Painless Plugins

Robert Chatley   Susan Eisenbach   Jeff Magee
Department of Computing
Imperial College London
180 Queen's Gate, London SW7 2AZ

E-mail: {rbc,sue,jnm}@doc.ic.ac.uk

Abstract

Using plugins as a mechanism for evolving applications is appealing, but current implementations are limited in scope. 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. In this paper we explore the design space of plugin architectures, present a framework that addresses the aforementioned issues, and demonstrate some examples of applications implemented using our plugin architecture.

1 Introduction

Maintenance is a very important part of the software development process. 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.

Traditionally, performing upgrades, fixes or reconfigurations on a software system has required either recompilation of the source code or at least stopping and restarting the system. High availability and safety critical 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.

Unanticipated software evolution tries to allow for the evolution of systems in response to changes in requirements that were not known at the initial design time. There have been a number of attempts at solving these problems at the levels of evolving methods and classes [5, 7], components [13] and services [19]. In this paper we consider an approach to software evolution at the architectural level, in terms of *plugin* components.

In [18] Oreizy *et al* identify three types of architectural change that are desirable at runtime: component addition, component removal and component replacement. We believe that it is possible to engineer a generalised and flexible plugin architecture which will allow all of these changes to be made at runtime. Here we present a framework that allows system evolution through component addition. In future we hope to go on to provide for component removal and replacement.

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 [4]. Components can be combined to create systems. 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, 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.

Plugins have previously been used to address each of these different situations individually, but the architectures designed have generally been quite specifically targeted and

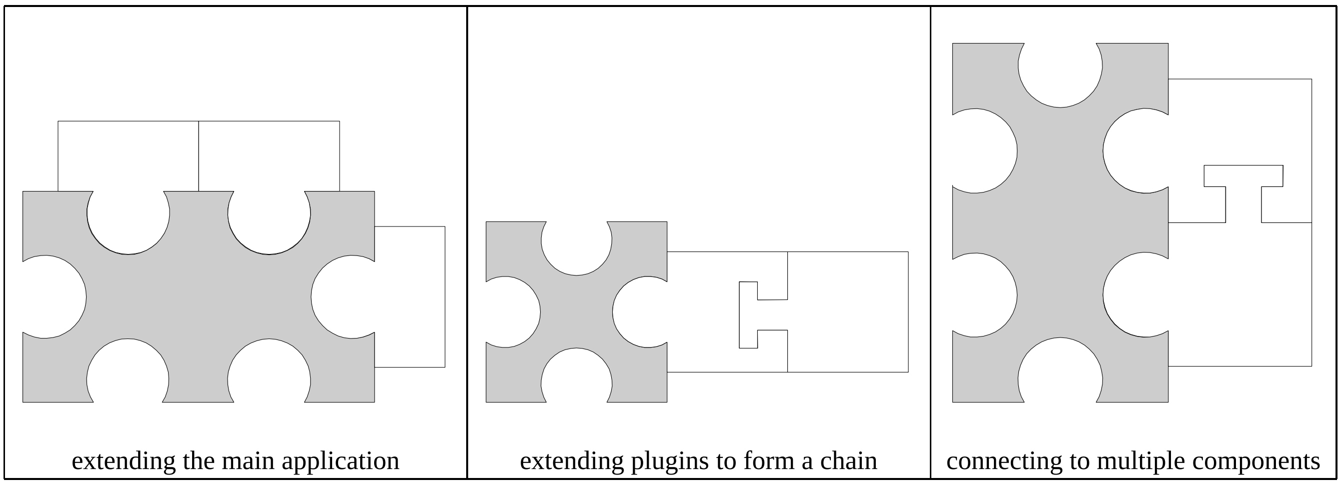
extending the main application
extending plugins to form a chain
connecting to multiple components

Figure 1. Some possible configurations of plugins

therefore limited. Here we present a generalised framework that could deal with any of them.

Extending the functionality of a system is something that is often necessary as it is not possible to know all of the requirements for the system when it is initially developed [24, 3]. For instance, consider the development of a web browser. Over time new media types will be developed and people will want to use them on the web. In order to view these new media types (for instance new video formats, or document types like Scalable Vector Graphics [23]) extra code will have to be added to the browser. It is not possible to know all of the future media types when the browser is initially developed, but it is undesirable to have to release a new version of the entire browser every time that support for a new media type is added. By providing a mechanism by which extra functionality can be plugged in, the browser could be incrementally upgraded as new features are developed. An example of this is Macromedia's plugin [15] which allows their Shockwave Flash animations to be displayed in popular web browsers.

