MagicBeans: a Platform for Deploying Plugin Components

Robert Chatley, Susan Eisenbach, and Jeff Magee

Dept of Computing, Imperial College London,
180 Queensgate, London, SW7 2AZ, UK
{rbc,sue,jnm}@doc.ic.ac.uk

Abstract. Plugins are optional components which can be used to enable the dynamic construction of flexible and complex systems, passing as much of the configuration management effort as possible to the system rather than the user, allowing graceful upgrading of systems over time without stopping and restarting. Using plugins as a mechanism for evolving applications is appealing, but current implementations have limited functionality. In this paper we present a framework that supports the construction and evolution of applications with a plugin architecture.

1 Introduction

Almost all software will need to go through some form of evolution over the course of its lifetime to keep pace with changes in requirements and to fix bugs and problems with the software as they are discovered. Maintaining systems where components have been deployed is a challenging problem, especially if different configurations of components are deployed at different sites.

Traditionally, performing upgrades, fixes or reconfigurations of a software system has required either recompilation of the source code or at least stopping and restarting the system. High availability systems have high costs and risks associated with shutting them down for any period of time [18]. In other situations, although continuous availability may not be safety or business critical, it is simply inconvenient to interrupt the execution of a piece of software in order to perform an upgrade.

It is important to be able to cater for the evolution of systems in response to changes in requirements that were not known at the initial design time (unanticipated software evolution). There have been a number of attempts at solving these problems at the levels of evolving methods and classes [5, 2], components [13] and services [19]. In this paper we consider an approach to software evolution at the architectural level, in terms of *plugin* components.

Oreizy *et al* [18] identify three types of architectural change that are desirable at runtime: component addition, component removal and component replacement. It is possible to engineer a generalised and flexible plugin architecture which will allow all of these changes to be made at runtime.

The benefits of building software out of a number of modules have long been recognised. Encapsulating certain functionality in modules and exposing an interface evolved into component oriented software development [3]. An important difference between plugin based architectures and other component based architectures is that plugins are optional rather than required components. The system should run equally well regardless of whether or not plugin components have been added. Plugins allow the possibility of easily adding components to a working system after it has been deployed, adding extra functionality as it is required. Plugins can be used to address the following issues:

– the need to extend the functionality of a system,
– the decomposition of large systems so that only the software required in a particular situation is loaded,
– the upgrading of long-running applications without restarting,
– incorporating extensions developed by third parties.

Plugin systems have previously been developed to address each of these different situations individually, but the architectures designed have generally been quite specifically targeted and therefore limited. Either there are constraints on what can be added, or creating extensions requires a lot of work on behalf of the developer, writing architectural definitions that describe how components can be combined [17]. In this paper we describe a more generalised and flexible plugin architecture, not requiring the connections between components to be explicitly stated.

In the remainder of this paper we describe the implementation of a platform for managing plugin-based applications, which we call MagicBeans. We highlight some technical issues of particular interest, and present a case study of the system in use. Finally we discuss related work and future directions.

2 The Software

We have implemented a generalised infrastructure for our plugin architecture, which we call MagicBeans. In this section we describe the requirements and details of the implementation. We also present an example application that runs on top of the MagicBeans platform. This application is a large piece of analysis software, which has been extended in various ways through plugin components.

2.1 Requirements

To enable the evolution of software systems through the addition, removal and coordination of plugin components at runtime, we require some kind of runtime framework to be built. We have a number of functional requirements for the system.

The framework should form a platform on top of which an application can run. The platform should launch the application, and from then on manage the configuration of plugin components. It should work as automatically as possible, so that the right interfaces and classes from each component are detected and loaded, and components are matched and bound appropriately by the framework. It should be possible to plug components together in chains and other configurations. The configuration should be managed entirely by the platform.

