1(leave this here)

Operating and Window Systems will never strike back or Independence day for Java developers

Niels Fricke, Carola Lilienthal, Martin Lippert, Stefan Roock, Henning Wolf
Department of Computer Science
Software Engineering Group
University of Hamburg
Vogt-Kölln-Str. 30
D-22527 Hamburg
+49 40 54 94-2307
{1fricke, lilienth, 4lippert, 1roock, 1wolf}@informatik.uni-hamburg.de

Abstract

With the use of Java and the JDK[*], independence of specific platforms (operating systems and window systems) becomes possible. For the development of commercial applications however, Java and the JDK are not enough, and tools such as a GUI-builder are needed. When employing a GUI-builder the new dependencies on the builder-generated code have to be considered. Independence is an important prerequisite for enabling portability and reusability. There are, on the one hand, many GUI-builders on the market and it is impossible to say which will end up being successful. On the other hand, the integration of existing programs designed with different GUI-builders is an ongoing problem. We have designed and implemented a Java framework which uses a GUI-builder independent format to store GUI resources. This framework can be combined with any GUI-builder that supports JavaBeans.

[*] JDK = Java Development Kit. The actual version is JDK 1.1. The version of the JDK is also the version of the Java language Gosling, Joy, Steele (1997).

Keywords
Java, GUI-builder, Portability, Independence, Framework

INTRODUCTION

Java is a new and exciting technology. More and more software is developed with it and it is included in computer science courses everywhere. It seems to be agreed by almost all that Java is here to stay and we should become involved with this trend right away.

To develop a real world application with Java more than the JDK tools will be required. We will have to find an integrated development environment (IDE) for Java such as those available for C++ and Smalltalk. At the moment the decision as to which IDE to use is a pretty difficult and possibly painful one. If we look at the Java-IDE market, it is obvious that many IDEs are offered by various companies but it is not foreseeable who will still be in the market in the next months or years[*]. Java and the JDK have themselves changed appreciably in the last twelve months. The choice of an IDE will tie us to a company and its success on the market.

To alleviate this problem we have developed a framework in Java that provides IDE-integrated GUI-builder independence (cf. Fricke, Lippert, Roock, Wolf, 1997). During the framework development process we realized that there are several aspects and dimensions to be dealt with when discussing real independence. These will be covered in this article and solutions for each level will be presented. At first we'll examine which parts of our software product depend on which component from an IDE or the Java environment. Different steps towards total independence show how to make the different parts of source code independent and thus truly portable.

When referring to Java specific components the following terms will be used:

- **Java** refers to the Java programming language which was released by Sun Microsystems in 1995.
- **Java-VM**: The term *Java-VM* stands for the Java Virtual Machine which is the interpreter for the byte code generated by the Java compiler. Java programs are collections of portable class files (extension .class) which contain the compiler generated byte code. These class files can be executed on every platform which is supplied with a virtual machine.
- **JDK**: The *JDK* (Java Development Kit) is a set of tools, such as compiler and virtual machine, accompanied by a set of default libraries which are called *standard packages*.

* Java as a source of conflict between Microsoft and SUN Microsystems is another special and serious problem. At the moment SUN is setting the standard specifications for Java while Microsoft as an operating system company is trying to relegate Java to merely another programming language dependent on the Windows platform.

DEPENDENCY

Dependency involves the relationship between two things, whereby one is reliant or dependent on the other. Firstly we isolate, on a general level, the different parts of a software product that are dependent and show why independence is required. The next step is to examine what the software product becomes or is dependent upon in the process of software development and why this could be a problem. With the use of Java and the JDK some independence is already gained. The final part of this section therefore, reduces the number of dependencies to those which are specific to the development with IDEs and the Java environment.

What is dependent?

Basically every application that is coded these days depends on some other program or toolkit. We will use the following classification to split up the various dependent parts of a software product. This classification will be used later to introduce a step by step solution as to how to overcome the specific dependencies. We'll begin with the domain specific parts and move on to the more technical ones.

