JPloy: User-Centric Deployment Support in a Component Platform

Chris Lüer and André van der Hoek

School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
USA
{chl,andre}@ics.uci.edu

Abstract. Based on a vision that, in the future, applications will be flexibly built out of small-grained components, we argue that current technologies do not adequately support component deployment in such a setting. Specifically, current technologies realize deployment processes where most decisions are made by the application manufacturer. When using small-grained components, however, the component user needs to have more control over the deployment process; *user-centric deployment* is needed. In this paper, we describe our initial efforts at providing user-centric deployment. We present JPloy, a prototypical tool that gives a user more control about the configuration of installed Java components. JPloy extends the Java class loader so that custom configurations can be applied to existing components, without having to modify the components themselves. For example, name space or versioning conflicts among components can be elegantly resolved in this way. We demonstrate JPloy by applying it to an example application.

1 Introduction

It is assumed that in the future, applications will consist out of large numbers of independently developed, reusable components. The role of the *application builder* will emerge (see Figure 1): application builders compose applications out of reusable components licensed from a number of different component manufacturers. In such settings, application builders need to have control over the deployment of individual components, so that they can determine the structure of the application that is built. Furthermore, they should not have to program extensive amounts of code, rather, they should be able to *compose* applications. To enable this kind of user-centric composition, new deployment technologies are needed.

Current support for the deployment and composition of component-based applications can be divided into two kinds of approaches: component platforms and deployment tools. Current component platforms [15], such as Java [3], Dotnet [6], Koala [22], or ArchStudio [19], make it possible to configure applications at the user's site (i.e., they support deployable components). The process of installing and configuring components, however, is manual and error-prone. Current deployment tools, on the

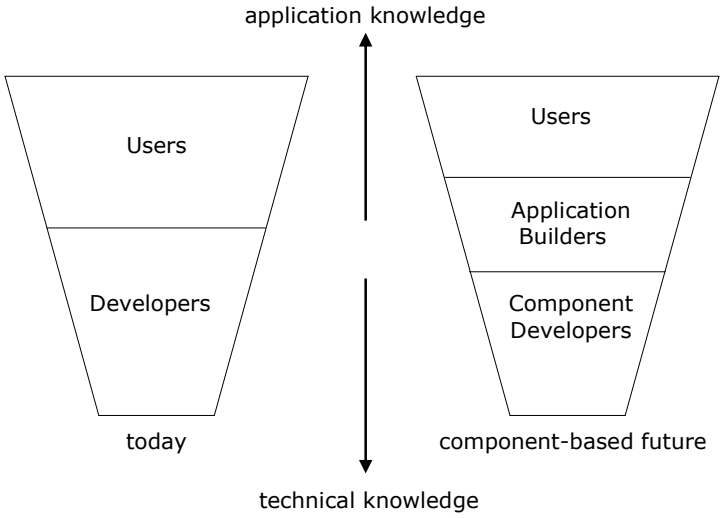

Fig. 1. Traditional and component-based development scenarios.

other hand, do not support deployable components; they assume that applications are configured mainly at the manufacturer's site.

We believe that there is a need for a blending of the two approaches, in effect creating a deployment technology for component platforms that is *user-centric*. This would put full control of the application structure into the hands of an application builder located at the user's site, and thus would leverage the full potential of deployable components. As a consequence of the shift from manufacturer-centric to user-centric deployment, deployment support has to move from a dedicated deployment tool (such as an installer) into the component platform. This is because, in a manufacturer-centric setting, deployment is an occasional activity; it is performed only when a new product version is available or when the user's computer system changes. In a setting with small-grained deployable components, activities such as installing and configuring components must be performed continuously. For example, if an application consists of large numbers of components, updates for individual components will frequently be available. Even if a user chooses to perform these updates manually at a later point of time, the number of components and the complexity of their relationships will make tool support indispensable.