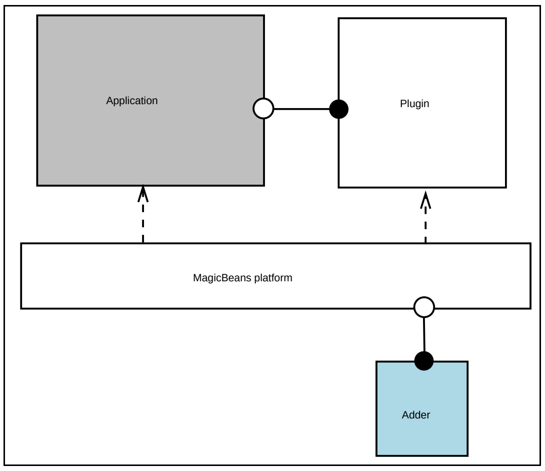


Fig. 1. Platform architecture managing a two component application

Using the plugin platform should have minimal impact on the developer or the user (or system administrator). The developer should not be forced to design their software in a particular way, to make extensive calls to an API, or to write complex descriptions of their components in any form of architecture description language. There should be no particular installation procedure that needs to be gone through in order to add a component, simply allowing the platform to become aware of the new component's location should be enough.

The mechanism by which new components are introduced to the system should not be prescribed by the platform. It should be possible to adapt easily the framework to allow components to be added in new ways, for instance: located by a user, or discovered in the filesystem or network, etc.

2.2 Implementing Plugin Addition

MagicBeans is implemented in Java, and allows a system to be composed from a set of components, each of which is comprised of a set of Java classes and other resources (such as graphics files) stored in a Jar archive.

When a new plugin is added to the system, the platform searches through the classes and interfaces present in the new component's Jar file to determine what services are provided and required by the new component, and how it can be connected to the components currently in the system.

A class signifies that it provides a service by declaring that it implements an interface. In the example below, we show an AirBrush component that might be added as an extension to a paint program. The `AirBrush` class implements the `GraphicsTool` interface. It can be added to any application that can accept a `GraphicsTool` as a plugin.

```
interface GraphicsTool {

    void draw( int x, int y, Canvas c );
}

class AirBrush implements GraphicsTool {

    void draw( int x, int y, Canvas c ) {

        //implement drawing code here
    }
}
```

A component that can accept a plugin has a slightly more complex design. Here we show a paint program that can accept and use our `GraphicsTool` plugin. For a component to use the services provided by a plugin, it must obtain a reference to an object from the plugin. This is achieved through a notification mechanism. The mechanism is based on the Observer pattern [6]. Any object can register with the platform to be notified when a new binding is made which is relevant to it.

To register as an observer, an object calls the following (static) method in the `PluginManager` class:

```
PluginManager.getInstance().addObserver( this );
```

This only registers that an object is interested in new plugins, but does not specify the plugin type. An object signifies that it can accept a plugin of a certain type by declaring a method `pluginAdded(...)` that takes a parameter of that type, in this case `GraphicsTool`

When the AirBrush plugin is added, observing objects with a `pluginAdded()` method that can take a `GraphicsTool` as their parameter are notified. This is done by calling the `pluginAdded()` method, passing a reference to the new `GraphicsTool` object, through which methods can be called. It is normal to assign this reference to a field of the object or add it to a collection within `pluginAdded()` so that the reference is maintained. In the example below, the new tool is added to a list of all the available tools.

Classes can define multiple `pluginAdded()` methods with different parameter types, so that they can accept several different plugins of different types.

```
class PaintProgram {

    List tools;
    GraphicsTool current;

    PaintProgram() {

        PluginManager.getInstance().addObserver( this );
    }
    void pluginAdded( GraphicsTool gt ) {

        tools.add( gt );
    }

    void redrawScreen() {

        for ( Iterator i = tools.iterator() ; i.hasNext() ; ) {

            drawButton( (GraphicsTool)i.next() );
        }

        ...

        if ( current != null ) {current.draw( x, y, o_canvas );}

        ...
    }
}
```

For each component, the plugin manager iterates through all of the classes contained inside the Jar file, checking each for implemented interfaces (provisions) and `pluginAdded()` methods, and finding all the pairs that are compatible. For a class to be compatible with an interface, it must be a non-abstract subtype of that interface. The matching process is performed using Java's reflection [8], custom loading [14] and dynamic linking features, which allow classes to be inspected at runtime. If a match is found, a binding between the two components is added to the system. The class in question is instantiated (if it has not been already), and the notification process is triggered.

