An Integrated Unit Test Tool for Software Components

Perla Inés Velasco Elizondo
Universidad Autónoma de Tlaxcala
Laboratorio Nacional de Informática Avanzada, A.C.
pvelasco@lania.mx

Juan Manuel Fernández Peña
Facultad de Estadística e Informática
Universidad Veracruzana
jfernandez@uv.mx

Abstract. Although component-based software development has become a relatively accepted approach, one of its principal limitations is the lack of formal testing methods. JavaBeans is one alternative for constructing component-based software that has gained widespread acceptance. This article introduces the development of a beans testing tool The purpose of the tool is to provide the user with guidelines that permit the performance of component selection and evaluation tasks through the automatic generation and execution of test cases.

1. Introduction

At present, reuse is a common practice during software development processes. Situations such as market competition to generate new products and update versions have motivated developers to seek alternatives to generate software rapidly through the use of methodologies that contemplate reuse as a major activity; this has led to improvements in the speed with which programs are built. Developmental approaches such as object-oriented and component-based design are significant examples of this.

Although tools to test object-oriented software exist and, under some circumstances, can be applied to components, many of them rest on the source code and, because components are usually offered as finished products not offered with the source code, they are difficult to use.

In the case of component-based software, which has become a relatively widespread and attractive approach to quick software construction, there are important limitations regarding the formality of the testing techniques used. The lack of guarantees that the software elements being reused will work correctly under all circumstances may compromise the quality of the products developed considerably.

Added to this, many final component users lack the formal knowledge necessary to carry out tests. Many times the evaluation that they conduct consists of a manual test that is neither systematized nor well-founded. Should a more exhaustive test be necessary, it would be broader and more formal but would be hard to automate; furthermore, to carry it out a considerable amount of time would be needed.

The purpose of this article, which considers component-based development and software development central themes, is to present the development of a component testing tool that has been coined "PACJavaBeans" (JavaBeans Automated Component Test). JavaBeans [SUN02] is one of the most popular models for the construction of this type of software, and because is an innovation in component testing material, we decided to experiment with the component model proposed by Sun Microsystems, JavaBeans. The assumption was made that previous experiences with the construction of objected/oriented software with Java would give the research stronger direction.

The tool is intended to permit the automatic generation and execution of test cases, so that with the results, the component user can obtain guidelines that facilitate their selection and evaluation. The type of test that the tool performs takes place at unit level. With functional test techniques being used.

2. The Beans test: present status

JavaBeans is an API implemented for the construction and use of components written in Java, components that are known as "beans." This API, formally implemented in the Bean Development Kit (BDK), permits components to be loaded, used, modified, and connected together, so that new beans, applets, or complete applications can be built. The BDK has three main elements: the BeanBox (developer environment), ToolBox (bean repository), and the PropertySheet (bean properties).

The BeanBox offers certain advantages in bean testing. The user can charge, execute, and interconnect components within the BeanBox. The PropertySheet, in turn, can directly modify the values of certain properties or visualize the effects on them as a result of the execution of certain methods. These activities are repeated as many times as necessary; by monitoring the results obtained, the user can determine if the component is functioning as it should. The features offer are, however, quite different from those of a tool created specifically for software testing, as the test is purely visual.

If BeanBox reveals that a bean is faulty, it is not always possible to determine which is the defective part. With a little luck, the user may be able to identify under what circumstances the fault was generated, but the test remains murky. Exist beans methods cannot be directly touched by the user; using BDK, it is difficult to test them.

Commercial tools such as JUnit [MCW02] and Jtest [PAR02] were not conceived for components; nevertheless, they permit testing Java classes. Thus, they can be used with JavaBeans. It is important to emphasize that these tools depend on the source code and do not work directly with JAR files, the bean presentation format.