In this paper, we present JPloy, a tool that provides deployment support for Java components. JPloy extends the Java runtime environment so that it can read in *configuration files*, which are used to connect and adapt deployed components. In particular, configuration files specify the usage relations between components, and how components are modified at runtime. The remainder of this paper is structured as follows. In Section 2, we discuss the background of deployment technologies. Section 3

identifies what we consider core requirements for a user-centric deployment technology, and Section 4 presents our approach, JPloy. Section 5 demonstrates how we applied JPloy to an example application and addressed two critical deployment problems of this application. Section 6 evaluates our approach, Section 7 discusses related approaches, and Section 8 gives an overview of future work.

2 Background

Deployment consists of those activities that need to be performed with a software product after it has been released [11]. Deployment includes installing, configuring, and updating the program or component; its goal is to make it possible to execute the program on a user's computer. Current deployment approaches are *manufacturer-centric*; this means that the way in which a program is deployed is largely determined by the manufacturer of the program. For example, the manufacturer typically provides an installation script that performs the automatic deployment of the software. If a program depends on other programs or components, these dependencies are typically specified in a rigid way.

Historically, component-based software development has had three stages [26]. In the first stage, source code components were reused. These components disappear after compilation and thus have no influence on deployment activities. In the second historical stage, the manufacturer can link binary components into different products from a product line. Current deployment tools (for example, the Software Dock [11] [13], Bark [24], or RPM [5]) are designed to support this stage of component technology. They assume that all available forms of an application are determined before it is shipped to the user's site, and hence the user's control over the application structure is limited. All deployment activities are controlled by the manufacturer. Typical deployment activities in such a setting are:

- *Release:* the activity of ending development and making the software product ready to be shipped.
- *Retire:* its opposite, performed when support for a product is terminated.
- *Install:* the activity of copying a product to a user's machine and preparing it so that it can be used.
- *Remove:* its opposite, removing all traces of a product from the user's machine.
- *Update:* a partial remove and install; typically used when a new version of the software product is released and downloaded.

Current and future component systems, however, must be designed to support *deployable components* (stage 3) [15]: components that can be deployed individually, and that are composed into different applications at the user's site. This means, that in this stage, the two steps of configuring and shipping code have been swapped: first, components are shipped to the user; only afterwards, they are configured into applications.

3 Requirements

To make user-centric deployment a reality, certain requirements must be met. In this section, we identify three requirements that we consider to be at the core of any solution to user-centric deployment. *Interference-free deployment* requires that deploying one component should not change the behavior of other deployed components. *Independent deployment* requires that components can be deployed, as far as possible, independently from each other; especially, this criterion implies absence of strict dependencies. Further, user-centric deployment requires *compatibility with legacy components*.

3.1 Interference-Free Deployment

Installation and configuration of components should not change the behavior of already installed components. This is a general requirement of deployment technologies and this holds as well in user-centric deployment. As a consequence, installing and configuring of components needs to be performed without side-effects or irreversible changes [9].

To realize interference-free deployment, a component platform must not:

- overwrite information in configuration files during install or configuration;
- occupy a location in a namespace that is also used by a previously installed component; for example: a component uses the same name as another one, so that one of the two components cannot be accessed any longer; and
- change global properties of a system such as environment variables or class paths.

A special case of interference-free deployment is concurrent deployment of versions. Since two components in one applications may require different versions of the same component (or, in the general case, two different applications use two different versions of the same component), it has to be possible to install and configure several versions of one component at the same time [20].

3.2 Independent Deployability and Absence of Strict Dependencies

Since components are units of deployment, there should be no strict dependencies among them. A strict dependency is a dependency that can be resolved only by one component. Instead, if a component requires another component, this requirement has to be formulated in terms that allow several options of how to fulfill requirements, thereby changing the model from manufacturer-centric to user-centric deployment. In particular, the prohibition of strict dependencies is needed to enable competition among component developers, and to give full control over which components are employed in an application to the application builder.

In the absence of strict dependencies, a component platform needs to provide mechanisms to specify dependencies that are not based on component identities. I.e., a dependency has to be specified on a higher level than the level of component names.