There are various mechanisms through which plugins could be introduced to the system, and which is chosen depends on the developer and the application. Possibilities include that the user initiates the loading of plugins by selecting them from a menu, or locating them in the filesystem; that the application monitors a certain filesystem location for new plugins; or that there is some sort of network discovery mechanism that triggers events, in the manner of Sun's Jini [12]. MagicBeans does not prescribe the use of any of these. It uses a known filesystem location as a bootstrap, but components that discover new plugins can be added to the platform in the form of plugin components themselves (the platform manages its own configuration as well as that of the target application) which implement the `Adder` interface. Figure 1 shows an example of the platform running, managing an application extended with one plugin, with an Adder component plugged in to the platform itself. Each Adder is run in its own thread, so different types can operate concurrently. Whenever an Adder becomes aware of a new plugin, it informs the platform and the platform carries out the binding process. We have written example applications that load plugins from a known filesystem location, and that allow the user to load plugins manually using a standard "open file" dialog box.

2.3 Plugin Removal

As well as adding new plugins to add to the functionality of a system over time, it may be desirable to remove components (to reclaim resources when certain functionality is no longer needed) or to upgrade components by replacing them with a newer version. Together these form the three types of evolution identified in [18].

Removal is not as straightforward as addition. In order to allow for the removal of components, we need to address the issue of how to revoke the bindings that were made when a component was added. The platform could remove any bindings involving the component concerned from its representation of what is bound to what, but when the component was added and bound, the classes implementing the relevant interfaces were instantiated, and references to them passed to the components to which they were connected. These components will retain these references and may continue to call methods on the objects after the platform has removed the bindings. It is not possible to force all of the objects in the system to release their references. If the motivation for removing the plugin was to release resources then this objective will not be met.

We can have the platform inform any components concerned when a plugin is about to be removed. This is done using the same observer notification mechanism as we use when a component is added, but calling a different method `pluginRemoved()`. Any references to the plugin or resources the component provides should then be released. For example:

```
class PaintProgram {

    ...

    void pluginRemoved( GraphicsTool gt ) {

        tools.remove( gt );
    }
}
```

When all of the notifications have been performed, the bindings can be removed. However, this technique relies on the cooperation of the plugins. We cannot force references to be released, only request that components release them. Components could be programmed simply to ignore notifications (or may not even register to be notified) and in this case will continue to retain their references after a binding is removed.

As a solution to this problem, in addition to using the notifiation mechanism, when classes providing services are initially instantiated, instead of providing another component with a reference directly to that object, the reference passed is to a proxy. All the references to the objects from the plugin are then under the control of the platform, as the platform maintains a reference to each proxy. When a component is removed, the reference from each proxy to the object that provides its implementation can be nullified, or pointed at a dummy implementation. In this way we can force that resources are released. In the event that at component does try to access a plugin that has been removed, we can throw a suitable exception.

In order to provide this level of indirection, we use Java's `Proxy` class (from the standard API) to create a proxy object for each binding created. The `Proxy` class allows us to create an object dynamically that implements a given interface (or interfaces), but which, when a method is called, delegates to a given `InvocationHandler` which actually implements the method or passes the call on to another object. Using this mechanism, the implementation of the method can be switched or removed at runtime simply by reassigning object references. This gives us exactly what we need. When a plugin is removed we can nullify the reference to the delegate. When a method is called we check for the presence of a delegate, and if it has been removed throw a suitable exception back to the caller.

Another major concern is deciding when it is safe to remove a component. For instance, it will be very difficult to replace a component if the system is currently executing a method from a class belonging to that component. This problem is solved by synchronising the methods in the proxy object, so that a component cannot be removed whilst another object is in the midst of executing a method supplied by this component.

2.4 Plugin Replacement

We can perform plugin replacement in order to effect an upgrade of a system. However, before removing the old component, checks must be made to ensure that the new version is compatible.

A safe criterion for compatability of components might be that the new one must provide at least the services that the one it is replacing provides, and must not require more from the rest of the system [22]. In this way we can compare two components in isolation to decide whether one is a suitable substitute for the other.

However, in the case of plugin systems, there are a few more subtleties to be considered. With plugin systems, components that are used by other components are not strictly *required* but optional extensions that may be accepted. Therefore, in comparing components for compatability we do not need to consider the case of what the components require, only what they provide. It is only this that is critical to the success of the upgrade.