- The *application program* can be directly executed and is delivered to users who work with the program to complete their tasks. The users are the first to face dependencies. They would like to be independent of the operating and window system, so as to keep up with any new developments in this area. This independence would include the possibility to execute the application program as well as to store objects in files and display them (or their data).
- The *application sources* are developed and maintained by the application developers. The application source files are normally compiled to one or several executables which make up the application program and are delivered to the customer. Because application developers are mainly concerned with transferring the results of domain analyses to the computer they'd really prefer not to deal with dependencies (and naturally they want and need independence to fulfill a user's requirements).
- A *framework* provides a number of basic services which are used by the application developers to write the domain specific code. Frameworks are often developed to encapsulate existing software. A framework can be developed to help achieve independence for the application source. Redeveloping parts of a framework is relatively light work compared to the redevelopment of large parts of an application program. Nevertheless it would be extremely useful to have a framework that is itself portable, because it could be transferred to any platform without changes.
- The *resources* contain additional information that is normally generated by a GUI-builder. If the GUI-builder has a resource format which is portable over several platforms some independence is gained. But still the resources can only be re-read by the same GUI-builder.

What do we depend on?

Throughout our work we discovered several items we usually depend on. We depend on the *operating system* since every operating system has different services which are provided in different ways. File-I/O-services are a good example, where the separator sign for directories is a slash under Unix and a backslash under Windows. Every *window system* has its own widgets, which differ in their look and feel. The operating system and the window system are often subsumed under the term *platform*. For anyone who has been involved with long term software development it is presumably obvious why platform dependence should be avoided. We really cannot know which operating system will be the most popular in 5 or 10 years time and we do not know which new features this operating system will contain. It was to be expected that Microsoft would hold onto its position with the update from Windows 3.1 to Windows 95. What degree of effort was required, however to provide real Windows 95 applications? Consider the amount of work needed to alter the entire platform as opposed to merely the platform 'version'.

The other central item often depended upon, is in most cases ignored - the IDE and its tools. An IDE should offer a GUI-builder to support us with the time consuming tasks of creating the graphical user interface, a configuration management tool, a tool for source code versioning and several other tools that help us with the coding. It would be advantageous to be independent of all of these development tools. It is still a long road, however until we will reach this stage. This article will help in the quest for IDE- independence by looking specifically at the GUI-builders. When using a GUI-builder we have to consider the following aspects of dependency.

- Firstly, all resources will need to be totally rebuilt if we decide to replace the GUI-builder. This tends to encourage us to avoid replacing it but sometimes, of course, it's necessary. Unfortunately, GUI-builders are worse than platforms as far as stability and life span goes (cf. Valaer, Babb, 1997). How long will the provider of our GUI-builder be in the market? Some software development companies even opt to create their own GUI-builders in an attempt to be impervious to such uncertainties. Is it really economical though to spend thousands of dollars on the development of a tool that will usually not reach the quality or productivity of one that can be bought for a few hundred?
- Secondly, rebuilding the resources is the least that could happen. Much more likely would be that our code would have to be rewritten to correspond to the generated code or resources of different GUI-builders. This is because the generated code and the resources of one GUI-builder has an implicit view of the world. We have to be aware of the structure that a GUI-builder forces us to follow, while developing our code. A different GUI-builder with its own assumptions, would probably affect the structure of the whole application.

Several examples of the problems encountered with GUI-builders and program structure can be seen by comparing the common GUI-builders. The SUN Java-Workshop creates a main window (cf. Javasoft 1996c), and closure of this window results in cancellation of the whole application. Other GUI-builders have different approaches e.g., generating one class per window. The SUN Java-Workshop generates only one class per application, so that all the widgets have to be given unambiguous names. Finally, most GUI-builders use a different approach to divide functionality between classes, e.g., turning an existing application into an applet.

Actual Dependencies

Using the JDK eliminates some of the dependencies that would otherwise have occurred with other programming languages.

- The independence of the *application program* from the operating system is achieved by the Java-VM. An application program in Java is a collection of executable java class files which have been compiled to a platform independent byte code. This byte code is interpreted by the Java-VM at runtime. Regardless of which operating system the user employs, the only thing required is to obtain the relevant Java-VM. Nowadays there are virtual machines available for all the popular operating systems. If we want to store objects within a Java program a serialization mechanism can be used which stores values in a strictly defined and portable format.
- To make the *application program* independent of the window system, a Java standard package named AWT (abstract window toolkit) can be employed. This package provides a uniform interface for the window system. With the AWT the application program becomes portable and can be used for all supported window systems.
- The *source code* and the *framework* are portable in different operating and window systems for similar reasons. The Java language definition is a standard developed by SUN. Within the source code and the framework only the Java language is used and there is no need to call the operation system directly. The use of the AWT guarantees the portability of source code and framework.