It must be possible to deploy each component individually; and the application builder at the user's site must be able to decide on the concrete configuration of an application.

An implication of independent deployability is the need to deploy configurations (or, architectures) independently from components. Since components cannot contain strict dependencies, the actual instantiations of component interactions have to be stored elsewhere.

3.3 Compatibility with Legacy Code

A platform that is not backwards-compatible to legacy code will be impracticable due to the large number of components that already exist. A component deployment platform needs to be able to support legacy code without much effort on the application builder's part. For example, legacy code may be wrapped in some way to make it compatible to the platform. It may be possible to create such wrappers in part automatically. A consequence of backwards compatibility is that user-centric deployment support may not reduce the expressiveness of the platform, i.e., it cannot restrict the space of valid executable files.

3.4 Other Requirements

We can certainly identify many additional requirements that build upon the core requirements. Two interesting such requirements are transparent updating and incremental builds. Transparent updating requires a mechanism that can automatically install and configure updates of deployed components; since a user's system may consist of a very large number of components, manual updating would not be feasible. Incremental builds requires the ability to configure and test partial applications. For the purpose of this paper, we focus on the core requirements in order to establish a base on which we and others can build.

4 The JPloy Approach

4.1 Motivation of the Approach

Support for component deployment could be provided either by a programming language, by external tools, or by a component platform. Language level support has the disadvantage that deployment information will be lost after compiling. Since components are generally shipped in compiled form, programming language support would not be helpful in assisting an application builder. External tools can modify existing components by editing their binary representation; but this is generally undesirable since it has the potential to create a large number of different binary component versions. This leaves us with the component platform as the optimal location for deployment support: the component platform can modify compiled components at run-

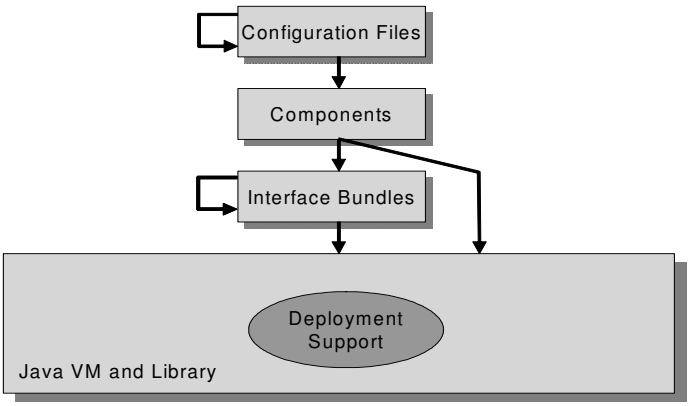

Fig. 2. Architecture of a JPloy application. Arrows represent *uses* relations.

time and on-the-fly, without requiring source access and without multiplying the number of component files.

There are two possible ways to add deployment support to a component platform: extending the application programming interface (API) of the platform, or modifying the loading and linking mechanisms of the platform. Additional tools that are external to the platform can be used with both approaches.

When extending the API of the platform, byte code instructions or operations in the base library are added. This makes it possible for components to call platform functionality that ensures proper installation and configuration. But for this approach to work, components need to be required to call this functionality at the appropriate times. This may be hard to enforce, and makes the use of legacy components that were not specifically written for the platform impossible.

We choose the second approach: extending the loading and linking mechanisms. This approach has the advantage that all modifications to the platform are transparent to components. This simplifies the development of new components and at the same time enables the use of legacy components.

To implement the approach, we had to select a component platform. Currently there are two industrial-quality component platforms, Dotnet [6] and Java [27]. We selected the Java platform, since it provides a well-documented solution for extending its loading and linking mechanism: the Java class loader [14]. Our solution extends the Java class loader, so that it can read in configuration files and modify binary code (i.e., Java class files) in an appropriate manner while the code is being loaded into memory.

Figure 2 shows the architecture of a running system under this approach. At the bottom, one can see the Java platform, which consists of a virtual machine and a class library. Inserted into the platform is the deployment support module, which extends the functionality provided by the platform. In particular, the deployment support module interprets configuration files and interface bundles to instantiate a desired application.

