Extensible Systems: Plugin versus Component Substitution Architectures

Andrew McVeigh, Jeff Kramer and Jeff Magee

Department of Computing
Imperial College
London SW7 2BZ, United Kingdom
{amcveigh, jk, jnm}@doc.ic.ac.uk

Abstract

Many extensible systems are delivered in the form of a base application with a plugin architecture. Plugins can be added to the application to extend its functionality, allowing it to be tailored for different needs.

We introduce component substitution architectures as an alternative to this style, providing a more flexible and granular extension model. This approach is also shown to cope well with unplanned extension. We demonstrate how substitution, combined with a structural form of component inheritance called resemblance, can be used to effectively model and extend a system.

To evaluate both styles, we examine part of the architecture of the Eclipse development tool and indicate how each approach would handle the same extension requirement. Component substitution is shown to more naturally model the situation, with the further advantage that the cost of introducing the extension is closely aligned to the size of the change required to the architecture.

1 Introduction

One way to structure an extensible system is to provide a base application with predefined extension points where the application can be extended. Developers create plugins, which "plug into" the extension points, and these plugins can then be selectively added to an installation of the application to customize it. The base application acts as a platform, providing a substrate for a family of applications.

This approach is referred to as a plugin architecture and the basic concepts can be succinctly described using a simple design pattern [17]. Advanced plugin architectures, such as Eclipse [12], allow plugins to offer extension points also, which can then be further extended by other plugins. The model in [6] also supports this feature, but calls extension points "holes" and the elements that extend them "pegs".

One of the benefits of this architectural style is that plugins can be created by developers who are not affiliated with the developers of the original application. The original developers are then free to concentrate on the base platform without having to continually expand the system to cater for every requirement.

Examples of successful applications using this approach are the Eclipse Integrated Development Environment (IDE) [11] and Firefox [5]. The COM [20] add-in model of the Microsoft Office suite could also be regarded as another example of this style.

The ability to add plugins to a system primarily facilitates additive change to a system, according to the extension points defined already in the base. Extending and customizing a system can involve more than just addition, however. We use the term *extension* to refer to a unit (analogous to a plugin) that can add, replace or delete functionality. The plugin concept is a therefore a subset of the extension concept.

The component substitution model is offered as an alternative to the plugin approach. The foundation of this approach is to allow an extension to substitute any other application component with one of its own. Combined with resemblance, which allows a component to structurally inherit from others, an extension can incrementally modify any part of a system providing great flexibility to reshape the architecture to meet new requirements.

Backbone is an architecture description language (ADL) developed as part of this work to demonstrate the component substitution concepts. This features a hierarchical component model at its core along with extensibility constructs to complement this. The aim is to make extension as natural an engineering process as initial creation, and to cope with unplanned change by allowing any part of a system to effectively become an extension point. This lifts the burden off developers to factor in predictive extension points.

To evaluate the two approaches, we consider an extension to the Eclipse IDE. This is modeled with the Eclipse plugin approach, and also in Backbone allowing direct comparison of the two styles.

The rest of the paper is structured as follows. Section 2 examines the characteristics of plugin architectures, focusing

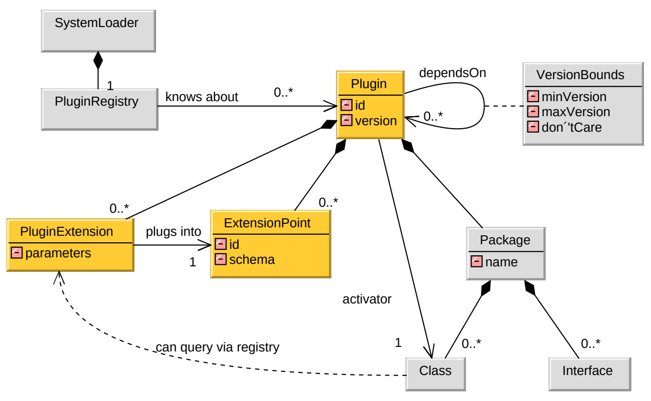

Figure 1. The plugin model

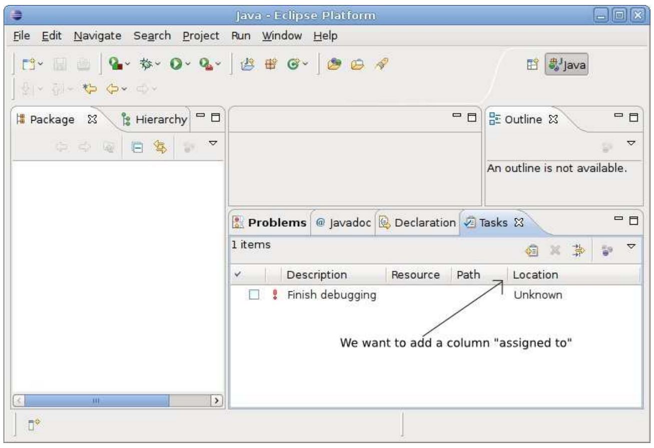

Figure 2. The Eclipse task view

specifically on the Eclipse model. Section 3 introduces the component substitution approach and the Backbone ADL. Section 4 presents the outline of a formal model for Backbone, summarizing various properties that result. Related work is reviewed in section 5 and section 6 presents conclusions and discusses further work.