Also, as we are performing upgrades at runtime, during system operation, we have more information than we would have if we just had the components in isolation. At any point the MagicBeans platform knows the current structure of the system, and so knows which of the interfaces that a plugin provides are actually being used (those for which a binding has been created during the addition process). We can therefore say that new component is safe to replace another if it provides at least the same services as those that are provided by the old one, and are currently being used.

For example, we might have a component Brush which contains classes that implement `GraphicsTool` and `Help`. We can use this plugin with a graphics application as was shown previously, which will use the `GraphicsTool` interface. We could also use it with an application that allowed help to be browsed, or an application that combined both of these features. However, let us consider the case where we are using the Brush with our simple paint application. In this case, only the `GraphicsTool` interface will be used.

We may now write or purchase a new tool, say a SuperBrush. We want to upgrade the system to use this instead of the Brush. The SuperBrush does not provide the `Help` interface, but its implementation of `GraphicsTool` is far superior. If we use our first criterion for deciding compatability, then we will not be able to upgrade from a Brush to a SuperBrush, as SuperBrush does not provide all the services that Brush does. However, if we use the second criterion, then in the context of the simple paint application, SuperBrush provides all the services that are being used from Brush, (*i.e.* just `GraphicsTool`) and so we can perform the upgrade.

Replacement could be done by first removing the old component, and subsequently adding the new one, using the mechanisms as described above. However, due to the presence of the proxy objects which allow us to manange the references between plugins, we can swap the object that implements a service, without having to notify the client that is using it. In this way it is possible to effect a seamless upgrade.

3 Technical Innovations

3.1 BackDatedObserver

There are some cases in which the notification system described above has limitations. If, on adding a new plugin, multiple bindings are formed, it may be the case that bindings are created before the objects that will observe the creation of these bindings have been initialised and registered as observers.

For example, consider the case where we have two components, each providing one service to and accepting one service from the other. If component A is already part of the system, and component B is added, a binding may be formed connecting B’s requirement with A’s provision. Currently no observers from B have registered, and so none are informed of the new binding.

A second binding is then formed between B’s provision and A’s requirement. At this point, a class from B is instantiated. A reference to this object is passed to any observers in A. During the creation of the object from B, the constructor is run, and the object registers as an observer with the PluginManager. As the registration is too late, although the PluginManager matched two pairs of interfaces to create bindings, the situation that results is that A holds a reference to B, but not the other way around.

To solve this problem, we introduce the notion of a BackDatedObserver. This is an observer which, on registering, has the opportunity to catch up on previous events that occurred before it registered, but which are relevant to it. In the last example, having the observers register as BackDatedObservers would mean that the observer from B would be passed a reference to the object from A as soon as it registers, and it would be possible to call methods in both directions.

Implementing this variation on the traditional observer pattern requires that the participant that performs the notification keeps a history of past events, so that it can forward them to new observers when they register.

3.2 Distinguishing components

In order to be able to tell which observers need to be notified about which new bindings, it is necessary to maintain a record of which objects come from which components. That is to say, which component contains the class from which the object was created.

This could be done by calling a special factory method that would create objects and update the relevant data structures. However, such a scheme would impinge greatly on the natural style of programming. It would be necessary for the programmer to write something like:

```
A myA = (A)ObjectFactory.create( ‘‘A’’ );
```

instead of the usual

```
A myA = new A();
```

There are a number of problems with this. Firstly, it is a lot more cumbersome to write. Secondly, it removes static type safety. Thirdly, we cannot force programmers to use this mechanism, and no information will be recorded about any objects created in the normal style.

In a language that allows operator overloading (for example C++), we could implement a `new` operator that performs the appropriate record keeping, allowing object creation using the normal syntax. However, operator overloading is not available in Java.

The solution to this problem that has been adopted utilises the fact that in Java every object created holds a reference to its class, and every class in turn to its class loader. By associating a separate class loader with each plugin component, we can group objects into components on the basis of their class loaders. In fact, we made the class `Component`, which manages all of the information relevant to a particular plugin, a subclass of `java.lang.ClassLoader`, so that for any object, calling

```
getClass().getClassLoader()
```

will return a reference to its `Component`.