4.2 Extended Class Loader

The core technology of our approach is an extended class loader. It is able to avoid name clashes by renaming classes at runtime, to adapt classes and operations when needed, and to handle different deployment units.

Whenever a component is loaded as specified in the configuration file, the renaming class loader changes the names of classes that may create name clashes to appropriate new values. In effect, this separates the namespace used to identify components (and entities contained in components, such as interfaces) at composition time from the namespace used to identify components at runtime. For example, two versions of one component are considered equivalent at composition time because they both implement the same specification. At runtime, differences in behavior between the two components versions may occur, and so they are treated as different components there.

A *deployment unit* is a file that can be deployed with our system. We distinguish three kinds of deployment units, each of which has to be treated somewhat differently by the class loader: interface bundles, components, and configurations.

An *interface bundle* is a collection of Java interfaces. The interfaces are used to specify components in a given domain. Public interfaces need to be deployed separately from components since each interface can be implemented by several classes contained in several components. Thus, an interface bundle specifies data types in a domain, but does not contain any implementation code.

A *component* is a collection of class files, typically in the form of a jar file. Any Java class files can be used, including legacy code. All public classes in the component are by default its provided services. All classes that are used by classes in the component and are not contained in the component are its required services.

Any name in a component that refers to an external entity (such as a class or method that is not defined inside the component) is subject to renaming by the extended class loader. For example, if class C refers to class D, and D is not defined in the same component, the class loader uses a configuration file to determine which class in which component is supposed to stand in for D. The class loader will then, during loading time, replace all references to D with references to the stand-in.

A *configuration* is a file that specifies how a set of components interoperate. It lists 1) the components that are needed, 2) how they are connected, and 3) optionally, adaptations. Configurations are created and edited by application builders in order to define applications, exchange components in an existing application, etc. Thus, they are stored in a human-readable form. Each component may be used in any number of configurations. When the extended class loader encounters a configuration, it treats it as a set of instructions on how to treat entities that occur inside of components.

The conceptual difference between a component and a configuration is that components cannot refer to other components or to configurations. They refer only to programming-language entities such as classes, and all matching between the names of classes outside of a component and the components that provide an implementation to them will be done by the extended class loader. Configurations, on the other hand, can refer to components and to other configurations. Components can refer to interface bundles, though; and interface bundles can refer to other interface bundles. Referring

to interface bundles is needed to enable type checking as normally done by a typed component platform.

4.3 Configuration Language

The syntax of configuration files currently consists out of the following commands; see Figure 3 for an example configuration.

1. A command for importing a new component and assigning it an alias (lines 1-5 in the example). All components that are imported in this way can be referred to in other parts of the configuration, and may be executed as part of the application. Currently, components are identified by a local file name. In a later version of the system, a uniform resource identifier will be used instead, so that configurations will be independent of location.
2. The `main` command (line 7). It is needed to declare the entry point for execution; i.e., with which component execution starts when the configuration is executed. The main component is required to include a main class, i.e., a class that has an operation with the signature of a Java main method. The information which class in the main component is the main class is taken from the component's manifest; alternatively the name of the main class can be given as a parameter to the "main" command.
3. The `use` command (lines 9-12) defines a usage relationship between two components; a set of `use` commands thus defines the architecture of the application. `A use B` means that `A` is allowed to access features provided by `B`. Without such a command, all attempts by `A` to use features in `B` will fail. However, it is not enforced that A actually uses any of the features in `B`. In a sense, this command is a replacement for the Java class path, which is a simple way to declare which components provide the features needed by an application. The disadvantage of the class path is that it does not allow to define individual relations between components; all relations are all-to-all (every component that is part of the application can access any other).
4. The `replace` command (line 14) removes a class from a component, and replaces all references to it by something else. This allows an application builder to replace parts of a component that are not compatible with the current application, and to replace them by alternatives (see Section 5.2). The command syntax is: *`requiring_component`* `replace` *`old_class new_class`*`.` This will cause two actions to happen: one, *`old_class`* is never loaded from component *`requiring_component`*. Two, all references to *`old_class`* in *`requiring_component`* will be replaced by references to *`new_class`*, which will typically not be located in *`requiring_component`*, but in one of the components that provide features to it (example: *`new_class`* is located in component *`B`*, and there is a command: *`requiring_component`* `use` *`B`*).
5. The `wrap` command (not shown) wraps a feature (a class or operation) with another name whenever it is used by the given component. In a case where component `A` expects a feature called `X`, and component `B` provides this feature under the name `Y`, the `wrap` command provides a simple way to resolve this problem. The