2 Plugin Architectures

Figure 1 expands on the basic plugin design pattern of [17] to present a UML class diagram of a more general model. This describes the Eclipse approach where plugins can also have extension points. The distinction between plugins and the base application is therefore blurred, as the base is a relative concept which can be thought of as a system loader and a given collection of plugins. This is certainly the case in Eclipse, where even the fundamental run-time and editing concepts reside in plugins.

SystemLoader is the class which bootstraps the initial system, discovers any Plugins and registers them with the PluginRegistry. The registry knows about all Plugins, and can be used to query for a specific Plugin via the "id" and "version" attributes.

A Plugin is a versioned collection of packages, classes and interfaces. A Plugin may depend on other Plugins, for both extension points and library classes. The VersionBundle association class shows that this dependency can be expressed as a reference to a specific Plugin version, a bounded set of versions, or no particular version ("don'tCare").

A Plugin may provide a set of PluginExtensions[1] which "plug into" the ExtensionPoints of other Plugins. A PluginExtension uniquely identifies the relevant ExtensionPoint by its identifier ("id"). A Plugin may further define its own set of ExtensionPoints, each of which declares which parameters must be supplied (conforming to "schema") by a PluginExtension. The model allows an ExtensionPoint to accept multiple PluginExtensions, although each PluginExtension can only plug into one ExtensionPoint. The motivation is that if an ExtensionPoint could only accommodate a single PluginExtension, then multiple Plugins could all try to fill the point, and conflict structurally.

At run-time, the Plugins are discovered and registered, and the PluginExtensions matched up to ExtensionPoints. Control is then passed to a distinguished Plugin for bootstrapping. Each Plugin is able to query the PluginExtensions which extend its points, and the parameters passed can include class names (for object instantiation) and values.

The extensibility of the approach comes from Plugins not knowing (or needing to know) what PluginExtensions will be provided for their extension points until run-time. The actual PluginExtensions for a Plugin's extension points is a function of how many extending Plugins are discovered in the environment.

An ExtensionPoint is equivalent to a number of optionally required interfaces (to handle the multiplicity), and a PluginExtension is equivalent to a provided interface. Rather than model these concepts directly using interfaces, however, the Eclipse model expresses the data required and provided via metadata (XML files). Some of this data can refer directly to class names, and a Plugin can choose to instantiate an object based on a class name passed to its extension point.

2.1 Extending Eclipse: Adding a Column to the Task View

As a case study, we chose to enhance a small aspect of the Eclipse (version 3.3) task view. As shown in figure 2, this shows a list of tasks along with certain columns. We wish to *add* a further column "assigned to" in order to show which person has been allocated the task.

The first step in making the addition is to find the plugin

[1] In Eclipse, the actual terminology for a PluginExtension is "extension". We have used PluginExtension to differentiate this concept from our notion of an extension as being analogous to a plugin that can add, remove and replace functionality.

responsible for viewing tasks. This proved to be straight forward: a class called TaskView exists in the *org.eclipse.ui.ide* plugin. However, there is a problem. Looking at the source code for this class shows that the set of columns is hard-coded.

```
public class TaskView extends MarkerView {
    ...
    private final IField[] VISIBLE_FIELDS =
       { new FieldDone(), new FieldPriority(),
         new FieldMessage(), new FieldResource(),
         new FieldFolder(), new FieldLineNumber() };
    ...
```

The developers clearly did not anticipate this scenario. We must look for an alternative approach to using an extension point.

In an ideal world, it would be possible to simply create a new plugin, with a new EnhancedTaskView class and substitute this class for the existing TaskView class. This is the intent of the substitution facilities in component substitution architectures. However, since the plugin model primarily facilitates addition, this type of effect is not possible through the adding of a new plugin.

Examining the metadata (plugin.xml file) for this plugin shows how the task view is instantiated. The PluginExtension declaration references the TaskView class and plugs into the appropriate extension point, as shown below. (Note that the following is a declaration of a PluginExtension, rather than an extension point, which has a tag of extension-point)

```
<extension point="org.eclipse.ui.preferencePages">
  <view name="%Views.Task"
    icon="$nl$/icons/full/eview16/tasks_tsk.gif"
    category="org.eclipse.ui"
    class=
     "org.eclipse.ui.views.markers.internal.TaskView"
    id="org.eclipse.ui.views.TaskList">
  </view>
</extension>
```

If we could change this file we could substitute our EnhancedTaskView class as the parameter and our new view would be used instead of the previous one. However, the plugin.xml file lives in the org.eclipse.ui.ide plugin itself, and to replace the lines, we must create a new version of the plugin.

Creating a new version is not a perfect solution, however, due to other characteristics of the model. Firstly, the plugin consists of around 300 Java classes. The effort required in forking this plugin to create a new version is heavily out of proportion to the small architectural addition required.

Introducing a new version will also cause a problem if any plugins explicitly declare a dependency on the old version. We will end up in that case with two referenced versions of the plugin (old and new), and concurrent versions of the same plugin are not allowed in Eclipse for anything that contributes to extension points. Even if this were allowed, having two concurrent versions of a plugin holding shared state would not be a desirable outcome.