3.3 Multi-methods

As all of the objects that come from plugin components may be of types that are unknown to the MagicBeans platform, objects are created using reflection, and the references that are used to point to them have the static type `Object`, which is the ultimate superclass of all classes in Java.

If the PluginManager were to attempt to call one of the `pluginAdded()` methods in a component, it would pass a parameter with static type `Object` and the Java runtime would require that the method being called took a parameter of type `Object` even if the dynamic type of the parameter was something more specific.

In fact, during the compilation of the plugin component, the Java compiler will complain that there is no method `pluginAdded( Object o )`. If the developer adds this method, this is the one that will be called at runtime, regardless of the dynamic type of the parameter passed. The reason for this is that methods in Java are only dispatched dynamically on the type of the receiver, not that of the parameters [7].

This causes a problem as we wish to use `pluginAdded()` methods with different parameter types to specify the types of plugin that a component can accept.

In the implementation of MagicBeans we have overcome this problem by using reflection to dispatch certain methods dynamically on the parameter types as well as the receiver. This is often called “double-dispatch” or “multi-methods” [4].

We created a class `MultiMethod` which has a static method `dispatch()` which takes as parameters the intended receiver, the name of the method and the parameter.

```
MultiMethod.dispatch( receiver , ‘‘pluginAdded’’ , parameter );
```

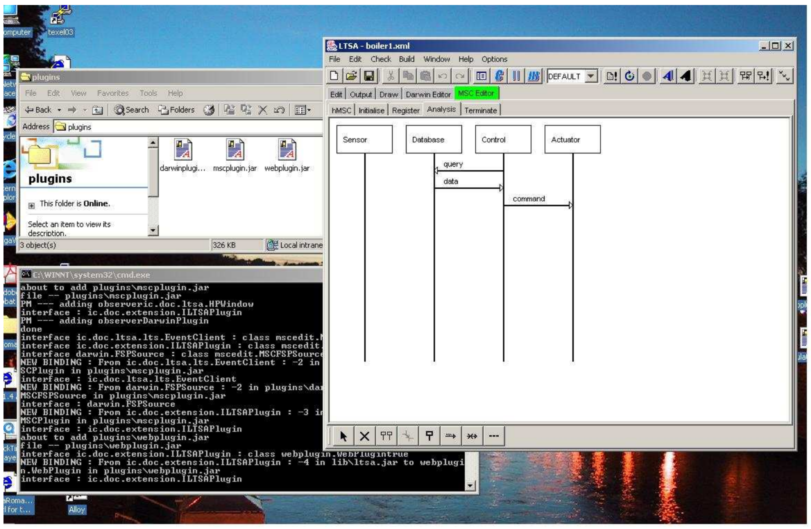
Reflection is used to search through the methods of the given receiver to find one with the given name and a formal parameter type that matches that of the parameter passed as closely as possible. This method is then invoked, again using the reflection mechanism. Double dispatch is only used when calling the `pluginAdded()` and `pluginRemoved()` methods, not for any subsequent calls between components. This means that the performance penalty incurred by calling methods in this way is kept to a minimum.

4 Case study : Extensible LTSA

The Labelled Transition System Analyser (LTSA) [10] is a Java application that allows systems to be modelled as labelled transition systems. These models can be checked for various properties, making sure that either nothing bad happens (safety) or that eventually something good happens (liveness). The core functionality of LTSA is to take textual input in the form of the FSP process calculus, to compile this into state models which can be displayed graphically and animated, and to check properties of these models.

On top of this core functionality, various extensions have been built, notably to allow more illustrative animations of the behaviour of models; to allow FSP to be synthesised from graphical Message Sequence Charts (MSCs) representing scenarios [23] so that properties of these scenarios can be analysed; to harness the sructural information given by the Darwin architecture definition language (ADL) in generating models; and to provide a facility for interacting with behaviour models by means of clicking items on web pages served over the internet to a web browser [21]. The various extensions have been implemented as plugins using MagicBeans. Figure 2 shows the LTSA tool running with three plugins connected. The console windows shows the output from MagicBeans as it loaded and bound the plugin components. The MSC and ADL plugins interact directly if both are present. Figure 3 shows the different classes and interfaces in the different components of LTSA. The grey boxes represent components which contain classes and interfaces. These are all loaded and managed by the MagicBeans platform (not pictured). The dashed arrows signify implementation of an interface, so the class MSCSpec implements the FSPSource interface. These interface implementations form the basis for the bindings between the components. For example, the ADL Editor can use an FSPSource, and an implementation of this is provided by the MSCPlugin component, so when both of these plugins are present, a binding is formed between them.