command is equivalent to writing a wrapper class; it serves as a simple means to adapt a component in the case of an incompatibility.

Naming conflicts between components are avoided automatically; this does not require the use of a special command. This means that, whenever a component is loaded, the extended class loader checks whether its names conflict with those of any previously loaded components, and if so, all concerned classes are automatically renamed.

5 Example

In the following, we will detail two exemplary situations that illustrate how JPloy can be used to solve practical software development problems. Problem 1 deals with concurrent versioning; JPloy is used to deploy an older and a newer version of the same component simultaneously. Problem 2 addresses the architectural modification of an existing application; here JPloy allows a reconfiguration of the application without source code manipulation.

5.1 Problem 1: Concurrent Versioning

We recently developed a component-based developed environment called Wren [17], which uses the Argo/UML [1] graphical design tool, and the xArch [8] architecture description language. When integrating these technologies, a version conflict occurred. Both Argo and xArch use XML files to store their configuration; but Argo (in the version employed) uses an older version of the same library. As a consequence, it was not possible to integrate the application as planned. It is not possible on the Java platform to load two different packages with the same name simultaneously; and it was not possible to replace one of the two versions by the other, because the two versions were not compatible with each other.

To solve this problem, a separation between the development-time component namespace and the deployment-time component namespace is needed. It is useful that the two implementations of the XML component have the same name at development time, since they are intended to solve the same problem, and to implement the same specification. A developer would not choose to use both at the same time, instead he/she would settle on the newer version. But at deployment time, this choice may not exist; a separate namespace is needed to avoid problems like this. Thus, each component implementation should have its own identifier at deployment time, no matter which specification it implements.

Using JPloy, we can specify the architecture of the application as a uses-graph (see Fig. 3 for the JPloy notation). This allows us to map different versions of the same package to different requirers: the Argo component is paired with the (older) Xerces version of the XML component, while the xArch component is implicitly paired with the newer version of the XML component that is contained in the Java platform library (since component `xarch` has no `use`-command that connects it to an XML component, the only place where it can access XML features is in the platform).

```
1    wren       = c:\wren\wrenclient.jar
2    argo       = c:\wren\argouml.jar
3    xarch      = c:\wren\xarchlibs.jar
4    xerces     = c:\wren\xerces.jar
5    argoinit   = c:\wren\argoinit.jar
6
7    wren main
8
9    wren use argo
10   wren use xarch
11   argo use xerces
12   argo use argoinit
13
14   argo replace
         org.argouml.application.PreloadClasses
         edu.uci.wren.PreloadClasses
```

Fig. 3. JPloy configuration for the example application.

At runtime of the application, the JPloy class loader checks, whenever a class from the XML package is requested, which component is requesting it. If the requester is a part of Argo or of Xerces, the class is loaded from Xerces; if the requester is a part of xArch, then the class is loaded from the platform library.

5.2 Problem 2: Reconfiguration of a Third-Party Application

In the Wren project from example 1, we reused code from the Argo project for drawing customized design diagrams. Since Argo is a full-featured UML tool, only a small part of its diagramming functionality was needed. Unfortunately, Argo performs extensive initializations at program start time in order to avoid performance overheads later; but since we were reusing only a small part of its functionality, most of this lengthy initialization was unneeded in our project. We decided to modify Argo's code in this respect; this did not prove very difficult, because Argo is a well-documented open source project.