To prevent the situation where two different versions of the same plugin are required, Eclipse plugin versions follow a convention. All dependencies on plugins are expressed as a bounded version range from 3.0.0 up to (but not including) 4.0.0. The leftmost digit of the versioning scheme indicates API-breaking changes. Without this approach, we would not be able to easily introduce even a minor, non-breaking change as explicit dependencies on the old plugin version would mean having both old and new present.

It is also not possible to introduce a breaking change without also creating a new version of all plugins (incrementing the leftmost digit) which depend on this plugin, and so on in cascade fashion. This certainly constrains the type of change we can introduce, even if we are willing to update any upstream artifacts which have issues with the change. We will end up having to update most of the plugins in the system.

Even creating a new non-breaking version is likely to be a short term solution. It is known that Eclipse 3.4 is introducing a new version of this plugin, with an enhanced task view that colors tasks according to priority. As it is not possible to run two versions concurrently, we will have to accept the fact that we must merge our source changes into each new release of the plugin. Regardless of how important we view our change, it is unlikely that the maintainer (the Eclipse Foundation) will incorporate our (and everyone else's) changes into the plugin that they own and maintain. Our version will be superseded by any new ones from the maintainer, and our changes will be replaced.

2.2 Coarse-Grained Plugins

A key problem in the above scenario is the coarseness of plugins. If the plugins could somehow be made more fine-grained then the problem would be easier to solve, as we would be creating a new version of a smaller artifact.

A tension exists, however, between making the architecture fine-grained and making it manageable and understandable. If we make plugins too small, we will end up with literally many thousands. As shown by figure 3, the plugin structure for Eclipse is already complex. The figure shows around 80 plugins: a typical environment contains around 200, and an enterprise product based on Eclipse is known to contain over 500 [9]. Having more plugins implies a less manageable system. Some form of nesting or composition would address this, providing a way to view the system at multiple levels of abstraction[2].

2.3 Characteristics of the Plug-In Model

The plugin model is essentially additive, as it allows a plugin to add to existing extension points. Any extension beyond addition requires that a new version of an existing plugin be

[2]Eclipse plugins have names conforming to Java-like conventions. However, like Java packages, the name is just a convention and does not indicate any precise nesting.

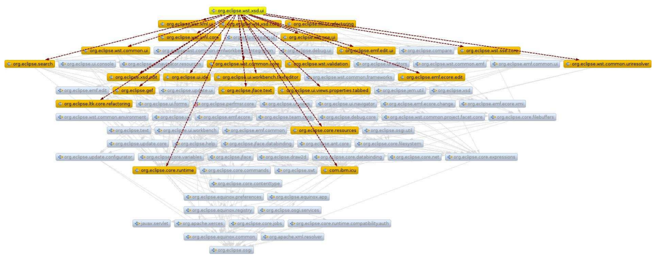

Figure 3. A partial plugin dependency graph

created. As shown, even minor changes can lead to this situation.

Our small extension required a new version of a plugin. This presented several problems. Firstly, the plugin is a sizable artifact due to the lack of composition, and creating a new version is a change that is out of proportion to the size of the architectural change (addition) required. Next, introducing a new version can lead to a situation where multiple concurrent versions of a plugin are implied, which is prohibited according to the rules of the platform. Finally, creating a new version leads to its own problems in that we are not the main developers of this plugin and will therefore have to acquiesce to performing a source level merge of our changes whenever a new version is published by the Eclipse Foundation.

As it turns out, our small change was not planned for – the creators of the plugin did not anticipate or cater for this type of change via addition. This is an interesting characteristic in that unplanned changes, even those notionally adding a feature, must be characterized as replacement. If the requirement had been foreseen, an extension point could have been provided to allow the registration of extra columns for the task view. Clearly, however, anticipating all future changes is a costly and largely unrewarding exercise. The architecture will become polluted with extension points, creating a lot of extra development work, which in turn may not in fact capture all possible future requirements.

The issues found in this example relate closely to our work on component reuse [18], where we elaborated a set of requirements for effective component reuse and extension. As Eclipse structures itself as a set of components, we can assess its approach against these requirements. In this case, the system fails to provide sufficient flexibility because we are unable to make changes without copying and modifying the source code for the plugin (ALTER, NOSOURCE). We are also unable to seamlessly accept a new version without having to perform a source level merge (UPGRADE). The requirement that changes have no impact (NO IMPACT) on existing consumers of the plugin is met, as there is no need to force the upgrade on those who do not wish to see the new version.

As it is, the Eclipse plugin model contains practical and undesirable limitations on how easily a system can be extended, understood and managed. The next section introduces component substitution architectures, which provide constructs which remove or ameliorate the limitations so far discussed.

3 Component Substitution Architectures

The foundation of the substitution approach is an extension, which is a packaged collection of interfaces and components. An extension is loosely analogous to a plugin.

Unlike a plugin, however, an extension can choose to substitute any component (in the system being extended) with one of its own. This gives an extension the ability to re-make the underlying architecture in order to effect any required changes. A further construct, called resemblance, allows a component to inherit and incrementally modify (add, delete, replace) the structure of other components. Used together, substitution and resemblance allow an extension to make an incremental modification to an existing component in a system. This is termed incremental substitution.