More extensions for LTSA are currently in development and the use of the plugin framework has made it very easy to for different parties to develop new functionality and integrate it with the tool.


LTSA - boiler1.xml
File Edit Check Build Window Help Options
DEFAULT
Edit Output Draw Darwin Editor MSC Editor
hMSC Initialise Register Analysis Terminate
Sensor
Database
Control
Actuator
query
data
command
plugins
File Edit View Favorites Tools Help
Back Search Folders
Address plugins
plugins
This folder is Online.
Select an item to view its description.
darwinplugi... mscplugin.jar webplugin.jar
3 object(s) 326 KB Local intrane
C:\WINNT\system32\cmd.exe

```
about to add plugins\mscplugin.jar
file -- plugins\mscplugin.jar
PM --- adding observeric.doc.ltsa.HPWindow
interface : ic.doc.extension.ILTSAPlugin
PM --- adding observerDarwinPlugin
done
interface ic.doc.ltsa.lts.EventClient : class mscedit.M
interface ic.doc.extension.ILTSAPlugin : class mscedit.
interface darwin.FSPSource : class mscedit.MSCFSPSource
NEW BINDING : From ic.doc.ltsa.lts.EventClient : -2 in
SCPlugin in plugins\mscplugin.jar
interface : ic.doc.ltsa.lts.EventClient
NEW BINDING : From darwin.FSPSource : -2 in plugins\dar
MSCFSPSource in plugins\mscplugin.jar
interface : darwin.FSPSource
NEW BINDING : From ic.doc.extension.ILTSAPlugin : -3 in
MSCPlugin in plugins\mscplugin.jar
interface : ic.doc.extension.ILTSAPlugin
about to add plugins\webplugin.jar
file -- plugins\webplugin.jar
interface ic.doc.extension.ILTSAPlugin : class webplugin.WebPlugintrue
NEW BINDING : From ic.doc.extension.ILTSAPlugin : -4 in lib\ltsa.jar to webplugi
n.WebPlugin in plugins\webplugin.jar
interface : ic.doc.extension.ILTSAPlugin
```

Fig. 2. LTSA tool running with plugins

The aim of using the plugin architecture was that rather than having one monolithic tool which combined all of the above functionality, the different extensions could be encapsulated in separate modules, and only the modules that the user required would be loaded. This selection of features should be able to be done in a dynamic way, so that no changes to the source code need to be made in order to add or remove features. The use of the MagicBeans platform provides this.

By providing a standard interface for LTSA plugins, the core of the application can use any extensions that the user requires. To use a new plugin, all that the user has to do is to drop the relevant Jar file into a certain directory. The application interrogates each plugin to find out whether it provides certain types of GUI features (menus, tool bar buttons etc) that should be added to main application's user interface. The plugins then respond to the user clicking on the buttons or menus by executing code from handler classes inside the relevant extension component.

5 Related Work

5.1 JavaBeans

JavaBeans [11] is Sun's original component technology for Java. Beans are formed by packaging classes and other resources into Jar files, but the focus is on combining and customising Beans to create an application using a graphical builder tool. They