With large systems, it is common that different users require different subsets of the total available functionality. If everyone has to have all of the functionality, this may lead to unnecessary use of memory and other hardware. If the program can be modularised and the modules combined in configurations tailored to each individual user, then resource usage can be minimised. Also, users will be exposed to an interface tailored to their needs, and the software vendor can sell different elements of functionality separately. Plugins can allow for this.

An example of such modularisation is the Eclipse Integrated Developent Environment [17]. It is possible to work with numerous programming languages in this development environment, and adding plugins gives support for the different languages required by different developers. This is achieved without all developers having to install the tools for all languages, only the subset that they require.

Upgrading long running applications is often a problem. Using traditional software models, or even component based software, it is not normally possible to change the configuration of a system (especially in terms of adding new functionality) without halting execution and restarting the application. This is particularly a problem with safety critical systems and e-commerce and other business critical servers. In a more everyday context it is just inconvenient for users to have to stop and restart their applications in order to perform upgrades. Plugins can allow for the possibility of adding code modules to reconfigure a system without having to restart.

Extensions to applications are often developed by specialist third party companies. For instance, companies specialising in computer vision technology may write extensions to major video and film processing software. The developers of the main applications are unlikely to be willing to release their proprietary source code to third party developers, yet they may want to allow their applications to be extended. Providing a plugin extension mechanism allows for this, as extensions can be purchased and added to the system separately.

Plugins have been used in existing systems, but generally in a fairly restricted way. 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]. We believe that it is possible to engineer a more generalised and flexible plugin architecture not requiring the connections between components to be explicitly stated.

In the remainder of this paper we present a way of think-

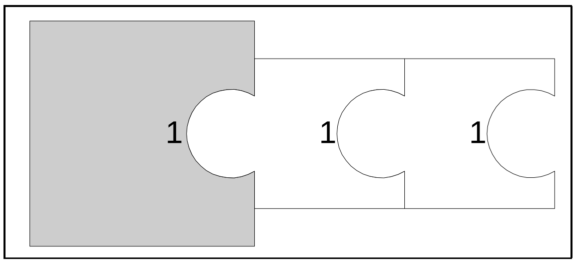

Figure 2. Chaining with cardinality constraints

ing about and modelling flexible plugin architectures based on a familiar analogy. We explore the design space using this analogy. We then describe the requirements and implementation of a framework for managing the addition of plugins to systems and present examples that use it. Finally we discuss related work and future directions.

2 An Analogy

We think of the way that components fit together in a plugin architecture as being similar to the way that pieces of a jigsaw puzzle fit together. As long as a jigsaw piece has the right shaped peg, it can connect to another piece that has a corresponding hole.

The main application provides a number of holes, into which components providing extra functionality can plug. Plugins are optional. The holes represent an interface known to the main application, and the pegs represent classes in the plugin components that implement this interface. The interface defines the signatures of methods in the class. If an application has an interface that allows other components to extend it, and a plugin contains a class that implements this interface, a connection can be made between them. The peg will fit into the hole. This situation, adding components to a central application, is shown in the first example Figure in 1.

Thinking about plugins in this way, it becomes clear that some other more sophisticated configurations would be possible if we allow plugin components to have holes as well as pegs, *i.e.* if we allow plugins to extend other plugins rather than only allowing them to extend the main application. We can then have chains of plugins as shown in the middle example in Figure 1. An example of this situation might be if the main application were a word processor, which was extended by plugging in a graphics editor, and this graphics editor was in turn extended by plugging in a new drawing tool.