However, modifying Argo in this way created a maintenance and deployment problem. The code modification could not be integrated into the standard Argo code base, since it contradicted the purpose of the project (providing full-featured UML modeling support); it applied only to our use of it. Hence, both the original Argo component and the modified one had to be maintained and deployed separately; each time a new Argo version was released, the modification had to be edited in manually. Also, the code distribution was redundant: people using both Wren and Argo had to have available two almost, but not quite, identical copies of the Argo component.

To solve this problem, we used a JPloy configuration that contained the `replace` command: we replaced the initialization class in the Argo component (`org.argouml.application.PreloadClasses`) with an abbreviated version (`edu.uci.wren.PreloadClasses`; see Figure 3). The effect of this is that the Argo component is loaded without the original initialization class, and instead it

requests this class from component `argoinit`, which was written by us and only contains the simplified initialization class. In this way, when the program is executed, we save the initialization overhead that is normally needed for the Argo component to work, but which is not needed for our specific use of it.

This example shows how JPloy configurations can be used to override architectural constraints that are embedded in the code of a component. Doing so requires some knowledge of the internal structure of the component, but so does modifying the source code. The advantage of our approach, though, is that no maintenance problem occurs, and that the component can be simultaneously deployed both with and without the modification.

6 Evaluation of the Approach

JPloy is an initial implementation of a deployment tool. In this section, we briefly discuss how it compares to the requirements stated in Section 3, and what its limitations are.

Requirement: interference-free deployment. JPloy's capability to automatically avoid name conflicts solves an important part of the problem of interference-free deployment. There is no case in which a component cannot be installed because another component that conflicts with it is already installed. Also, the strict separation between components and configurations prevents many possible side-effects of deployment operations. For example, in some system an update operation may accidentally remove components that are still needed by some applications.

However, there are other aspects of interference-free deployment. For example, when a component modifies an operating system variable, this will interfere with the behavior of other installed components. More research is needed to determine how these kinds of interferences between components can be avoided.

Requirement: independent deployability. Independent deployability is given by the fact that every component can be replaced by another, compatible one. There is a strict separation between specifications and components; for any specification, there may be any number of components implementing it, so that any component is replaceable. When a component refers to features from another component in its code, this dependency can always be remapped to another component at runtime.

To replace a given component used in an application, all that needs to be done is to replace the identifier of the old component with the identifier of the new component in the application's configuration file.

Requirement: compatibility to legacy components. This is given through the transparency of the JPloy approach. Most legacy components will continue to work without change; the only exception is components that use reflection. A renaming class loader cannot correctly work with reflection, because reflection allows Turing-complete modifications of code.

We note that a main limitation of the current system is its reliance on close compatibility of the components that are used together (as with many CBSE approaches to date). In practice, glue code is often needed to make components that are not com-

pletely compatible interoperate with each other. See Section 8 for a discussion on how we believe that JPloy lays a basis for an effective and elegant solution to this problem.

7 Related Work

In the following paragraphs, we briefly summarize the most relevant related systems.

Jiazzi [18] is a linker that can compose applications out of Java components and Jiazzi scripts. The scripts determine the structure of the application, and according to the information in the scripts, the linker will modify the Java class files of the components so that together they form the desired application. JPloy has in common with Jiazzi the ability to impose a configuration on a set of preexisting components. The disadvantage of Jiazzi, however, is that it modifies the actual component files, thus creating a number of variants of each component in the file system. JPloy does not modify files; all modifications happen exclusively at runtime.

MJ [7] is an extension of the Java platform that introduces a module concept. Modules are defined in module files; each module consists out of a number of classes and has well-defined uses relations. Modules are enforced at run-time by an extended class loader. MJ utilizes a similar technique as JPloy, but it does not have the focus on user-centric deployment.

Eclipse [2] is an integrated development environment for Java with an advanced plug-in concept. A plug-in is a component that can be integrated into the environment. Eclipse has in common with JPloy the ability to define use-relations between components. It accomplishes this by giving each component a metadata file that describes which other components it is allowed to use. Unlike JPloy, the Eclipse model is limited to acyclic dependency graphs.