In the Eclipse example, a new extension could simply incrementally substitute TaskView to add the new column. Although this approach would work even when using a class-based model [10], there are limitations related to inheritance. Class-based inheritance allows method and attribute addition and a level of method replacement, which limits the type of extension possible. This was the motivation behind the introduction of a full component model, along with the resemblance construct. The relationship between components and classes will be described in this section.

An important element of the substitution approach is a hierarchical component model, where composite components are made up of instances of other components. This allows the architecture of a system to be structure hierarchically. A system can be manageably structured as a component which decomposes into other components and so on until we reach fine-grained leaf components. This gives the ability to finely decompose a system, permitting any substitution to take place at the appropriate level of abstraction.

These concepts address the major issues found in the plugin example. The model can be finely structured without becoming difficult to manage, and incremental substitution makes any constituent of an existing component into a potential extension point.

3.1 Modeling the Task View in Backbone

This section explains the Backbone ADL through the task view extension example.

The Backbone language has been developed as part of this work, in order to evaluate the substitution model. Backbone features the substitution and resemblance constructs, and a hierarchical component model where each leaf component

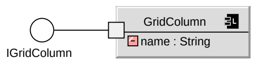

Figure 4. A column component

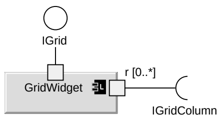

Figure 5. A grid widget component

(leaves are not further decomposable) describes a Java class. A Backbone interpreter has been created, which can assemble the system from an architectural description combined with Java classes for the leaves. More details on the motivation behind the constructs is described in [18][3].

Going back to the task view example of 2.1, consider modeling the concept of a view column (e.g. "Description") in Backbone. Figure 4 shows GridColumn as a leaf component. It has a single attribute "name" and provides the IGridColumn interface. We will later use instances of this component to configure the task view's visible columns.

As this is a leaf (note the "L" in the top right corner), it also describes a Java class. The component definition is represented graphically (or textually) in Backbone. The implementation is associated with a Java class, as shown below[4].

```
public class GridColumn {
  private IGridColumn
    g_IGridColumnProvided =
        new IGridProvidedImpl();
  private String name;

  private class IGridProvidedImpl
        implements IGridColumn {
    ...
  }}
```

The grid user interface widget (figure 5) displays a set of columns and their associated data on the screen. It is the user interface widget for displaying a task view, providing IGrid, and requiring IGridColumn. The small box named "r" indicates a port, which allows a multiplicity to be added for interfaces provided or required. The [0..*] therefore indicates that zero to many provisions are required, thus giving the same effect as an extension point which can accommodate many PluginExtensions.

Required interfaces are analogous to extension points in Eclipse. Provided interfaces are analogous to PluginExtensions, which provide data to extension points. Interfaces are more expressive however, as object references can be passed in addition to data and class references.

A composite component is constructed out of instances of other components. These instances, in the terminology of

[3]The referenced paper refers to substitution as "redefinition".

[4]The Java source code is not required for this example. It is simply shown to demonstrate the link between components and implementation classes.

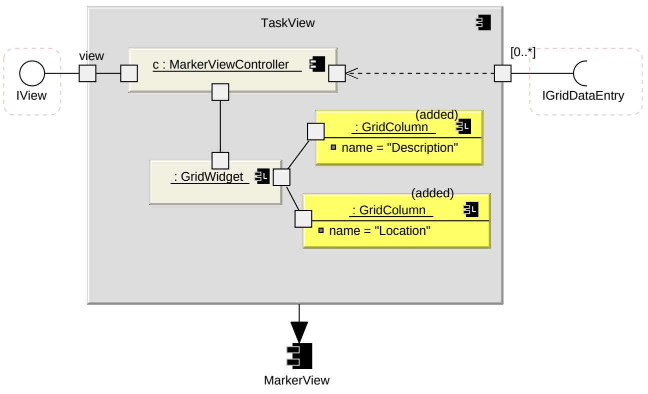

Figure 6. The task view component

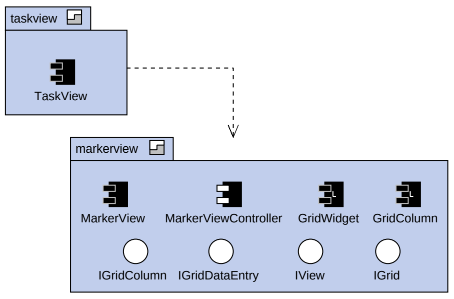

Figure 7. The taskview and markerview strata

UML2, are known as parts [22]. A composite is effectively shorthand for instructions for wiring up a set of other parts. As such, all components can be "flattened" into a connected set of leaf component instances.

Figure 6 shows the task view as a composite component. The inner boxes are instances of other components (parts). In this case, there are two columns ("Description", "Location") configured up to the GridWidget part. Note also that TaskView resembles MarkerView (fully shown). The instances of MarkerViewController and GridWidget are structurally inherited from MarkerView, which defines a generic viewer of marker information. TaskView has added the two GridColumn parts.

Note that TaskView is not a leaf, and therefore does not have a Java representation. Instances of this component must be constructed from Backbone, which will flatten the component hierarchy and connect together leaf object instances.