It is possible that a component has several holes and pegs of different shapes (probably the most common situation in

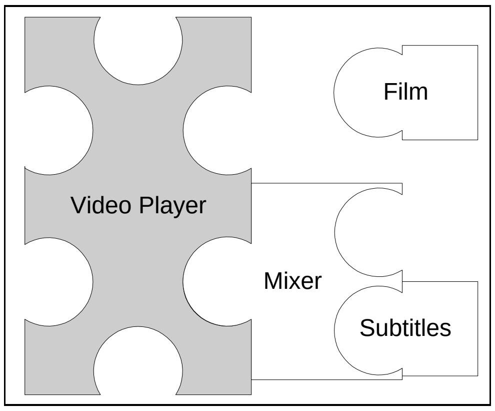

Figure 3. Non-determinism

traditional jigsaw puzzles). This can lead to more complicated configurations of components, such as those shown in the rightmost example in Figure 1. Such a configuration might be useful in a situation where the main application was, say, an integrated development environment, the first plugin was a help browser, and the second a debugging tool. The debugging tool plugs into the the main application, but also into the help browser so that it can contribute help relevant to debugging. In this way the help browser can display help provided by all of the different tools in the IDE, with the help being stored locally in each of the separate tools. It is clear that we cannot represent all possible configurations of plugins using these simple planar jigsaw representations, but they provide a useful metaphor for thinking about what might be possible.

If we think once again about the first case, then it seems that we should be able to keep on adding plugins to the application as long as they implement the right interface, but there might be cases where we want to put limits on the number of plugins that can be attached. This might be the case when each plugin that is added consumes a resource held by the main application, of which a limited quantity is available. Cardinality constraints can also be employed to constrain the shapes that the configuration can take.

To see the effect of using cardinalities, consider a main application which accepts a certain type of plugin, without a restriction on how many plugins can be added. If three compatible plugins are added, all three will be loaded and connected to the system. If, however, we change the cardinality of the interface to be ≤ 2, *i.e.* any number up to a maximum of two, after two plugins have been added, a third cannot be. It might be possible to remove plugin 1 or 2, and to replace it with plugin 3, but it is not possible to plug in all three at the same time. In practice though it seems that the two cardinalities used most often will probably be ≤ 1 and “any number”.

Revisiting the chaining patterns that we saw earlier (see the second example in Figure 1), but employing cardinalities, we can chain together a number of different components of the same type, by having each provide and accept one peg of the same shape (limiting the number of pegs accepted requires a cardinality constraint - see Figure 2). This is almost like a Decorator pattern [6] for components. A decorator conforms to the interface of the component it decorates so that it adds functionality but its presence is transparent to the component's clients. Such a situation might be useful if, for instance, we wanted to chain together video filters, each of which took a video stream as an input and provided another stream as an output. Each filter could perform a different transformation (for example converting the image to black and white, or inverting it) but the components could be combined in any order, regardless of the number in the chain. Plugins would allow this configuration to be changed dynamically over time.

It is our aim to provide the described plugin architectures in self-assembling systems [8]. It should be possible to introduce new components over time. For each additional component the system should make connections to join it to the existing system in accordance with its accepted and provided interfaces. It should not be necessary for the user or developer to provide extra information about how or where the component should be connected, as they may not have total information about the current configuration, or they may just want to delegate responsibility for managing the configuration to the system itself. The plugin framework should be able to assemble the components according to the types of the classes they contain.

Figure 3 shows a possible configuration of a video replay application. The main application displays video streams which are supplied by plugin components. The mixer component mixes two video streams into one, so can be used to add subtitles to a film. In the figure a mixer and a set of subtitles have been added to the application, and a film source is about to be added. The film source could connect either to the mixer or directly to the video player. In the first case, the subtitles will be applied to the film, in the second case the film and the subtitles will be displayed separately. We would like to be able to ensure that the behaviour desired by the provider of the film component is implemented or at very least to predict what will happen in this case. We need to know that the same thing will happen if the same components are combined on different occasions.

It is desirable that the behaviour of self-assembling systems can be made to be deterministic: it should be possible to determine what connections will be made when a certain component is added to a certain configuration. To ensure that this is the case, provision needs to be made for defining a strategy to decide between different possible bindings in a predictable way. The technique we use for this is described in more detail in the following sections.

3 Software

We have implemented a generalised infrastructure for our plugin architecture. In this section we describe the requirements and details of the implementation, and present an example application which demonstrates the details of working with the extension mechanism. We have also used this technology to implement an extensible architecture for a large piece of analysis software, and describe that in the following subsection.

We call our plugin infrastructure MagicBeans. The name comes from the fact that what ended up being developed was quite similar in concept to Sun's JavaBeans [11], but they require that the developer provides information about the bean in a manifest file.