Table 1 shows the independence gained so far. The cells show which technology provides us with portability. As can be seen there is still work to be done.

Table 1: How independence is achieved with Java and AWT

These ➔ *depends on these* ➘	*Program*	*Source*	*Framework*	*Resource*
Operating System	**Java Virtual Machine (VM)**	**Java Language**	**Java Language**	not supported
Window System	**Abstract Window Toolkit (AWT)**	**AWT**	**AWT**	not supported
GUI-Builder	not required	not supported	not supported	not supported

Unfortunately, SUN did not set a standard on GUI-builders and resource files when inventing the Java language and the AWT. Our source code and the framework will therefore be dependent on the resource files or some code, both generated by a GUI-builder. There is no support available to exchange resources between different GUI-builders. We see two reasons for this. On the one hand most of the available GUI-builders are built on preexisting GUI-builders which were used for other programming languages such as C++. The second reason is that GUI-builder suppliers are usually not interested in offering software developers the opportunity to substitute their GUI-builder for another one.

STEPS TOWARDS INDEPENDENCE

The independence we are looking for should allow us to use any GUI-builder available to create our user interfaces. The resources that describe the user interfaces should be independent from the operating system, the window system and the GUI-builder. If this is achieved, resources designed in one GUI-builder should be readable and changeable in another. The following three steps provide solutions for the different dependencies.

Step 1: Providing a GUI-Framework

The first step in extending the portability of the application source is to use a framework with GUI support. Such a framework would encapsulate the AWT widgets, GUI-resources and GUI-generated source code and provide a uniform view of the GUI for the application program. The rapid development of language and standard packages shows that this is a very important step. The AWT-packages changed substantially from JDK1.0 to JDK1.1 and we had to deal with ‘deprecated’-warnings all along. With JDK1.2 SUN will present the ‘Java

Foundation Classes' which may replace the AWT in the future (although, it is still hard to say). To prevent a massive amount of future change our framework provides an abstract layer. This layer is used by the application source code and by other parts of the framework. With the help of that framework we can write portable application sources which do not depend on the GUI-builders that were used (s. Table 2). A similar step is normally taken in every bona fide software development (c.f. Bäumer, Gryczan, Knoll, Lilienthal, Riehle, Züllighoven, 1997).

Table 2: Enhancing Portability with a GUI-Framework

These ➔ *depends on these* ➘	*Program*	*Source*	*Framework*	*Resource*
Operating System	Java Virtual Machine (VM)	Java Language	Java Language	not supported
Window System	Abstract Window Toolkit (AWT)	AWT	AWT	not supported
GUI-Builder	not required	**Framework**	not supported	not supported

Unfortunately, this is not the whole story. A framework helps us a lot in gaining independence from the GUI-builder but their are some drawbacks. Firstly, we have to rewrite that part of the framework that encapsulates the GUI-builder for every GUI-builder we want to support. Secondly we might not be able to encapsulate all the generated source code. In some cases the GUI-implicit view of the world will still shine through. An example of such an uncooperative GUI-builder is Microsoft's Visual-J++, which generates source code that can not easily be encapsulated because the Visual-J++ generated classes have no common super-class (cf. Microsoft 1996).

Step 2: Providing a Serializer-Bean

What can be done to get properly portable source code and a portable framework? Two features of JDK1.1 Serialization and JavaBeans make a solution possible (cf. Javasoft, 1996b and Javasoft 1996a). JavaBeans is the component technology incorporated in JDK1.1. A JavaBean is a class with a standardized interface which can be used with any GUI-builder that supports JavaBeans. Serialization allows us to store any object which 'implements' the empty interface *Serializable* into a file

and read it again without writing any code. All JavaBeans should 'implement' the *Serializable* interface as well.
We have combined these two features to form a Serializer-Bean. The Serializer-Bean can be added to the component palette of a GUI-builder and can then be used like any other Bean. If we create a user interface with a GUI-builder, the Serializer-Bean is presented on the screen like a normal button (s. Figure 1).