3. Testing tool

When dealing with concepts such as testing and test case [IEEE90] and readdressing the ideas presented in [WOH98] and [FER99] for tool design, four stages of the testing process were defined:

a) Selection of units to be tested,
b) Generation of test cases,
c) Execution of testing plan, and
d) Presentation of results

The tool carries out these steps with eight modules that, with the exception of the first, all use and generate a set of interconnected outputs. The eight modules are reached sequentially from the dispatcher module; see Figure 1.

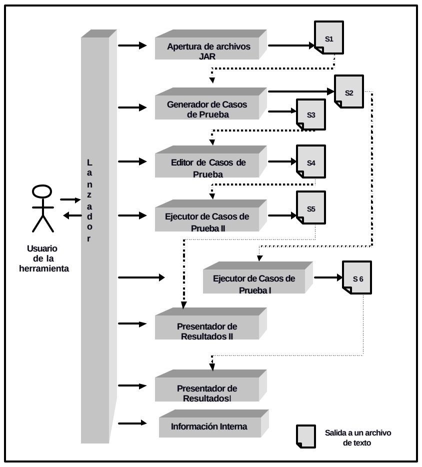

Usuario de la herramienta
Lanzador
Apertura de archivos JAR
S1
Generador de Casos de Prueba
S2
S3
Editor de Casos de Prueba
S4
Ejecutor de Casos de Prueba II
S5
Ejecutor de Casos de Prueba I
S 6
Presentador de Resultados II
Presentador de ResultadosI
Información Interna
Salida a un archivo de texto

Figure 1. Architecture of testing tool

The following sections will describe with more detail the tool's modules.

3.1 Module for opening JAR files

Object-oriented software is generally built on a foundation of class sets which offer and use services between each other in order to conduct certain operations. Apart from classes, there are other elements necessary for application functioning, such as image, configuration, and help files. This situation also arises for beans

Most of the components constructed under the JavaBeans model contain the following elements:

a) One or more classes defined as beans
b) A file represented by an icon, and
c) One or more classes that are not defined as beans but that are necessary in order for the component to function.

To lessen the difficulty of managing these elements individually, beans appears as a JAR file that packages all its elements in one unit. When this file is loaded in the BeanBox, its content is extracted so that it can be used in a work setting.

Similar to BeanBox, the constructed tool's opening module has the capacity to read JAR files under the following conditions:

- *Regarding content:*
  - That components be implemented according to JavaBeans model considerations.
  - That the component be valid—that is, that can be correctly loaded and used in the BeanBox.

- *Regarding function:*
  - That the user understand, at least in general terms, how the component works.

The generated output on the module's exit consists of a text file within which are stored the names of each bean found in the JAR; the user is presented with this information through the module's interface.

3.2 Test case generator module

In accordance with [BIN95], [PER90], [FER99], and [PRE98], object-oriented software unit test establish the unit as the element of evaluation, specifically an instance of a class, that is, an object; special attention is given to its attributes and operations. Thus, an ideal test could be one in which all methods contained in the class are considered.

In the context of components, and because the user who tries them out does not usually have the source code, a functional testing approach is taken, as this permits the tester to experiment with given inputs and obtained outputs. PACJavaBeans offers two alternatives for the generation of test cases: cases edited by the user and those generated automatically.

For the first alternative, the tool can accept a text file that contains those test cases of interest to the user, who is responsible for designing and editing them.

The second alternative, cases generated automatically, was the fruit of more formal work. In order to automate testing as much as possible, programs are executed with predefined test cases; the following testing techniques are employed:

a) *Limit values test:*
Because there are component methods or functions in which the user expects a certain behavior relating to a given input, automatically generated test cases were conceived under the premise that upon providing border values, it is possible that errors will arise. Nevertheless, quite simple test cases were also considered.

b) *State-based test:*
As there are situations in which a method is executed without problems yet the final component state is not as expected, it was decided to include some considerations for this testing technique. Specifically, the initial and final bean states are inspected in regard to method execution.