3.1 Requirements

To enable the evolution of software systems through the addition and coordination of plugin components at runtime, we require some kind of runtime framework to be built. Examining the different cases we considered in terms of the model in the previous section, 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.

We want plugins to work as automatically as possible, so that the right interfaces and classes from each component are detected, loaded and bound by the framework without the developer having to do any extra work. The matching of components should be taken care of by the framework.

It should be possible to plug components together in chains and other configurations as seen in Figure 1. The configuration should be managed entirely by the platform.

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 definition 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 easily adapt the framework to allow components to be added in new ways, for instance:

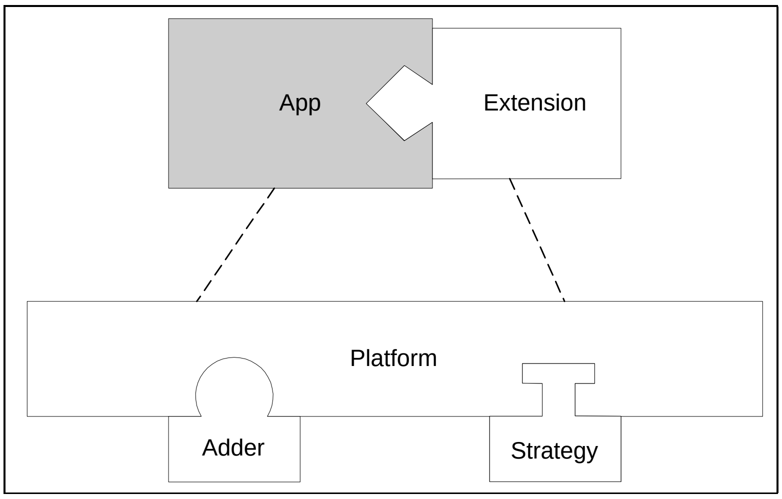

Figure 4. Platform architecture managing a two component application

located by a user, or discovered in the filesystem or network etc.

In order to successfully deal with resource management, it should be possible to specify the maximum number of plugins of a certain type that may be connected to a certain interface. The managing framework should ensure that such cardinality constraints are enforced.

In the case that there are multiple possible places where a new component could be connected to the system, it needs to be possible to define a strategy for deciding where the new component should be bound. Without this, we may have an unpredictable system where combining the same components in the same order will produce different configurations, which may well behave differently, on different occasions.

An example of using a strategy to deal with such a situation is given in a later section.

3.2 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.

The platform maintains lists of all of the components in the system and the bindings between them. We make extensive use of Java's reflection [9] mechanism and the ability to define custom classloaders [14].

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 how it can be connected to the components currently in the system.

For each component, the plugin manager iterates through all of the classes contained inside the Jar file, checking each for compatability with each of the interfaces in each of the other components currently in the system. For a class to be compatible with an interface, it must be a subtype of the interface and it must not be abstract. This matching is performed using Java's reflection, custom loading and dynamic linking features, which allow classes to be inspected at runtime. If a match is found, a binding between a class and interface (and their associated components) is added to the system. The class in question is instantiated.

Any object can register with the platform as an observer [6], so that it can be notified whenever a new binding is made to the component to which that object belongs (*i.e.* the component containing the class from which that object was created). The platform calls a method in each observer from the component that contains the interface (hole) that was matched, passing a reference to the newly instantiated object (peg). A list of instantiated pegs is maintained to ensure that in the case where a certain class implements several different interfaces, that class is instantiated only once, rather than once per interface. The code required to register an object as an observer is minimal. All that is required is the following:

```
class A implements Notifiable {

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

 void pluginAdded( Object o ) {

     //do something with the new plugin
 }
}
```

The Notifiable interface just declares the `plugi-`

nAdded() method which the platform calls to notify the observer that a new plugin has been connected and pass the object reference.

In our implementation the class Component is a subclass of ClassLoader. A Component is associated with a particular Jar archive and then used for instantiating any classes within that Jar file as necessary. Using this technique gives us the benefit that for any object in the application we can just call its `getClass().getClassloader()` method to identify which Component it is associated with, without having to keep our own records (keeping such records would be difficult anyway as unlike C++, Java has no operator overloading, and so adding code to run every time `new` is used would be difficult).