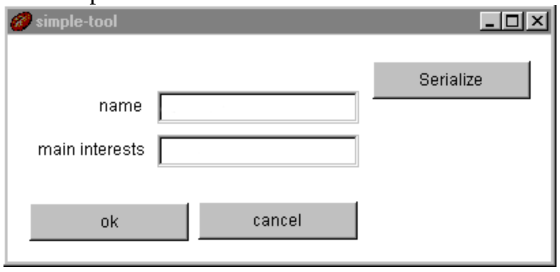


Figure 1: Example in the GUI-builder with Serializer-Bean

When the application developer has finished designing the graphical elements in the user interface, he clicks on the Serializer-Bean. A file- select box pops up and requests a directory and file name. The Serializer-Beans then serializes the user interface into a file using the object serialization provided with JDK 1.1.

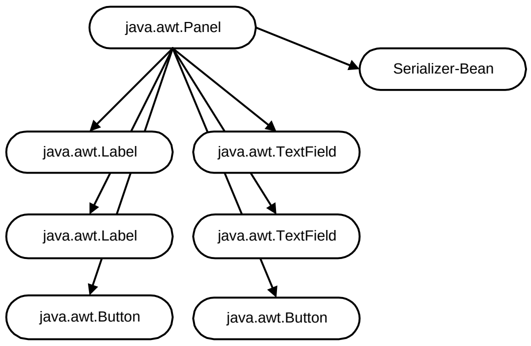
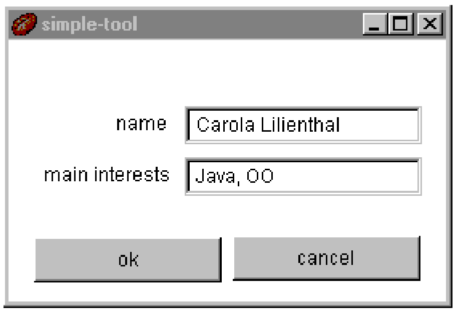


Figure 2: Hierarchy AWT-widgets and Serializer-Bean

Figure 2 shows the hierarchy of AWT-widgets that have been placed in the GUI-builder to form the user interface in Figure 1. The Serializer-Bean takes its parent, the panel (or frame), and writes it with all its dependents into a file. The serialized user interface then contains all Beans (the graphical elements of JDK1.1 are themselves all Beans) that were placed in the user interface as well as the Serializer-Bean itself. When the application is started the serialized user interfaces are loaded by the framework and connected to the application source. The framework hides the Serializer-Bean so that it is invisible when the application is running (s. Figure 3).



name: Carola Lilienthal

main interests: Java, OO

Figure 3: The started application

What sort of independence is gained by applying the Serializer-Bean (s. Table 3)?

- The *resource files* no longer depend on the operating system because they are stored using the independent serialization mechanism of JDK1.1.
- The *resource files* are independent of the window system because the resources only consist of framework and AWT objects. The resource files are transportable to any platform for which a Java-VM exists, providing only standard AWT classes are used.
- The *framework* is no longer dependent on the GUI-builder and the drawbacks of Step 1 are also eliminated. The framework and consequently the source code do not call the GUI-builder generated code anymore. The complete application is composed of framework and source classes as well as serialized AWT objects.

Table 3: Portability with the Serializer-Bean

These ➔ depends on these 🡻	*Program*	*Source*	*Framework*	*Resource*
Operating System	Java Virtual Machine (VM)	Java Language	Java Language	**Seriali-zation**
Window System	Abstract Window Toolkit (AWT)	AWT	AWT	**Serialized AWT Objects**
GUI-Builder	not required	Framework	**Serializer-Bean**	not supported

At this point we have portable resource files which retain the same format for all GUI-builders. We do not therefore have to alter the framework to support additional GUI-builders and we needn't worry about the generated source code. It is no longer significant what kind of source is generated or if source code is generated at all; it's not being used anyway.
There is one major difference between the various GUI-builders that use JavaBeans. This difference influences how the Serializer-Bean can be employed by the developers. Some GUI-builders use proxies for the JavaBeans at design time (e.g., Parts, cf. ObjectShare 1997) and some use objects at design time (e.g., VisualAge for Java (IBM 1997), the BeanBox from the BDK*). Dependent therefore on which GUI-builder is used, it is possible to serialize the user interface at design time, otherwise we have to start the user interface in test mode (which is supported by almost all GUI-builders).
As we can see in Table 3 only one dependency remains. If we want to use user interfaces in one GUI-builder which have been created with another, the resources will have to be recreated in the new GUI-builder. When many complex user interfaces are involved, this can be a very time consuming and error prone task.