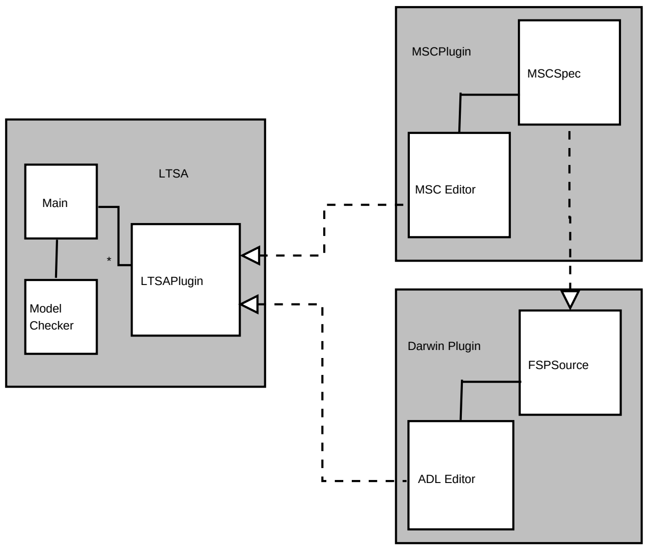


Fig. 3. Class diagram showing classes in different plugins

are commonly used for combining user interface components to create a complete GUI. The technology that we have presented here differs from this approach in that we intend the coordination of components to form applications to be as transparent as possible, and to be performed in a way that is reactive to the other components that have already been deployed in the system.

5.2 OSGi and Gravity

The Open Services Gateway initiative (OSGi) [20] Service Platform is a specification for a framework that supports the dynamic composition of services. An implementation of this specification can be integrated into applications to provide a plugin or extension mechanism. OSGi compliant applications work by managing “bundles” that are registered with a platform. Clients can query the OSGi registry for components that provide a certain service.

Gravity is an application that uses Oscar [9], an implementation of OSGi, to allow applications to be built dynamically from components that may vary in their availability. In order to use a bundle with Gravity, the component needs to contain an XML description of its provided and required services, which is not the case with MagicBeans.

5.3 Java Applets

Java applets [1] allow modules of code to be dynamically downloaded and run inside a web browser. The dynamic linking and loading of classes that is possible with Java allows extra code, extending the available functionality, to be loaded at any time.

A Java program can be made into an applet by making the main class extend `java.applet.Applet` and following a few conventions. The name of this main class and the location from where the code is to be loaded are included in the HTML of a web page. A Java enabled browser can then load and instantiate this class.

The applet concept has proved useful in the relatively constrained environment of a web browser, but it does not provide a generalised mechanism for creating extensible applications. As all applets must extend the provided Applet class, it is not possible to have an applet which has any other class as its parent (as Java has single inheritance).

5.4 Lightweight Application Development

In [15] Mayer *et al* present the plugin concept as a design pattern (in the style of [6]) and give an example implementation in Java. The architecture described in the design pattern is similar to that used by MagicBeans. It includes a plugin manager that loads classes and identifies those that implement an interface known to the main application.

Their work does allow for one application to be extended with multiple plugins, possibly with differing interfaces, but makes no mention of adding plugins to other plugins.

The plugin mechanism is described in terms of finding classes to add to the system, where we work in terms of components. Although our components do contain sets of classes (along with other resources such as graphics), it is the component as a whole that is added to provide the extension to the system.

5.5 ActiveX

ActiveX is a technology developed by Microsoft in the 1990's. ActiveX controls are reusable software components that can add specialised functionality to web sites, desktop applications, and development tools [16]. They are primarily designed for creating user-interface elements that can be added to container applications. There is no standard mechanism for establishing peer-to-peer connections between ActiveX components, only between the container and the control. This means that the configurations that can be created are a lot less flexible than what can be achieved using the MagicBeans framework.

5.6 Eclipse

The Eclipse Platform [17] is designed for building integrated development environments. It is built on a mechanism for discovering, integrating and running modules which it calls plugins.

Any plugin is free to define new extension points and to provide new APIs for other plugins to use. Plugins can extend the functionality of other plugins as well as extending the kernel. This provides flexibility to create more complex configurations.

Each plugin has to include a manifest file (XML) providing a detailed description of its interconnections to other plugins. The developer needs to know the names of the extension points present in other plugins in order to create a connection with them.

With the MagicBeans technology, the actual Java interfaces implemented by classes in plugins are interrogated using reflection, and this information is used to organise and connect components.

On start-up, the Eclipse Platform Runtime discovers the set of available plugins, reads their manifests and builds an in-memory plugin registry. Plugins cannot be added after start-up. This is a limitation as it is often desirable to add functionality to a running program without having to stop and restart it. Version 3.0 of Eclipse will address this by using an OSGi implementation to manage plugins.