We allow cardinality constraints to be defined for certain interfaces, by allowing the developer to include a special constant, `cardinality` in any of the Java interfaces in their components. For instance:

```
public static final int cardinality = 6;
```

The plugin manager checks for the presence of such a constant when it examines the interfaces present in a component. It then keeps a count of how many components are bound to each interface in the system and ensures that the cardinality constraints are not broken. If the developer does not specify a constraint, any number of components of the correct type may be bound to an interface.

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 which 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 4 shows an example of the platform running, managing an application extended with one plugin, with one Adder and one Strategy 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 use the first two mechanisms proposed in the list above for locating new plugins.

3.3 An Example

The Virtual Fish Tank is an example application which demonstrates the use of plugins using the MagicBeans infrastructure. The basic application displays an uninhabited fish tank on the user's screen. Over time different inhabitants can be added to the tank. These inhabitants are supplied in the form of plugin components.

Initially the system starts off with only the Tank component. In order to be added to the tank, a prospective inhabitant must have a class that implements the following interface (in terms of the jigsaw analogy, Tank has a hole with a shape defined by this interface):

```
interface Inhabitant {

  final static int cardinality = 6;

  move();
  draw();
}
```

All Inhabitants can therefore be asked to move themselves, and be asked to draw themselves on the screen. The constant `cardinality` is defined to instruct the platform that the maximum number of classes implementing this interface that can be bound simultaneously to the component containing it is 6 (to prevent the tank becoming too crowded). The platform keeps a counter which is decremented every time another class implementing the interface is bound to the interface. If the counter reaches zero, no more bindings can be made to this interface.

It is possible to add a Weed to the tank. A Weed component comprises only one class (but it is still enclosed in a Jar file), which knows how to draw a weed, and when asked to move will do nothing. This class provides the peg that allows the Weed component to connect to the Tank.

A Goldfish on the other hand is a component comprising two classes:

```
class Fish {
  draw() { ... }
}

class GoldFish extends Fish
               implements Inhabitant {

  getColor() { ... }
  move() { ... }
}
```

This component is implemented according to the template method pattern [6]. Behaviour common to all types of fish is defined in the superclass, with the subclass providing the detail specific to Goldfish about its colour and how it moves. The GoldFish class provides the peg to fit in an Inhabitant hole.

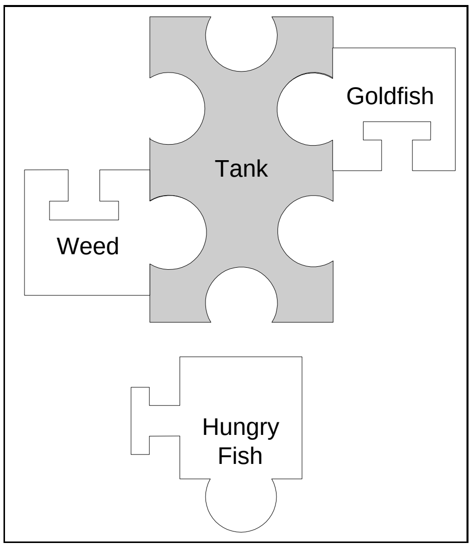

Figure 5. Adding a Predator to the fish tank

The Platinumfish is implemented in the same way (it has a different colour, and moves a bit faster than the Goldfish). In fact, the Fish class in both components is identical, however it is still necessary to include it in both components. It is not known which of the two will be added first (if they are added at all) and so each component must independently provide all of the resources it needs in order to function.

To allow a more complex configuration to be constructed, Inhabitants can be extended to include an interface that accepts a Predator. Predators can eat other Inhabitants.

```
interface Predator {

     void eat( Inhabitant food );
}
```

Figure 5 shows a situation in which a Hungryfish, which is both an Inhabitant and a Predator is about to be added to the Tank. The Hungryfish will be bound to the Tank by its Inhabitant interface. It will also be bound to one of the other Inhabitants, which it sees as food, by its Predator interface. In the situation shown, which Inhabitant the Predator will be bound to is not clear, there are two possibilities. In order to have a deterministic system, some sort of strategy needs to be defined that governs the linking behaviour. In this case a suitable strategy might be that Weeds should take precedence over Fish (we want hungry fish to eat the weeds in preference to eating other fish).