Step 3: Providing an Importer-Bean

It would be helpful if the GUI-builders could read serialized objects, however they can't or at least their developers didn't want them to.
There are two possible solutions to this problem. The first requires a fair degree of work for each GUI-builder that is supported, but can then be applied to all. What is needed is a special Bean-Translator for each GUI-builder. This Bean Translator reads a previously serialized user interface and traverses the object structure. On the basis of this object structure, a resource file for the GUI-builder is written. There are two points which make this approach somewhat cumbersome. Firstly, we have to be familiar with the particular resource format of each GUI-builder we want to support. And secondly, we could have problems with the analysis of the object structure because important attributes could be private.
The second approach uses a kind of inverted Serializer-Bean called the Importer-Bean. The Importer-Bean is placed on a user interface just like any other bean. To show how the Importer-Bean could work we have extended the example of Step 2. The new application includes the simple tool shown in Figures 1 and 3. Figure 4 is a screen shot of the new application and the shaded area represents the Importer-Bean. The example of Step 2 is located on top of the Importer-Bean.

* BDK is the Bean Development Kit from SUN (Javasoft, 1996a).

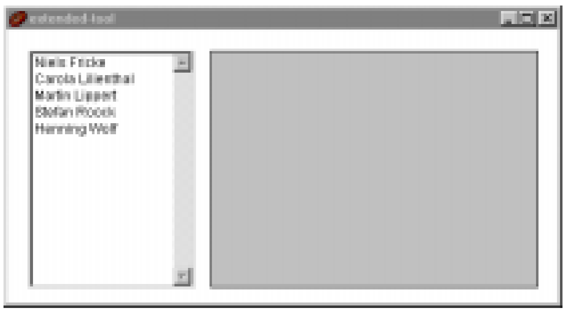


Figure 4: The extended tool with the Importer-Bean (shaded panel)

When the Importer-Bean is clicked a file select box pops up and the developer can choose a serialized file. The Importer-Bean then reads the serialized file and connects the serialized widgets to the user interface. Figure 5 shows the hierarchy of AWT-widgets of the extended tool. The Importer-Bean keeps a reference to the panel, which is the uppermost object within the widget hierarchy.

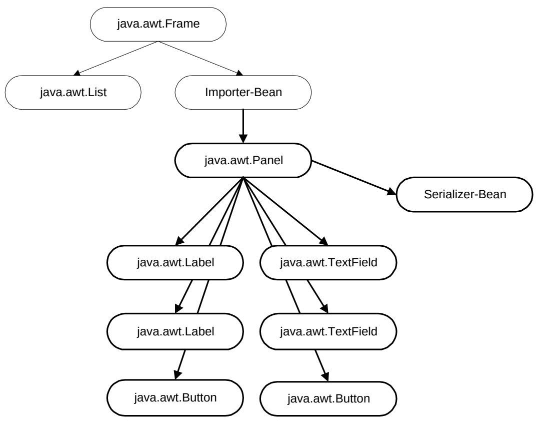
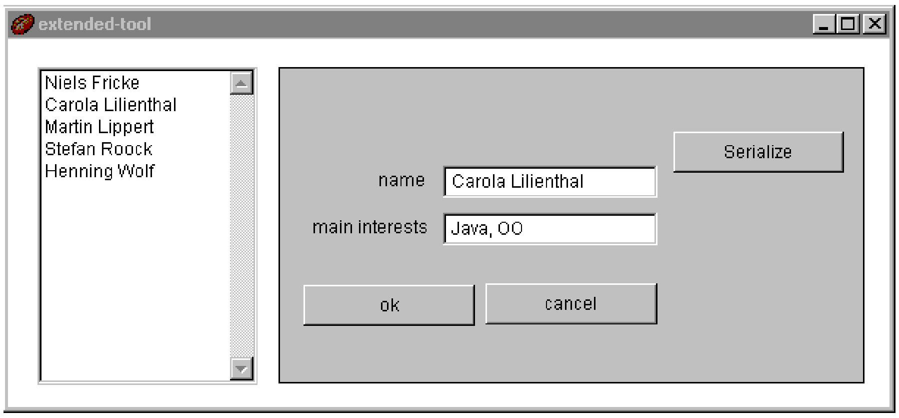


