with the possible choices. The result of this dialog is then used for the actual instantiation of the bean *tableModel*.

```
Bean jScrollPane1 = new JScrollPane {
    viewPortView = tableView;
}
Bean container = new JPanel {
    layout = new BorderLayout;
    // ask the programmer for a background color
    // using the appropriate property editor
    background = <>;
    add (jScrollPane1, BorderLayout.CENTER);
    add (refreshButton, BorderLayout.SOUTH);
}
```

In this snippet, we first create the *JScrollPane jScrollPane1* that contains our *tableView*, i.e. we set the property *viewPortView* accordingly. Finally we create the instance *container* of type *JPanel*, set its *layout* and *background* property and invoke its method *add* twice. The plan sets the property *background* to “<>”. Using the bare angle brackets tells the builder to invoke the appropriate property editor, in this case the colour picker. Finally, the initialiser invokes the method *add* of *container* to add *jScrollPane1* and *refreshButton* to the top-level visual container.

```
    // interconnect beans using events
    on refreshButton.Action.actionPerformed call tableView.repaint();
}
```

Plans define the event bindings at the end of their definitions after all beans have been declared. In this example, we want that the pressing of *refreshButton* causes *tableView* to be repainted. *tableView* is not an *ActionListener* and therefore the event *ActionListener.actionPerformed* of *refreshButton* invokes the method *tableView.repaint()*.

According to the JavaBeans specification [8], the names of all event listener interfaces end with “Listener”. Therefore, we omit it in order to increase the readability.

3. BeanPlan Details

In the following sections, we describe the different BeanPlan language features in depth. Section 3.1 describes the layout of a plan, Section 3.2 presents the functionality of interactive decision spots and Sections 3.3 to 3.4 show the usage of the different kinds of declarations. Finally, Section 3.5 explains the two different kinds of event bindings.

3.1. Layout

The layout and the general language syntax of BeanPlan are quite similar to the syntax of a Java class. A plan script starts with arbitrary many Java import statements. However, there is no package mechanism for the plans themselves. The actual plan consists of the keyword *plan*, its name and its definition between curly brackets.

```
import javax.swing.*;
import java.awt.event.*;

plan MyUI {
   // declarations
   // event bindings
}
```

The plan definition consists of arbitrarily many declarations (see Sections 3.3, 3.4 and 3.5) followed by event bindings (see Section 3.6) that glue the declared beans together. The definitions and event bindings are sequentially computed in the order of their appearance. While the plan builder processes the script, it may encounter language features that demand user interaction. There are three such features:

- Decision spots contain a set of possible choices enclosed within angle brackets. They are described in detail in Section 3.2.
- “<>” defines a property value that has to be supplied by the application programmer. The plan builder automatically opens the appropriate property editor.
- The keyword *customized* opens a bean customizer or property sheet (see also Section 3.4).

3.2. Interactive Decision Spots

The creator of a plan uses interactive decision spots whenever he does not know the exact type of a bean and therefore wants to delegate the final decision to the application programmer that runs the plan builder. A plan builder decides at run-time whether to ask the application programmer or not if it is not ambiguous. The result of a decision spot is either the name of a Java type or *null*. If the result of the spot is used to create a new bean instance and the spot’s value is *null*, the bean is not instantiated and the corresponding bean variable gets the *null* value.

Decision spots consist of one or several choices separated by “|” and are enclosed in angle brackets. The simplest form of an interactive decision spot contains a set of choices:

```
<FlowLayout | BorderLayout | CardLayout>
```

The above decision spot gives the application programmer the choice between three possible layout managers given by the classes *FlowLayout*, *BorderLayout*, and *CardLayout*. When the plan builder encounters this spot, it opens a dialog containing the three layout classes and all their subclasses. The application programmer can select on of the classes. If the subclasses of a class should be omitted, one has to insert the keyword *fixed*, which denotes that the following class name should not include its subclasses. E.g., fixed *CardLayout* offers only a *CardLayout* without subclasses.

This simple form of decision spots allows the dynamic extensibility of plans. Subclasses, like e.g. *MyCardLayout*, known only to the application programmer are displayed in the user dialog, even though the creator of the plan does not know them.

A choice within a decision spot may either be a class name as in the above example, the constant *null* or a *Spot* variable. A variable is supposed to contain either a class name or *null*, denoting that the application programmer specified no type name.