Finally, all the components and interfaces are bundled up into a module-like construct known as a stratum (figure 7). A stratum constitutes a unit of extension that can be applied to an application. The taskview stratum packages up TaskView, and the entire unit is dependent upon (and can legally refer to) the definitions in the markerview stratum.

Technically, a stratum is a stereotyped UML2 package. It uses different rules for package visibility, nesting and export to be more compatible with extensible systems. These rules

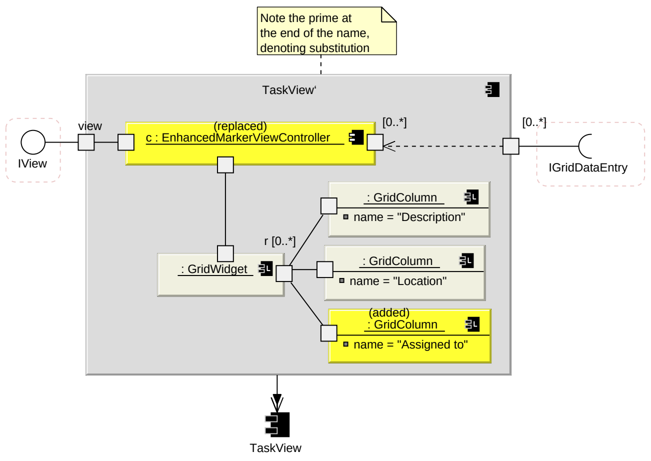

Figure 8. Incrementally substituting TaskView

aim to avoid the modularity limitations on the package construct, summarized in [24]. The strata rules are not further explained in this paper.

3.2 Extension via Substitution

The TaskView component and associated elements model the existing Eclipse task view, before our requirement to add the "assigned to" column. Our extension will need to add the extra column. To also present an example of replacement, we hypothetically suppose that the MarkerViewController instance also needs to be replaced to display the new column.

To achieve the above effect, we create a further stratum, called enhancedtaskview, which contains an incremental substitution of TaskView. The substitution, shown as TaskView‘ is shown in figure 8. The dual headed arrow between TaskView‘ and TaskView denotes incremental substitution (resemblance and substitution together).

The TaskView‘ component has replaced the MarkerViewController part with an instance of EnhancedMarkerViewController and added a GridColumn part for our new column. All other parts are inherited from the original TaskView.

The component is packaged into a stratum as shown in figure 9. We can create a system that includes this stratum, thereby applying the substitution and getting the additional column, or we can exclude it and recreate the original task view.

3.3 Resemblance as Deltas

It is vital to note that any resembling component is held as a set of deltas. For instance, the textual Backbone description of the substituting component (omitting the extra connector) is as follows:

```
{component-name: TaskView‘
 substitutes: taskview::TaskView
```

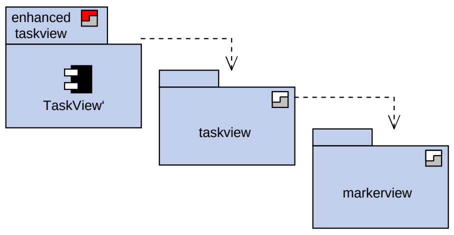

Figure 9. Packaging up the extension

```
resembles: taskview::TaskView
parts: { type: taskview::GridColumn,
slots: { attributes:
   { name: name, type: String,
     value: "Assigned to" }}}
replaceParts: {
  original: taskview::TaskView.c,
  part:{
  type:
    enhancedtaskview::EnhancedMarkerViewController}}
 ...}
```

The substituting component has replaced the part "c" with an EnhancedMarkerViewController part and added a GridColumn part. All other parts (figure 8) are inherited via resemblance from the original TaskView definition. Resemblance allows for constituent (part, port, attribute, connector) addition, replacement and deletion.

The use of deltas is important, as it allows the base component to be changed at a later point, and these changes will automatically flow through to any resembling components. A change to the base may happen through simple editing, or a further substitution when an upgrade occurs or another extension is applied.

In this approach, an evolution of a system can also be packaged as an extension. As such, Backbone merges the concept of an extension mechanism and a configuration management (CM) system. The deltas in the substitution above and the deltas from an evolution stratum can both be applied, satisfying the UPGRADE requirement outlined in [18]. This gives us the freedom to make modifications to components of which we are not the primary maintainer, which is problematic in the plugin model. The model will simply combine our changes with those of the upgrade, using strata dependencies to order the application of the deltas. In cases where only a partial strata order is present (e.g. the diamond-shaped dependencies in section 4) then multiple resemblance is used.

3.4 Substitution Model Characteristics

The component model is hierarchical, allowing components to be decomposed to a fine-grained level. This in turn allows any substitution to be more targeted than in the plugin model. Changes can be made at the appropriate level of abstraction. Large architectures can be represented and managed using hierarchy, as it is a scalable concept.

Further, any element of the model (component, part, attribute, connector etc) is a natural extension point, as it can be replaced via an incremental substitution. Unlike the plugin model which requires advance planning even for addition, the ability to extend is a seamless part of creating a system in the substitution model and extension point possibilities become more numerous as the model is elaborated.