Figure 5: Hierarchy of the extended tool example

In Figure 6 the extended tool can be seen as it would appear in the GUI-builder. This approach requires a GUI-builder that is used to work with living JavaBeans at design time, similar to the BeanBox approach (cf. Javasoft 1997). The Importer-Bean cannot otherwise be activated at design time and the serialized Beans wouldn’t be connected to anything.



Niels Fricke
Carola Lilienthal
Martin Lippert
Stefan Roock
Henning Wolf

name: Carola Lilienthal
main interests: Java, OO

Figure 6: The extended tool in the GUI-builder where the Serializer-Bean is visible

It is hoped that the next generation of GUI-builders will support living JavaBeans at design time. This seems to be justified in that everyone prefers to work with WYSIWYG in not only word processors but in GUI-builders as well.

Table 4: The overall picture

These ➔ *depends on these* 🡻	*Program*	*Source*	*Framework*	*Resource*
Operating System	Java Virtual Machine (VM)	Java Language	Java Language	Seriali-zation
Window System	Abstract Window Toolkit (AWT)	AWT	AWT	Serialized AWT Objects
GUI-Builder	not required	Framework	Serializer-Bean	**Importer-Bean**

Table 4 shows the actual situation, we have now achieved *real portability with java*.

LEVELS OF PORTABILITY

Up to now we have introduced a framework, the Serializer-Bean and the Importer-Bean. Bearing in mind that the situation is now being supported by a Serializer- and Importer-Bean, do we still really need a framework to achieve portability?

As a matter of fact, the framework is no longer required. The loading of the serialized user interfaces carried out by the framework could easily be handled by the application source itself and the application source could work directly on the AWT objects. A framework however, is still very useful when we consider the reuse of code or concepts (s. Bäumer, Gryczan, Knoll, Lilienthal, Riehle, Züllighoven, 1997).

We could use a framework to support other kinds of portability. What will a user interface look like in ten or twenty years time? Will we still be using a screen, keyboard or mouse? Perhaps some kind of data glove or suit will be used. If the general structure of user interfaces changes, the portability as described above will be at an end. Large parts of our applications will need to be rewritten from scratch. A good way to minimize the work would be to divide any user interface into interactive and functional parts (cf. Riehle, Züllighoven, 1995 and Strunk, Fröse, 1996), or a model and a view if Smalltalk and MFC are used. If the general structure of user interfaces changes, only the interactive parts or the views have to be rewritten. This division can and should be supported by a framework.

Table 4: This Table shows some of the tools that were tested with the Serializer- and Importer-Beans. As can be seen, Bean support in GUI-builder and JDK version 1.1.x (or higher) is necessary, and here the Serializer Bean works. The Importer Bean works in some, but editing the imported Beans is not yet possible with every tool. Only the Java Workshop requires additional classes (i.e., more than just the classes from the Java standard packages), others have optional additional classes (such as special Layout-classes).

	Java-Workshop	*Visual Café*	*JBuilder*	*Parts for Java*	*Visual Age for Java*	*PowerJ*	*J++*	*Code Warrior*
Company	Sun	Symantec	Borland	ParcPlace	IBM	Sybase	Micro-soft	Metro-werks
Version	2.0	2.0	1.0	2.0	1.0	2.1	1.1	Profess. 1
JDK version	1.1.x	1.1.x	1.1.x	1.1.x	1.1.x	1.1.x	1.0	1.0.x
Platform	Unix (Sparc), Windows 95/NT	Windows 95/NT	Windows 95/NT	Windows 95/NT	Windows 95/NT, Unix	Windows 95/NT	Windows 95/NT	Windows 95/NT, Macintosh
Bean-Support in GUI-builder	yes	yes	yes	yes	yes	yes	no	no
Serializer-Bean	works at design-time and run-time (with patched jws-classes)	works at run-time	works at design-time and run-time	works at run-time	works at run-time	works at run-time (with patched powerj-classes)	no	no
Importer-Bean	works without editing	doesn’t work	works without editing	works without editing	doesn’t work	doesn’t work	no	no
Additional-classes	special Java-WorkShop-classes (must be patched)	optional	optional	optional	no	powerj-specific classes has to be patched	no	no