In the tool, each test case is generated on the basis of entries that a method can receive and that have been identified through review of entry arguments. These can appear as:

- defined Java types, such as an int, char, Object, Color, etc., or
- a user-defined object

In both cases, the initial goal is to generate and provide test cases for each class method; however, this task may be complicated by the presence of parameters unrecognized by PACJavaBeans or when reference is made to external objects. To tackle this weakness, a mechanism has been implemented that permits the loading of user-defined "Plug-in's": basically, it consists of a file with test values for the new argument and the inclusion of the corresponding object builder
The test cases generated by these two modules are stored in a text file, and the user is given the results obtained after the process is complete.

3.3 Test case executor module

Once the bean's test cases have been generated, the following activity consists of feeding them to the corresponding modules for execution, along with capturing the outputs generated as a product of this series of executions. These are the tasks that the two available executor modules carry out, one executing cases generated automatically by the tool and the other executing user edited test cases.

The results of these executions are a series of outputs identified using the following alternatives:

a) return value method, and
b) exception mechanisms

Due to the nature of object-oriented software, it is convenient to do a more in-depth review and more thorough management of those outputs obtained: sometimes a correct output does not necessarily mean that the component state is also correct. For this reason, a third option is added:

c) object state

Returning to the concepts introduced in [TUR93] and [BAS99], data member representation is inspected along with the way that methods manipulate object representation in order to establish a set of object states. Concretely, these consist of a valid set of states from which an object can accept an input and a valid set of states generated after an output. This makes it possible to execute test cases on the basis of initial component state or when this state has changed as a result of previous operations. The object state consists of values of bean properties at a particular moment.

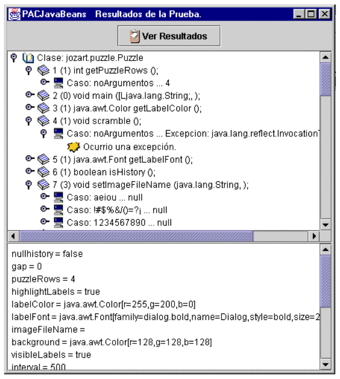
Thus, through user interface, both modules present a summary that informs the user of the results of the process that has been carried out.

3.4 Results presentation modules

The tool contains three modules that permit the user to visualize the results generated after the testing process is complete: two that provide the results generated from the execution of test cases—those generated automatically and edited by the user—and one to show internal class information implemented by the bean.

The first two modules (see Figure 2) provide the following information:

a) Name of class evaluated
b) For each method contained in the class,
   - Description of method arguments
   - Description of return value method.
   - Total number of test cases generated
c) For each test case generated,
   - Detail list of test case values
   - Component state before execution of test case
   - Component state after execution of test case



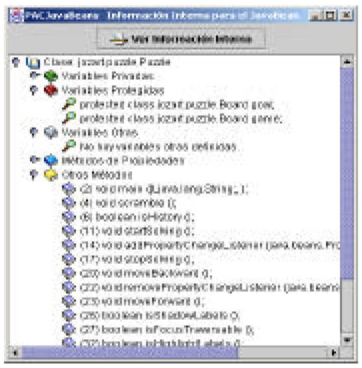
```
Clase: jozart.puzzle.Puzzle
   1 (1) int getPuzzleRows ();
      Caso: noArgumentos ... 4
   2 (0) void main ([Ljava.lang.String;, );
   3 (1) java.awt.Color getLabelColor ();
   4 (1) void scramble ();
      Caso: noArgumentos ... Excepcion: java.lang.reflect.Invocation
         Ocurrio una excepción.
   5 (1) java.awt.Font getLabelFont ();
   6 (1) boolean isHistory ();
   7 (3) void setImageFileName (java.lang.String, );
      Caso: aeiou ... null
      Caso: !#$%&/()=?¡ ... null
      Caso: 1234567890 ... null
```

```
nullhistory = false
gap = 0
puzzleRows = 4
highlightLabels = true
labelColor = java.awt.Color[r=255,g=200,b=0]
labelFont = java.awt.Font[family=dialog.bold,name=Dialog,style=bold,size=2
imageFileName =
background = java.awt.Color[r=128,g=128,b=128]
visibleLabels = true
interval = 500
```