Dotnet [6] is the core programming platform of recent Microsoft operating systems. It includes a component concept called *assembly*. Dotnet has the ability to concurrently deploy multiple versions of the same component. However, it does not support independent deployability; a component is typically tightly linked to components that provide services to it.

Hnětynka and Tůma [12] discuss name clashes on the Java platform in detail. They present a solution based on the Sofa component model that employs a renaming class loader to automatically avoid name clashes in a way similar to our approach. However, their solution does not address other deployment issues.

OSGI [4] is a standard for home appliance software components; it is implemented among others by the Oscar [10] project. OSGI provides capabilities for connecting components by extended class loaders; especially, it allows components to define uses-relations among themselves.

8 Future Directions

We believe that the our approach presents a powerful basis for component connection and adaptation at deployment time. Two areas of extensions to the approach are inter-

action styles and glue code; the concepts of the extended class loader and of configuration files will make integration of these extensions straightforward.

Interaction styles are restrictions on the way components may interact. They are known from the field of software architecture [25], where they form part of architectural styles. For example, the event-based style (also known as indirect invocation [21]) postulates that all communication must be performed through events; subscribers subscribe to event sources, and are notified when new events are published. Interaction styles are often essential for the architecture of an application. Hence, when using a given premanufactured component in a certain architecture, it may be necessary to adapt the interaction style of the component to the one required by the architecture. When done manually (assuming source code is available), this may be a labor-intensive task; when switching from a procedural to an event-based interaction style, all procedure calls that cross component boundaries need to be modified.

We plan to extend the JPloy configuration language to allow users to specify an interaction style for a configuration. Then, the JPloy runtime environment will generate the appropriate interaction code and enforce the interaction style when the components are deployed. While not all possible interaction styles can be automatically generated and enforced, we expect that our approach will be able to cover a selection of relevant interaction styles.

Glue code is the common name for code that is written to make independently developed components interoperate. It resolves the different syntactic and semantic incompatibilities that typically exist between components without adding much functionality.

Glue code is often considered essential for application composition, because it is unlikely that two components from different sources will be completely compatible with each other. But at the same time, glue code is often highly mechanical in nature, consisting only out of simple mappings between syntactically incompatible interfaces with the same underlying semantics. We will extend JPloy to generate glue code for syntactical component mismatches. The configuration language will also include support for easily specifying small parts of code that can be used in cases where the automatically generated code is insufficient.

The run-time code manipulation capabilities of a JPloy-like tool can potentially be useful for other problem areas besides component deployment, for example program compaction [16]. Subject-oriented programming tools such as HyperJ [23] provide similar capabilities for the purpose of multidimensional separation of concerns.

9 Conclusions

In this paper, we defined user-centric deployment and argued that current deployment approaches are insufficient in this respect. We identified a set of requirements for a possible solution, and presented JPloy, an initial approach towards addressing user-centric deployment in the Java platform. JPloy realizes interference-free deployment and independent deployability of components by extending Java's mechanism for loading compiled code into memory. JPloy does not require source code access and works with legacy components.

We applied JPloy to two deployment problems (concurrent versioning, and reconfiguration of a legacy application), and showed how it can be used in these situations. While we certainly recognize that JPloy is not a complete solution at this moment in time, we believe that its approach establishes a solid core for addressing the full spectrum of issues in a user-centric deployment environment. Our future work will build upon this core and work towards providing capabilities for injecting interaction styles and glue code into applications.