Another question has already been raised, namely how long will the AWT remain unchanged? And how different will a new Java toolkit such as the Java Foundation Classes be? By encapsulating the AWT with a framework we will only have to change the framework and not the entire application source. On the basis of these thoughts two general layers of Java portability can be defined.

- lightweight portability is achieved by the use of the Serializer-Bean and the Importer-Bean
- general portability can be achieved by the use of the Serializer-Bean, the Importer-Bean and a framework.

CONCLUSION

In this article we have defined different levels of dependency for the various parts of a software product. We have also discussed what the software product becomes or is dependent upon during the software development process. Java and its standard packages already offer us some portability. The GUI-builders and their generated resources still however create some strong dependencies. It was shown how real portability can be achieved by using a framework and some of the built-in Java features. We are now able to develop software products that are not only independent of the operating system and the window system but of the GUI-builder as well.

ACKNOWLEDGMENTS

We’d like to thank Frank Fröse and Charles MacInnes for their valuable hints.

REFERENCES

Bäumer, D., Gryczan, G., Knoll, R., Lilienthal, C., Riehle, D. and Züllighoven, H. (1997) Framework Development for Large Systems. Communications of the ACM, October 97, Vol. 40, No. 10, pp.52-59.

Fricke, N., Lippert, M., Roock, S. and Wolf, H. (1997) Java Framework. University of Hamburg, Department of Computer ScienceSoftware Engineering Group. http://swt-www.informatik.uni-hamburg.de/~Software/JWAMV1.0/ in German.

Gosling, J., Joy, B. and Steele, G. (1997) the Java language specification, Reading, MA: Addison-Wesley. http://java.sun.com/products/jdk/1.1/download-pdf-ps.html.

IBM (1997) http://www.software.ibm.com/ad/vajava/

Javasoft (1996a) JavaBeans 1.01 API Specification. SUN Microsystems. http://java.sun.com/ beans/spec.html.

Javasoft (1996b) Java Object Serialization Specification. SUN Microsystems.

Javasoft (1996c) Java-WorkShop. SUN Mircrosystems. http://www.sun.com/workshop/java/.

Javasoft (1997) BeanBox. SUN Microsystems. http://java.sun.com/beans/beanbox.html.

Microsoft (1996) Visual J++. http://www.microsoft.com/visualj/.

Nulden, U. (1997) The Why, What, and How of Reuse in Software Development. In Kristin Braa, Eric Monteiro (editors), Proceedings of Iris’20: Social Informatics, pp.407-414.

ObjectShare (1997) Parts for Java. http://www.objectshare.com/p4j/p4j2info.htm.

Riehle, D. and Züllighoven, H. (1995) A Pattern Language for Tool Construction and Integration Based on the Tools & Material Metaphor. In J.O. Coplien, D.C. Schmidt Pattern Languages of Program Design. Addison-Wesley, Reading, pp. 9-42.

Strunk, W. and Fröse, F. (1996) Using Design Patterns to Restructure the User Interface Part of an Application Framework. Theory and Practice of Object Systems 2, 1 (1996), pp. 53-60.

Valaer, L.A. and Babb, R.G. II (1997) Choosing a User Interface Development Tool. IEEE Software. Vol. 14, No. 4 , July-August 1997.

BIOGRAPHY

Carola Lilienthal is a research assistent at the University of Hamburg since August 1995. She is a member of the software engineering group. Before studying computer science Miss Lilienthal worked for two years at a German private bank. Her area of expertise are design pattern, frameworks as well as evolutionary and document-driven software development. Miss Lilienthal is a consultant in the area of object-oriented software development and has published several articles.

Niels Fricke, Martin Lippert, Stefan Roock and Henning Wolf are preparing their graduation in Computer Science at the University of Hamburg. They are all studying and working in the area of object-oriented software development.