Figure 2. Module that provides test results

The third module (see Figure 3), known as "internal information," provides the following:

a) For each variable:
   - Name
   - Type
b) Number of class methods
c) For each method:
   - Method name
   - Description of input arguments
   - Description of return value



```
Clase: jozart.puzzle.Puzzle
   Variables Privadas
   Variables Protegidas
      protected class jozart.puzzle.Board goal;
      protected class jozart.puzzle.Board game;
   Variables Otras
      No hay variables otras definidas
   Métodos de Propiedades
   Otros Métodos
      (2) void main ([Ljava.lang.String;, );
      (4) void scramble ();
      (6) boolean isHistory ();
      (11) void startBlinking ();
      (14) void addPropertyChangeListener (java.beans.Pro
      (17) void stopBlinking ();
      (20) void moveBackward ();
      (22) void removePropertyChangeListener (java.beans
      (23) void moveForward ();
      (26) boolean isShadowLabels ();
      (27) boolean isFocusTraversable ();
      (32) boolean isHighlightLabels ();
```

Figure 3. Internal information module

The information provided by this group of modules permits the user to obtain some guidelines for bean selection and evaluation.

These guidelines may be generated under the following circumstances:

- changes in bean state
- generated exceptions
- test case analysis, which permits identification of a particular component behavior.
- Analysis of methods and variables, which permits the acquisition of guidelines oriented to software metrics.
- total number of methods tested
- total number of test cases generated
- total number of unexecuted methods

4. Results

In order to verify that PACJavaBeans was working correctly, tests were carried out on several components. What follows are results obtained for the certain beans:

- *The Puzzle component.* Puzzle is, as its name suggests, a bean that represents a numerical puzzle. When it is used in combination with other components, an application can be built that permits the organized manipulation of the pieces that make it up.

- *The TextEditor component.* TextEditor is a component that offers the functions of a simple word processor.

- *The ProgressBar component.* This component implements a thermometer (slide bar) that permits the graphic representation of a particular percentage.

Table 1 shows the errors detected in this group of components.

In the Puzzle component, the tool could detect errors in a set of methods that corresponded to bean properties. For the setPuzzleCols(int) and setPuzzleRows(int) methods, which permit specification of the number of columns and rows of the puzzle, respectively, negative values should not be admitted. This same condition is applicable to methods setGap(int) and void setBelvelHeight(int), which establish the space between and shadowing of pieces. When negative values are administered, the bean does not generate a single exception or corrective action but does generate a change in state that admits this type of values in its properties.

Bean	Method	Input	Expected output	Obtained output

Puzzle	setPuzzleCols(int)
	void setPuzzleRows(int)
	setGap(int)
	void setBelvetHeigth(int)
		-2147483648
		-2147483647
		-2147483646
		-1073741824
		-1
		-2
			Exception or default value of property	Null and property equal to value provided

TextEditor	void save ()	Not applicable	Exception	Null
	void setFontName(java.lang.String)	aeiou !"#$%&/()=?¡ 1234567890	Exception or default value of property	Null and property equal to value provided
	void setFontSize(int) void setFontStyle(int).	-2147483648 -2147483647 -2147483646 -1073741824 -2 -1 0 2147483 645 2147483646 2147483647 1073741824	Exception or default value of property	Null and property equal to value provided
ProgressBar	void setPercent(int).	-1073741824	Exception or default value of property	1
		-2147483646		1
		2147483645		0
		1073741824		0

Table 1. Errors detected by PACJavaBeans

For the bean TextEditor, the void load() method, which permits a file to be loaded, generated an exception upon execution, as was expected. For the void save() method, a similar outcome was expected; this was, however, not the case, and it executed without problems. The void setFontName(java.lang.String) method allows one to set the font type used in a word processor; upon testing, it became clear that no type of validation for this input parameter existed. Something similar occurs in the case of methods that manipulate the size and style of the font, such as void setFontSize(int) and void setFontStyle(int).