The three choices of the above decision spot are actually three rules using the simple rule syntax that consists only of one identifier. There is a second kind of rule syntax:

```
<  (container is JPanel) ? JButton : Button >
```

This decision spot contains only one rule. This rule uses the ?-operator to distinguish two cases. The result of this spot depends on the value of the variable *container*. If it is of type *Bean* and the bean referenced by *container* is of type *JPanel*, the condition returns *true*. If *container* is of type *Spot,* the condition returns *true* only if is value is equal to “JPanel”. All other cases evaluate to false. If the condition is *true* then the result is *JButton*, otherwise it is *Button*. In this example we have only one choice, i.e. if the chosen class (*JButton* or *Button*) has no subclasses the plan builder does not need to consult the application programmer.

The above form of a rule is the most complex form. One can omit the else clause. If a condition of such a rule evaluates to *false*, the rule evaluates to nothing, i.e. the choice is discarded.

```
<
   (p is ThisName) ? JButton |
   (p is OtherName) ? Button |
   (default) ? JPanel
>
```

Let us assume that *p* is of type *Spot* and contains the value “OtherName”. In this case, the first condition evaluates to *false*, which means that the first choice is discarded. The second choice is *true*, i.e. *Button* is a valid choice. The condition of the third rule is the keyword *default* that has similar semantics as the corresponding Java keyword. It has the value *true* only if all other conditions are *false*. In our example, it is *false* and the third rule is omitted. Finally, the reduced decision spot is:

```
<Button>
```

This interactive decision spot produces a dialog with the class *Button* and all its subclasses. If there are no subclasses the plan builder does not show the dialog, as the decision is already determined.

3.3. The Spot Declaration

Spots serve as temporary storage for results of decision spots that are either used further down in the script or are used several times.

```
Spot button1 = < JButton | Button >
```

The value of a Spot variable is always determined by evaluating an interactive decision spot. The above example leads to a user dialog, where the application programmer can influence whether the plan should use Swing buttons or ordinary AWT buttons, i.e. *button1* will contain either “JButton” or “Button”. This value can then later be used for, e.g. bean declarations (see Section 3.4).

3.4. The Bean Declaration

There are two distinct versions for the creation of a new bean: creating a bean from scratch or cloning an already existing bean.

```
Bean button2 = clone button1 {}
Bean button3 = new JButton {}
```

button2 is a clone of *button1*, i.e. its properties are set accordingly and can later be modified in the curly brackets. *button3* is created using the no-arg constructor of the class *JButton*. The two examples above show the simplest forms of bean declarations. We introduce several possible extensions in the following subsections.

Different Ways to Define the Class of the New Instance

In order to create a new bean with the *new* operator one has to specify the class of the bean. This can be specified in several ways:

```
Bean b = new JButton {}
```

This possibility explicitly states the Java class that should be taken to create the new instance. No user interaction is necessary for this declaration to execute.

```
Spot buttonClass = .....
.....
Bean b = new buttonClass {}
```

This version takes the run-time value of the variable *buttonClass* as the desired class name. This technique enables the plan to ask the application programmer once for the desired button class and to reuse this knowledge for all created buttons later.

```
Bean b = new < JButton | Button > {}
```

This declaration uses a decision spot to specify the desired class. This leads to a user dialog to determine the actually used type.

Initialisation of the New Bean

Often, after the bean has been instantiated, some properties have to be set or the bean containment hierarchy has to be specified. The plan specifies such actions between the curly brackets at the end of the bean declaration. Our plan language currently supports two kinds of initialisation statements: setting of properties and invoking of methods of the new bean.

```
backgroundColor = Color.red;
```

The above statement sets the property *backgroundColor* of the enclosing bean to the value *Color.red*. There are several different types of such property assignments:

```
bgColor = otherBean.color; // property value of another bean
bgColor = Color.red; // value of public static final Java variable
layout = customized new <BorderLayout | CardLayout>; // new object
initialSize = 12; // value is a Java constant
bgColor = <>; // ask application programmer for value
```

The most interesting type of property assignment is the last one. As the angle brackets already indicate, the plan builder sets this property with the help of the application programmer by opening the appropriate property editor.

The second kind of initialisation statements is the method invocation. The language allows only invocations of methods without return value. Invoking methods on newly allocated beans is mostly important in order to arrange the beans of the plan in some kind of hierarchy.

```
Bean b1 = new JButton { text = "OK"; }
Bean b2 = new JButton { text = "Cancel"; }
Bean p = new JPanel {
   add(b1);
   add(b2, BorderLayout.NORTH);
}
```

The above declarations create three bean instances and initialise them. The two buttons set their property *text*. The panel sets no properties, but invokes its method *add* twice in order to include the two buttons into its set of contained components. The arguments of such invocations may either be previously defined designators, public static final variables (e.g. *BorderLayout.NORTH*) or Java constant values (e.g. 17).