Using a component model rather than a class model allows the internal structure of each artifact to be fully displayed graphically. As all component creation is controlled via Backbone, and is present in the graphical model as parts (or as factories for dynamic instantiation), the architecture is more explicit than a class-based model that hides object instantiation implicitly in code. Furthermore, resemblance is more powerful than inheritance, allowing deletion and replacement as well as addition. The tie between the Backbone model and the Java implementation is expressed at the leaf component level[5].

Resemblance keeps any changes, from the components being resembled, as deltas. This allows us to avoid copying errors when the base changes, and also allows us to combine substitutions from separate extensions. The graphical modeling approach always displays the expanded component (showing all constituents inherited via resemblance) even though deltas are recorded. This makes extension as straightforward as initial component creation.

An important point is that only the Backbone model and the compiled interface definitions are required to construct any extension to an application. This is equivalent to the the use of the plugin.xml metadata in the Eclipse case. Unlike the plugin model, however, in the substitution approach the implementation (Java) source is not required for replacement. Because component instantiation is completely specified within Backbone, by adjusting the configuration through the substitution construct we can replace or evolve a component.

By moving to a substitution model, more power is given to extension developers and less prediction of future extension points is required by the original application developers. However, as a model which allows deletion and replacement as well as additive change, there is the potential for structural interference between independently developed extensions. These may structurally overlap due to conflicting changes to the base application. The model is expressive enough to allow these types of conflicts to be resolved via substitution in a further stratum, an interesting property revealed by the formal model in the next section.

4 A Formal Model of Backbone

We modeled Backbone using Alloy [13], a relational logic coupled with a model finder. The model is presented in some

[5]The substitution model is applicable to other implementation languages also.

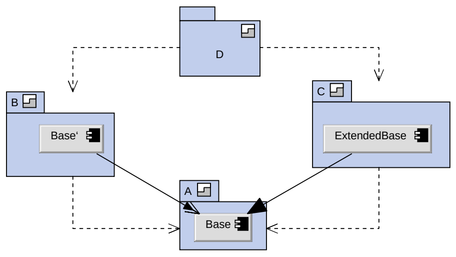

Figure 10. A diamond dependency structure

detail in [19]. This section summarizes how substitution and resemblance allow incremental changes to existing components, and how multiple substitutions are combined.

Both components and interfaces are types of elements, and can participate in resemblance and substitution relationships. (Parts of the definition not relevant to this discussion are omitted using ellipsis)

```
abstract sig Element {
  home: Stratum,
  substitutes: lone Element,
  resembles: set Element,
  resembles_e: Element -> Stratum,
  ... }
```

Each element is owned by its home stratum. It can optionally substitute (“substitutes”) for another single element of the same type, in another stratum. It can also resemble (“resembles”) one or more elements of the same type, in the home or any other stratum that the home can transitively see via dependencies.

Each element has a set of constituents, which are held as deltas. For a component, the constituents are attribute, port, part and connector.

```
sig Component extends Element {
  myParts: lone Parts/Deltas,
  myPorts: lone Ports/Deltas,
  myConnectors: lone Connectors/Deltas,
  myAttributes: lone Attributes/Deltas,
  ... }
```

A stratum owns its elements, and must explicitly express its dependencies on other strata. Via predicates, the dependency graph is guaranteed to be acyclic.

```
sig Stratum {
  parent: Stratum,
  dependsOn: set Stratum,
  ownedElements: set Element,
  ... }
```

Consider a small example system with four strata arranged in a diamond dependency structure (figure 10). The Base component lives in stratum A, and it is incrementally substituted by Base’ in stratum B. ExtendedBase in stratum C resembles base.

The resemblance graph is reordered by taking substitution into account, from the perspective of each stratum. This information is stored in the Element’s resembles_e field. For instance, from the perspective of C, ExtendedBase resembles Base. From the perspective of B, Base' also resembles Base. From the perspective of D, however, ExtendedBase resembles Base' which then resembles Base. This reflects both the fact that Base' has been substituted for Base and that Base' resembles Base.

```
all e: Element {
  all s: Stratum {
    e.resembles_e.s =
      topmostOfSubstituted + topmostOfResemblance
    ... }
```

In the above Alloy snippet, we set the resembles_e relation for each element to be the (potentially substituted) definition of the element it redefines or resembles. "Topmost" refers to the way that substitution is taken into account to take the highest substitution available. For redefinition it is the topmost substituted element in the immediately depended-upon strata (to avoid a graph cycle). For resemblance, even the current stratum's substitutions can be used.

Hence, the resembles_e values for ExtendedBase become ((Base, C), (Base', D)). This rewriting of the resemblance graph using the partial strata dependency order can result in multiple resemblance. Consider if another incremental substitution took place in stratum C (Base_C'). Then, the resembles_e values for ExtendedBase would be ((Base_C', C), (Base_C', D), (Base', D)). From the perspective of D, ExtendedBase resembles both Base_C' and Base'.

Once substitution has been factored into the resembles_e relation, the deltas are applied. In a purely linear resembles_e graph, we could just apply the deltas (add, delete, replace) in order from bottom to top. Some issues may occur even in this situation: for instance, in the system of figure 10, ExtendedBase can try to replace a part which existed from the perspective of C, but is deleted by Base' in the perspective of B and D. A small number of rules are defined to deal with such situations and they are generally in accordance with intuition.