An interesting aspect of the ProgressBar component was revealed through tes ting: it relates to the method that modifies the component percentage property, setPercent(int). Analysis of results shows that upon the introduction of values that do not comply with the characteristic of being higher than 0 and less than or equal to 100, the component replaces the value of property percentage with a valid figure. This replacement was not, however, always consistent, being either 0 or 1.

5. Conclusions and future research

The activities carried out for this study have led to the following conclusions:

a) Regarding the testing tool:

- The objective of developing a tool that tests JavaBeans components was realized. The tool is complete and provides an alternative for automatic testing.
- The automatic generation of test cases and their execution are significant advantages for users who lack formal knowledge of the subject.
- For implementation of the tool, relevant aspects of the software testing area are considered.
- Although the use of functional testing techniques could be considered a limitation, given the nature of beans, the testing strategies used correspond to the present context of component-based software.

b) *Regarding future applications and improvements in the tool:*

- Once the market offers more formal component specifications, higher levels of testing could be adopted.
- The potential to test servlets and even Enterprise JavaBeans with similar techniques.
- Improvements could be made in the user's test case editor, so that typing errors are minimized.
- Likewise, a mechanism could be included that provides the initial component state from which the testing process is begun. This would improve the state following bean instanciation.

The information that was gleaned from this study confirms a point that experience has long suggested: that although some might underestimate its importance, testing is a crucial procedure that cannot be overlooked.

References

[BAS99] Bashir, I. y Goel A. "*Testing object-oriented software. Life cycle solutions*". Springer-Verlag New York , Inc.1999.

[BIN95] Binder, R. "*Testing object-oriented systems: A status report*". http:///www.rbsc.com/pages/ootstat.html. Documento en Línea. Abril 1995.

[FER99] Fernández, J. M. "*Test de component-based software. Estado actual*" Informe Técnico N° 28 Serie Verde, CIC, IPN. 1999.

[IEEE90] "*IEEE Standard Glossary of Software Engineering Terminology*". 1990

[PER90] Perry, D. y Kaiser, G. "*Adecuate testing y object-oriented programming*". Testing Object-Oriented Software. IEEE Computer Society, 1998. pp 11-17.

[PRE98] Pressman, R. "*Ingeniería del software. Un enfoque práctico*". Cuarta Edición. McGraw-Hill/Interamaericana de España. 1998.

[SUN02] SUN "*Java Beans*". http:///java.sun.com . Documentación en línea 2002.

[TUR93] Turner, C. y Robson D. "*The sate-based testing of object-oriented programs*".Testing Object-Oriented Software. IEEE Computer Society, 1998. pp 133-142.

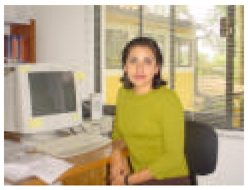
[WOH98] Wohlin C. y Regnell B., "*Reliability certification of software components*".Proceedings Fifth international Conference on Software Reuse, Canada, 1998. pp 56-65.

[MCW02]"*Unit testing Java code with Junit*". http://www.mcwestcorp.com/Junit.html . Documento en línea 2002.

[PAR02]"*Jest*". http://www.parasoft.com/jsp/products/home.jsp?product=Jtest&/quick.html . Documento en línea 2002.

Vitae

Juan M. Fernández has an undergraduate degree on Physics, and a master degree on Operations Research, both from UNAM, and a doctaorate degree on Computer Science from IPN. Had worked with UNAM, Universidad Autónoma de Baja California and Universidad Veracruzana. His present interests are Software Engineering and Software Technologies.



Perla I. Velasco has an undergraduate degree on Informatics from Universidad Veracruzana, México. She got her master degree in Computer Sciences from Universidad Autónoma de Tlaxcala, México. She is a Associate Researcher from LANIA, A.C. in Xalapa, Veracruz, México, and her present research and application interests are in Software Engineering and Data Bases.