References

1. Argo/UML, http://argouml.tigris.org/.
2. Eclipse.org, http://www.eclipse.org/.
3. Java 2 Platform, Standard Edition, v 1.4.0 API Specification, 2002. http://java.sun.com/j2se/1.4/docs/api/index.html.
4. Open Services Gateway Initiative, http://www.osgi.org/.
5. RPM Package Manager, http://www.rpm.org/.
6. Box, D. *Essential .NET*. Addison-Wesley, Boston, 2002.
7. Corwin, J., Bacon, D.F., Grove, D. and Murthy, C. MJ: A Rational Module System for Java and its Applications. In *OOPSLA 2003: 18th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM, New York, 2003, 241-253.
8. Dashofy, E.M., van der Hoek, A. and Taylor, R.N. An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages. In *Proceedings of the ICSE 2002 International Conference on Software Engineering*, ACM, New York, 2002.
9. Dolstra, E., Visser, E. and Jonge, M.d. Imposing a Memory Management Discipline on Software Deployment. In *International Conference on Software Engineering (ICSE)*, 2004, to appear.
10. Hall, R.S. and Cervantes, H. An OSGi Implementation and Experience Report. In *2004 IEEE Consumer Communication and Networking Conference*, 2004.
11. Hall, R.S., Heimbigner, D. and Wolf, A.L. A Cooperative Approach to Support Software Deployment Using the Software Dock. In *Proc. 1999 International Conference on Software Engineering*, ACM, New York, 1999, 174-183.
12. Hnetynka, P. and Tuma, P. Fighting Class Name Clashes in Java Component Systems. In *Proceedings JMLC 2003*, Springer, 2003.
13. van der Hoek, A., Hall, R.S., Heimbigner, D. and Wolf, A.L. Software Release Management. In *Proceedings of the Sixth European Software Engineering Conference*, Springer, Berlin, 1997, 159-175.
14. Liang, S. and Bracha, G. Dynamic Class Loading in the Java Virtual Machine. *Sigplan Notices*, *33* (10), 1998. 36-44.
15. Lüer, C. and van der Hoek, A. Composition Environments for Deployable Software Components, Technical Report UCI-ICS-02-18, Dept. of Information and Computer Science, University of California, Irvine, 2002.

16. Lüer, C. and Hoek, A.v.d. Architecture-Based Program Compaction. In *First Workshop on Reuse in Constrained Environments (RICE 2003) at OOPSLA 2003*, Anaheim, California, 2003.
17. Lüer, C. and Rosenblum, D.S. Wren—An Environment for Component-Based Development. *Software Engineering Notes*, *26* (5), 2001. 207-217.
18. McDirmid, S., Flatt, M. and Hsieh, W.C. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001.
19. Medvidovic, N., Oreizy, P., Taylor, R.N., Khare, R. and Guntersdorfer, M. An Architecture-Centered Approach to Software Environment Integration, Technical Report UCI-ICS-00-11, University of California, Irvine, Irvine, 2000.
20. Meijer, E. and Szyperski, C. Overcoming Independent Extensibility Challenges. *Communications of the ACM*, *45* (10), 2002. 41-44.
21. Notkin, D., Garlan, D., Griswold, W.G. and Sullivan, K. Adding Implicit Invocation to Languages: Three Approaches. In Nishio, S. and Yonezawa, A. eds. *Object Technologies for Advanced Software*, Springer, Berlin, 1993, 489-510.
22. van Ommering, R., van der Linden, F., Kramer, J. and Magee, J. The Koala Component Model for Consumer Electronics Software. *Computer*, *33* (3), 2000. 78-85.
23. Ossher, H. and Tarr, P. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *Proc. of the International Conference on Software Engineering (ICSE) 2000*, ACM, New York, 2000, 734-737.
24. Rutherford, M.J., Anderson, K., Carzaniga, A., Heimbigner, D. and Wolf, A.L. Reconfiguration in the Enterprise Java Beans Component Model. In *Component Deployment: IFIP/ACM Working Conference*, Springer, Berlin, 2002, 67-81.
25. Shaw, M. and Garlan, D. *Software Architecture*. Prentice Hall, Upper Saddle River, 1996.
26. Szyperski, C., Gruntz, D. and Murer, S. *Component Software*. Pearson Education, London, 2002.
27. Yellin, F. and Lindholm, T. *The Java Virtual Machine Specification*. Addison Wesley, 1998.