With multiple resemblance, however, we may get legitimate conflicts. For instance, Base' and Base_C' may try to replace the same part. In this case, both parts are present from the D perspective, and both are associated with the same part identifier[6]. This flags up the Base component (the component we are substituting) and ExtendedBase as being in error from this perspective. A further incremental substitution (Base_D') in D will result in the following resembles_e graph (figure 11). As such, Base_D' is able to make any corrections to handle the conflict. Base_D' can either replace the part definitively with one of its own, or delete it entirely to leave the system error-free in D.

This type of conflict usually occurs when we try to combine independently developed extensions into the same application, as it is possible that each extension will make overlapping and incompatible changes to the base application. The

[6]Backbone uses unique identifiers (UUIDs) to preserve logical identity, rather than names. The CASE tool hides this, showing names instead.

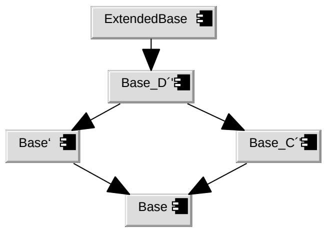

Figure 11. How Base_D' rewrites the resembles_e graph to allow conflicts correction

conflicts cannot be resolved automatically, as they represent different views of how the underlying structure should look, at the same level of abstraction. This situation is to be expected in a model which offers the flexibility to alter the base application, and it can be resolved using the same constructs that led to the conflict: resemblance and substitution.

Observe that the situation in figure 11 can be applied to the TaskView example in figure 8. TaskView acts as Base, and TaskView' acts as Base'. Base_C' represents the incremental substitution of TaskView when the maintainer upgrades TaskView to add coloring support for different task priorities (see section 2.1). Base_D' represents a further incremental substitution (Base_D') that resolves any conflicts from combining the two other substitutions.

5 Related Work

Plugin architectures have been used successfully for many systems. Eclipse is based on the OSGi module system, which provides import / export constructs and full versioning of plugins (bundles)[21]. Firefox has plugins for handling media types [5] along with more general additions [4].

COM provides a compositional component model for Microsoft Windows [2]. Applications are able to use this as an extension approach, as scripts loaded into the applications can call out to COM components. COM versions are held in a global registry, which leads to a situation called "DLL Hell", when multiple applications require different versions of the same component [27]. Even without a global registry, the problem still occurs, however, when combining multiple extensions into a single application.

Backbone is strongly influenced by architectural description languages, including Darwin [15], Koala [28], UML2's composite structure model [25] and ROOM [26].

The Backbone approach is closely related to architectural configuration management (CM) [23]. Backbone functions as a decentralized architectural CM system, preventing the need to share a common repository. Our approach also prevents the need to incorporate explicit extension points into a base architecture, unlike the plugin model and existing architectural approaches.

Product lines [7] provide a way to build up an application family from a set of related requirements. The use of gluons in [1] provides several of the same features as our approach, but is not specifically concerned with extensible applications.

Extensibility is closely related to work on language-level reuse. Mixins are effectively abstract subclasses, allowing functionality to be "mixed in" or reused by several classes [3]. However, the use of mixins must be pre-planned, as they call into methods of the superclass, and naming clashes can occur when combining multiple mixins into a single class. Aspects have been successfully used to extend systems via addition, providing a way to separate orthogonal implementation concerns [14]. Units and mixins are used in [8] to separate component creation and dependency binding. Once a binding is made, it cannot be replaced or removed, limiting the applicability for extensible systems.

UML2 contains the package merge construct, which allows packages to be added together, via an additive union [22]. The specification of this presents several problems [29], and no replacement or deletion facilities limit its utility for extensible systems. UML2 also contains the notion of redefinition. This is used in conjunction with inheritance, where the inheriting element can covariantly override the features of the base. This is closely related to resemblance, but does not allow deletion or arbitrary replacement.

6 Conclusions and Future Work

Component substitution architectures offer a more flexible alternative to plugin architectures, ameliorating or directly addressing the limitations of the plugin model. The foundation of the approach is to allow an extension to substitute any component in an application with one of its own. Combined with resemblance, which allows a component to structurally inherit from others, an extension can incrementally modify any part of the base application. Substitution and resemblance also apply to interfaces, providing a way to gracefully evolve the service capabilities of components.

A key issue with the plugin model is the lack of a composition hierarchy. Plugins cannot currently contain or nest other plugins – they all exist at the same level. Without the ability to nest, a fine-grained model of the Eclipse architecture would involve many thousands of plugins at the top level, and become extremely difficult to manage. As a consequence, a plugin in Eclipse must be relatively coarse-grained.

A further problem is that the plugin model primarily facilitates additive change, via plugging into existing extension points. If modification of existing plugins is required, because the required extension points are not present, then a new version must be created. This characteristic of turning notional additive change into replacement interacts badly with the coarse-grained nature of the plugins, leading to a disproportionate effort for simple changes. Further, distributing a new version of a plugin can be impractical particularly if others (including the primary source) are also releasing independently updated versions.

The Backbone substitution and resemblance constructs, coupled with a hierarchical component model, allow for fine-grained substitution at the appropriate level of abstraction. The constructs form a simple, but effective, decentralized configuration management system for an architecture. The keeping of changes as deltas allows multiple substitutions to be combined at the architectural level, without requiring the implementation source code.