This strategy can be implemented by providing a preference function that the platform can use to compare two candidate bindings. Strategies allow the platform to perform a pairwise comparison between binding targets to determine to which existing component plugins should be bound. As long the available strategies provide preference functions

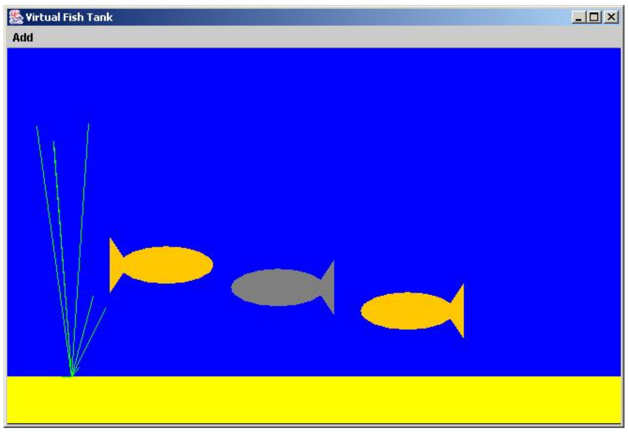

Figure 6. The Virtual Fish Tank application

for all situations where there is a possible choice of binding targets, the system is deterministic. As the very nature of plugins means that the types of components that need to be decided between are not necessarily known to the developer of the platform, we have made strategies be plugins themselves. They can be added to the platform as new types of plugin are introduced to the system. (It should only be possible to bind strategy plugins directly to the platform, as they are designed to solve the problems of non-determinism, and cannot do this if they themselves are subject to such problems - see Figure 4). Strategies implement the following interface:

```
interface Strategy {

     Binding prefer( Binding this ,
                     Binding that );
}
```

The different types of Inhabitant can be added in any combination up to six in all, since that is the cardinality constraint of the Inhabitant interface. Figure 6 shows a screenshot from the application running with one Weed, one Platinumfish and two Goldfish having been added. If a component is loaded multiple times, Java's default class loading behaviour means that the classes will be identified as already having been loaded, and the cached copies used rather than reloading.

It is possible that under certain circumstances this behaviour might be undesirable. If, for example, two different people created components for the fish tank, implementing them using different versions of a popular library class (which they package into their component) then it may be the case that a user wants to load both of these at the same time, side by side, so that each component uses the correct version of the library. An alternative loading mechanism could be written that loads separate copies of the class side by side. This cuts down on re-use where identical code is provided by two different plugins, but does avoid problems caused by conflicting versions.

In general, the situation that we would like to achieve is one in which separate copies of libraries can be loaded if they are not compatible, but that libraries should be upgraded and shared in a suitable way if they are compatible, in order to hold the minimum possible number of copies of each library in memory at any time.

It is possible that later plugins provide a later version of a class that an earlier plugin already provides. To avoid duplication in memory, it is desirable to upgrade the existing class to the new version if the two are compatible. A mechanism for upgrading already loaded classes would be required to do this. In [2] a method is suggested based on JMX, but this is not particularly convenient, as all objects must be instantiated in a particular way in order to take advantage of it. We have prototyped a solution based on the Java HotSwap technology described in [5]. In future work we hope to incorporate such a mechanism into the plugin platform.

3.4 Extensible LTSA

The Labelled Transition System Analyser (LTSA) [10] is a Java application which 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 representing scenarios [21] so that properties of these scenarios can be analysed, 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 [20]. The various extensions have been implemented as plugins using MagicBeans. More extensions for LTSA are currently in development and the use of the plugin framework has made it very easy to integrate new functionality with the tool.

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.

By providing a standard interface for LTSA plugins, the core of the application can use any extensions that the user requires (currently plugin components are placed in a specific directory and discovered when the application starts up, but the mechanism could easily be altered to allow plugins to be added at any time during execution). The application interrogates each plugin in turn 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.

If, as in the MagicBeans architecture, the holes and pegs in the jigsaw analogy are realised using interfaces and classes that implement those interfaces, it is only possible for the component with the interface (hole) to call methods in the component with the implementing class (peg), and not the other way around. With the LTSA extensions this was a problem as the extension components generally need to access functionality provided in the core application (for instance the model checker). This problem was solved by giving each plugin an interface through which they could call some of the main application's methods, and making one of the classes in the main application implement this. In this way two bindings are created each time a new plugin is loaded, allowing calls to be made both ways.