6 Conclusions

We have presented a system of plugin components that allows flexible applications to be constructed by deploying sets of components that are combined to form a system. Functionality can be added to an application over time, as it is required or becomes available, by deploying a new component alongside those that are already in use by the system. Components that are no longer needed can be removed, allowing resources to be reclaimed, and components can be replaced when later versions become available.

We described MagicBeans, a platform supporting self-assembling systems of plugin components, written in Java. The platform allows applications to be constructed and reconfigured dynamically at runtime. It allows the architecture of an application to be altered by adding, removing or replacing components, without halting execution or having to restart the system.

We showed how to write a program that uses the MagicBeans framework to allow components to be be added and removed dynamically, demonstrating that the extra code that a developer needs to write to take advantage of the system is minimal. We also discussed some of the technical challenges involved in implementing the platform.

Future work in this area could address the problems of coordinating components deployed across a distributed environment, or look at the possibility of expressing some sort of structural constraints or goals for the system, so that when components are assembled, not only are interfaces matched, but a certain structure, say a ring, is maintained. This could involve inserting components between components that are already connected, which is not something that our system currently allows.

7 Acknowledgments

We would like to acknowledge our colleagues in the Distributed Software Engineering group and the SLURP group at Imperial College London for their helpful discussions. We would also like to acknowledge the support of the European Union under grant STATUS (IST-2001-32298).

References

1. Applets. Technical report, Sun Microsystems, Inc., java.sun.com/applets/, 1995-2003.
2. G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalising dynamic software updating. In *Second International Workshop on Unanticipated Software Evolution at ETAPS '03*, 2003.

3. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Pub Co, 1997.
4. C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pages 130–145, 2000.
5. M. Dmitriev. HotSwap Client Tool. Technical report, Sun Microsystems, Inc., www.experimentalstuff.com/Technologies/ HotSwapTool/index.html, 2002-2003.
6. E. Gamma, R. Helm, R. Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.
7. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2 edition, June 2000.
8. D. Green. The Reflection API. Technical report, Sun Microsystems, Inc., http://java.sun.com/docs/books/tutorial/reflect/, 1997-2001.
9. R. S. Hall. Oscar. Technical report, ungoverned.org, oscar-osgi.sourceforge.net, 2003.
10. J. Magee and J. Kramer. *Concurrency – State Models and Java Programs*. John Wiley & Sons, 1999.
11. Javabeans. The Only Component Architecture for Java Technology. Technical report, Sun Microsystems, Inc., java.sun.com/products/javabeans/, 1997.
12. JINI. DJ - Discovery and Join. Technical report, Sun Microsystems, Inc., wwws.sun.com/software/jini/specs/jini1.2html/discovery-spec.html, 1997-2001.
13. J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE TSE*, 16(11):1293–1306, November 1990.
14. S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98)*, pages 36–44, 1998.
15. J. Mayer, I. Melzer, and F. Schweiggert. Lightweight plug-in-based application development, 2002.
16. Microsoft Corporation. How to Write and Use ActiveX Controls for Windows CE 2.1. Technical report, Microsoft Developer Network, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnce21/html/activexce.asp, 1999.
17. Object Technology International, Inc. Eclipse Platform Technical Overview. Technical report, IBM, www.eclipse.org/whitepapers/eclipse-overview.pdf, July 2001.
18. P. Oriezy, N. Medvidovic, and R. Taylor. Architecture-based runtime software evolution. In *ICSE '98*, 1998.
19. M. Oriol. Luckyj: an asynchronous evolution platform for component-based applications. In *Second International Workshop on Unanticipated Software Evolution at ETAPS '03*, 2003.
20. OSGi. Open Services Gateway initiative specification. Technical report, OSGi, http://www.osgi.org, 2001.
21. R. Chatley, J. Kramer, J. Magee and S. Uchitel. Model-based Simulation of Web Applications for Usability Assessment. In *Bridging the Gaps Between Software Engineering and Human-Computer Interaction*, May 2003.
22. S. Eisenbach, C. Sadler and S. Shaikh. Evolution of Distributed Java Programs. In *IFIP/ACM Working Conference on Component Deployment*, volume 2370 of *LNCS*. Springer-Verlag, June 2002.
23. S. Uchitel, R. Chatley, J. Kramer and J. Magee. LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios. In *Proc. of TACAS 2003*. LNCS, April 2003.