It is not necessary to have the implementation source code to extend an application written using Backbone, unlike in the plugin model which resorts to implementation source editing when replacement is required. Since all component instantiation is performed in Backbone rather than code, any component can be substituted using the constructs purely at the architectural description level.

In Backbone every constituent of every component is a potential extension point. This leads to systems which are extensible without placing the burden on the application designer to try to predict and factor in every possible extension point. Compared to plugin architectures, component substitution has been found to result in simpler architectures, whilst simultaneously offering more power to extension developers.

The component substitution approach gives more flexibility, but introduces an obvious issue compared to plugin architectures: we can no longer guarantee that combining independently developed, but potentially structurally overlapping, extensions into a single application will not produce some conflicts. To address this, we showed that a further extension can be used to rectify issues introduced in this way.

The current status of the work is as follows. The formal model has been completed and used to implement a UML2-based case tool, where modeling with deltas is handled by always showing the expanded structure of each component. This leads to a situation where component extension is as straightforward as initial component modeling. The previous Backbone interpreter is currently being rewritten to conform to the formal model.

Future work will focus on expressing the behavioral properties of components in an extension setting. We plan to express protocols using sequence diagrams (with operators) which we will process into the FSP process algebra [16]. The individual protocols of each part of a component will be composed to detect any protocol violations that occur when substituting components. Other work includes a baselining facility where progressive deltas from layered extensions can be compressed into a new base system without deltas. This will overcome the limitation of dealing with an ever increasing set of deltas.

References

[1] Don Batory, Roberto E. Lopez-Herrejon, and Jean-Philippe Martin. Generating product-lines of product-families. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, page 81, Washington, DC, USA, 2002. IEEE Computer Society.

[2] D. Box. *Essential COM*. Addison-Wesley Professional, 1997.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[4] Mozilla Developer Center. Firefox extensions. http://developer.mozilla.org/en/docs/Extensions.

[5] Mozilla Developer Center. Firefox plugins. http://developer.mozilla.org/en/docs/Plugins.

[6] Robert Chatley, Susan Eisenbach, and Jeff Magee. Modelling a framework for plugins. In *Specification and verification of component-based systems, September 2003*, 2003.

[7] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. Software product line modeling made practical. *Commun. ACM*, 49(12):49–54, 2006.

[8] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 94–104, New York, NY, USA, 1998. ACM.

[9] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.

[10] Yuuji Ichisugi and Akira Tanaka. Difference-based modules: A class-independent module mechanism. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 62–88, London, UK, 2002. Springer-Verlag.

[11] Object Technology International Inc. Eclipse platform. *URL*, http://www.eclipse.org, July 2001.

[12] Object Technology International Inc. Eclipse platform technical overview. *Technical Report*, http://www.eclipse.org/whitepapers/eclipse-overview.pdf, July 2001.

[13] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[14] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[15] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[16] J. Magee and J. Kramer. *Concurrency (State Models & Java Programs)*. John Wiley and Sons Ltd, 1999.

[17] Johannes Mayer, Ingo Melzer, and Franz Schweiggert. Lightweight plug-in-based application development. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 87–102, London, UK, 2003. Springer-Verlag.

[18] A. McVeigh, J. Magee, and J. Kramer. Using resemblance to support component reuse and evolution. In *Specification and Verification of Component Based Systems Workshop (to be published)*, 2006.

[19] Andrew McVeigh. Alloy specification of backbone. Technical report, Imperial College, 2007.

[20] Microsoft. Com: Component object model technologies. *Website*, http://www.microsoft.com/com/default.mspx, 2006.

[21] O.Gruber, B.J.Hargrave, J.McAffer, P.Rapicault, and T.Watson. The eclipse 3.0 platform: adopting osgi technology. *IBM Syst.J.*, 44(2):289–299, 2005.

[22] OMG. Uml 2.0 specification. *Website*, http://www.omg.org/technology/documents/formal/uml.htm, 2005.

[23] R. Roshandel, A. Van Der Hoek, M. Mikic-Rakic, and N. Medvidovic. Mae—a system model and environment for managing architectural evolution. *ACM Trans. Softw. Eng. Methodol.*, 13(2):240–276, 2004.

[24] Andy Schürr and Andreas J. Winter. Formal definition and refinement of uml's module/package concept. In *ECOOP '97: Proceedings of the Workshops on Object-Oriented Technology*, pages 211–215, London, UK, 1998. Springer-Verlag.

[25] B. Selic. Tutorial d: An overview of uml 2.0, 2003.

[26] B. Selic, G. Gullekson, and P.T. Ward. Inheritance. In *Real-Time Object-Oriented Modeling*, volume First, pages 255–285. Wiley, 1994.

[27] A. Stuckenholz. Component evolution and versioning state of the art. *SIGSOFT Softw.Eng.Notes*, 30(1):7–, 2005.

[28] R. van Ommering. Mechanisms for handling diversity in a product population. In *ISAW-4: The Fourth International Software Architecture Workshop*, 2000.

[29] Dingel J Zito A, Diskin Z. Package merge in uml 2: Practice vs. theory? *Model Driven Engineering Languages and Systems*, pages 185–199, 2006.