4 Related Work

4.1 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 that extends the functionality available to the user 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 (due to Java's single inheritance model).

4.2 Lightweight Application Development

In [16] 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.

4.3 PluggableComponent

PluggableComponent is a pattern which provides an infrastructure or architecture for exchanging components at runtime [22]. The architecture features a registry to manage the different types of PluggableComponent. The registry is used by a configuration tool to provide a list of available components that administrators can use to configure their applications, so configuration is human driven, where our approach allows automatic configuration without total knowledge of the system.

All PluggableComponents are derived from the PluggableComponent base class. As with applets, this is not as flexible as the solution using an interface which plugins must implement as any class which is derived from a class other than PluggableComponent cannot be used as a plugin.

An interesting feature is the provision of a mechanism for storing and transferring configured PluggableComponents based on Java serialization.

4.4 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. However, there it is not possible to place restrictions on the number of any type of plugin added in this architecture. As discussed previously this may be important where resources are limited.

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.

5 Conclusions

We have presented a system of plugin components that allows flexible configurations of plugins to be assembled adding functionality to an application over time, as it is required or becomes available. Using the familiar analogy of jigsaw puzzle pieces, we considered different possible configurations of plugin components.

We described a self-assembling system, which produces a configuration based on information in the code of the components, rather than relying on additional configuration information being supplied by the user or the developer. Using the system has little overhead in terms of effort for the developer or user.

The example software presented here implements a platform for composing systems from plugin components which manages connections between components and cardinality constraints.

We found that there are cases where the system in its basic form is non-deterministic. We described a mechanism based on preference functions making it possible to provide a strategy for dealing with the cases where there are options as to where to connect components. As long as the strategy is known, the behaviour is predictable. This allows us to implement a deterministic self-assembling system.

In future we plan to investigate woking with different versions of components, and to extend our system to cover the more difficult cases of plugin removal and replacement.

6 Acknowledgments

We would like to acknowledge the SLURP group at Imperial College London, especially Sophia Drossopoulou for the fish example, and Matthew Smith for his help in producing the diagrams that appear in this paper.

References

[1] Applets. Technical report, Sun Microsystems, Inc., java.sun.com/applets/, 1995-2003.

[2] M. Barr and S. Eisenbach. Safe Upgrading without Restarting. In *IEEE International Conference on Software Maintenance*, Sept 2003.

[3] B.Nuseibeh. Weaving together requirements and architecture. *IEEE Computer*, 34(3):115–117, March 2001.

[4] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Pub Co, 1997.

[5] M. Dmitriev. HotSwap Client Tool. Technical report, Sun Microsystems, Inc., www.experimentalstuff.com/Technologies/ HotSwap-Tool/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] P. S. G. Bierman, M. Hicks and G. Stoyle. Formalising dynamic software updating. In *Second International Workshop on Unanticipated Software Evolution at ETAPS '03*, 2003.

[8] D. Garlan, J. Kramer, and A. Wolf, editors. *Proc. of the First ACM SIFGOSFT Workshop on Self-Healing Systems*. ACM Press, November 2002.

[9] D. Green. The Reflection API. Technical report, Sun Microsystems, Inc., http://java.sun.com/docs/books/tutorial/reflect/, 1997-2001.

[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] Macromedia. Macromedia Shockwave Player. Technical report, Macromedia, Inc., www.macromedia.com/ software/shockwaveplayer/, 1995-2003.

[16] J. Mayer, I. Melzer, and F. Schweiggert. Lightweight plug-in-based application development, 2002.

[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] 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.

[21] 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.

[22] M. Völter. Pluggable Component - A Pattern for Interactive System Configuration. In *EuroPLoP '99*, 1999.

[23] W3C. Scalable Vector Graphics (SVG) 1.0 Specification. Technical report, W3C, http://www.w3.org/TR/SVG/, 2001.

[24] D. Zowghi, A. Ghose, and P. Peppas. A Framework for Reasoning about Requirement Evolution. In N. Y. Foo and R. Goebel, editors, *Proceedings of the 4th Pacific Rim International Conference on Artificial Intelligence, Cairns, Australia, 1996*, pages 157–168. Springer Verlag, 1996.