Using Bean Customizers

Regardless of whether a bean is created or cloned, it may be desirable that the customizer of its class is activated after it has been instantiated. Adding the keyword *customized* to the bean declaration can specify this.

```
Bean b1 = customized new JButton {}
Bean b2 = customized clone b1 {}
```

The above declarations create two new beans. This process executes the following three steps:

1. The new bean is instantiated either from scratch or by cloning.
2. The corresponding initialisation (curly brackets) is executed.
3. The bean customizer for the classes of *b1* and *b2* are executed. This allows the application programmer to set the properties of these beans that are either not known to the plan or that are of no interest for it.

Coping with *null* values

Depending on the definition of the decision spot, its result may be the *null* value.

```
Bean b = new < JButton | Button | null > {}
```

If the application programmer chooses the alternative *null*, it is not possible to create a bean instance for the variable *b*. In this case, no bean is instantiated and the variable *b* denotes *null*. Initialisation and customisation for this bean is omitted.

3.5. The Event Bindings

Event bindings interconnect, i.e. glue, previously declared components together. There are two kinds of event bindings. Once can use the first form if the two components are compatible, i.e. the receiver understands the event sent by the event source.

```
on button.Action notify myBean;
```

This statement registers the bean *myBean* for Action events generated by *button*. Of course, *myBean* has to implement the *ActionListener* interface. If this is not the case, i.e. the two beans are not directly compatible, one has to use the second form of the event binding. This form connects a specified method of an event listener with an arbitrary method of the receiver.

```
on button.Action.actionPerformed call myBean.startJob();
```

This statement connects the event method *actionPerformed* of the bean *button* with the method *startJob* of *myBean*. This allows gluing incompatible beans together. The return value of the receiving method is ignored. However, the receiving method may have arguments. They may either be previously defined designators, public static final variables (e.g. *BorderLayout.NORTH*), Java constant values (e.g. 17), the handled event or attributes of the handled event. The last two kinds of arguments start with the predefined name *event*.

```
on button.Action.actionPerformed call myBean.start1(event, false);
on button.Action.actionPerformed call myBean.start2(event.source, 17);
```

Using interactive decision spots within a plan may result in bean variables with a *null* value, i.e. the sender or the receiver of an event binding was not instantiated by the plan builder. In this case, it is not possible to connect the two beans of the statement. The builder ignores such event bindings.

4. Conclusions

In this paper, we have argued that there is a need for managing large component pools and for improving the deployment of Java software components in order to reduce software development time. We think that this area has been largely neglected so far. Our bean plans fill this gap. At design time the application programmer executes a bean plan and determines interactively some features of this plan instantiation. Similar to a component composition language, a plan script describes common composition patterns. However, the plan script runs at design time and not at run time of the assembled program.

Bean plans introduce the concept of decision spots for user interactions. Similar to a wizard, a plan guides an application programmer semi automatically through the assembly process. However, a plan script is not fixed and can easily be adopted.

Currently we are developing a prototype of the plan builder and the grammar of the scripting language is still not fixed. We plan to integrate text comments that the plan builder displays as additional information in its dialogs. Besides transforming a plan script into a concrete bean design we plan to offer the inverse functionality. This allows application programmers to store their typical compositions as bean plans by translating the XML design (extracted from their bean assembly tools) into a plan for later reuse.

Other topics for future research include to deal with indexed properties and to allow more flexible event binding mechanisms.

5. References

[1] Achermann Franz, Nierstrasz Oscar: *Applications = Components + Scripts -- A tour of Piccola*. Software Architectures and Component Technology. Mehmet Aksit (Ed.). Kluwer. 2000.

[2] http://www.developer.com/directories/

[3] ECMA 262: *ECMAScript Language Specification*. 3rd Edition. Dezember 1999. http://www.ecma.ch

[4] Eddon Guy, Eddon Henry: Inside Distributed COM. Microsoft Press. 1998.

[5] Gosling James, McGilton Henry: *The Java Programming Language*. OOPSLA '95 Tutorial Notes. Austin, TX. 1995.

[6] Milne Philip, Walrath Kathy*: Long-Term Persistence for JavaBeans. Sun Microsystems*. 24. Nov. 1999. http://java.sun.com/products/jfc/tsc/articles/persistence

[7] JSR 000057 Long-Term Persistence for JavaBeans™ Specification. Sun Microsystems. 14. Feb. 2000. http://java.sun.com/aboutJava/communityprocess/jsr/jsr_057_jbprs.html

[8] Sun Microsystems: *JavaBeans API Specification*. 1997. http://java.sun.com/beans/spec.html

[9] Clemens Szyperski: *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley. 1997.

[10] W3C: *Extensible Markup Language (XML) 1.0*. W3C Recommendation. 10. Feb. 1998. http://www.w3.org/TR/REC-xml