The Implementation of the Backbone Interpreter

4th October 2006

1 Introduction

Backbone is an ADL based on the composite structure model of UML2. A Backbone configuration (program) describes a system in terms of composite and leaf components, their initial states at startup, and the initial connections between them. In this respect, it is very similar in concept to the Darwin ADL [2], although Backbone is focussed on objects in the same process space rather than distributed objects.

The purpose of the interpreter is to analyse a configuration and construct a running Java system. Each leaf component must have a corresponding Java class, and at startup the interpreter instantiates these classes and joins them in the ways specified by the connectors. After this is done, control is passed to a selected class and execution commences.

During runtime, a class can call back into the Backbone system to dynamically construct further component instances. Factory components are provided for this purpose. Deletion of instances is not currently supported.

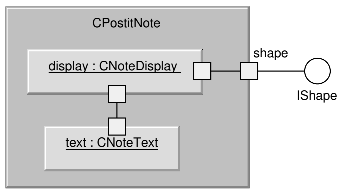
Backbone also includes an inheritance like construct called resemblance, which allows new components to be defined in terms of changes to an existing component. Redefinition can be used to alter the definition of an existing component. These are currently design-time constructs that can not be applied dynamically during program execution, but effect the definition at startup time only.

This document describes the implementation of the interpreter.

2 The Backbone ADL

In keeping with Darwin and UML2 [5], Backbone defines a component as an instantiable, class-like construct which explicitly describes the interfaces that it provides and requires. An interface represents a collection of methods defining a service and may inherit from other interfaces. Interfaces can only be provided or required via ports. Ports serve to name the role of interfaces as services offered or required by a component.

The ADL is described in more detail in [3], which also contains the description of a small drawing framework. This document uses the CPostitNote component from that paper for an example. This is a composite component with two parts. The graphical and textual form are shown in figure 1.



```
component CPostitNote
{
    ports:
        shape provides IShape;

    parts:
        CNoteDisplay display;
        CNoteText text;

    connectors:
        shape join shape to shape@display;
        text joins data@text to text@display;
}
```

Figure 1: Definition of CPostitNote

3 Interpreting a Backbone Configuration

The interpreter has three stages: parsing, creation and construction. This is shown in figure 2.

In stage (1), the parser turns the textual Backbone configuration files into a single abstract syntax tree (AST) consisting of parse nodes (P). At this point, all references from one node to another are via textual names. Parsing includes checks to ensure that referenced elements actually exist.

The creation phase (2) consists of walking the AST and generating a model consisting of elements (M). The model is similar to the AST, except that all references from one component or interface to another have been resolved from names into direct relationships. This makes navigation and analysis much easier. The model represents the definitions, initial structure and state of the system.

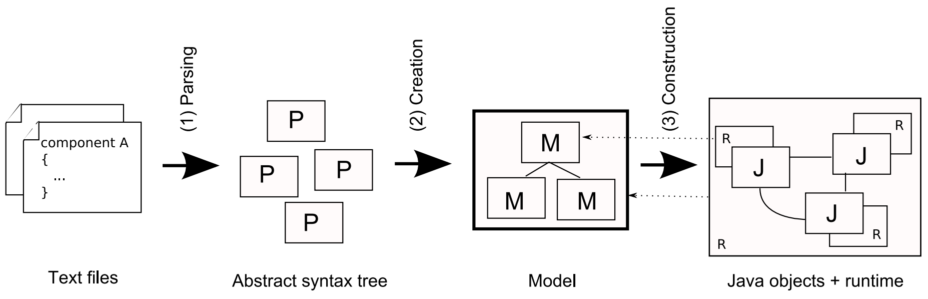


Figure 2: The three interpreter stages

The model is always static – even if more instances are dynamically added during system execution, the model always stays unchanged.

During the creation stage, any changes due to redefinitions are performed. Redefinitions are explicitly represented in the AST as parse nodes, but are not represented in the model. The model represents the system after all redefinitions have taken place. This approach will be revisited in the future as representing redefinitions directly in the model will allow for easier analysis.

The construction phase (3) uses the model to construct the Java objects representing the leaves and connects these up according to the connectors in the system. An object is "connected" to another by Java references. For instance, if leaf component A is connected to leaf B then the construction stage will set a field in the Java object representing A to a reference of the Java object representing B.

Each Java object (representing a leaf component instance) has a reference to a runtime object (R). This object allows access to the Backbone runtime system for dynamic component creation and it contains a link back to the object's Backbone component in the model. If component instances are dynamically created, the runtime objects are updated, but the model is not.

Once the system has been constructed, control is given to a nominated class, and execution can commence.

4 The Parsing Stage

Parsing takes text configuration files as input, and produces an in-memory AST as output. Error handling in the parser is basic, and the parsing stops at the first syntax error.

A Backbone configuration consists of a stratum load list specified as a .combine file. This contains the names of the strata names to load. Each stratum is defined in its own directory, and the .bb files in that directory contain the element definitions for that stratum. See figure 3 for an example of what the configuration for the note taking application [3] might look like. CPostitNode is contained in the cpostit.bb file in the postit-note stratum.

The parser is generated by JavaCC [4] from a grammar file. JavaCC also generates the classes that represent the AST.

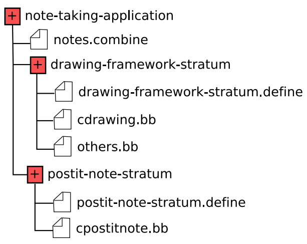


Figure 3: The directory and file structure of the example configuration

See section A.1 for implementation notes on this stage.

5 The Creation Stage

Creation takes the AST and produces an in-memory model. This is very similar to the AST, with names turned into direct references. The model can be thought of as a simplified AST which supports easier navigation and analysis.

In the AST, a component must refer to another component or interface by its name. In the creation stage, all these names are resolved and the model contains direct references to the elements. If an element name does not exist an error will be generated.

The other main function of this stage is to interpret the redefinition constructs and apply them to the model. The AST has a representation of the redefinition changes, but the model does not. Hence, creation involves setting up the model to reflect these changes. This situation is likely to change – in order to make it simpler to reason about redefinitions, it is likely that the model will also have to represent redefinition constructs.

Figure 4 shows an object diagram of the CPostitNote model.

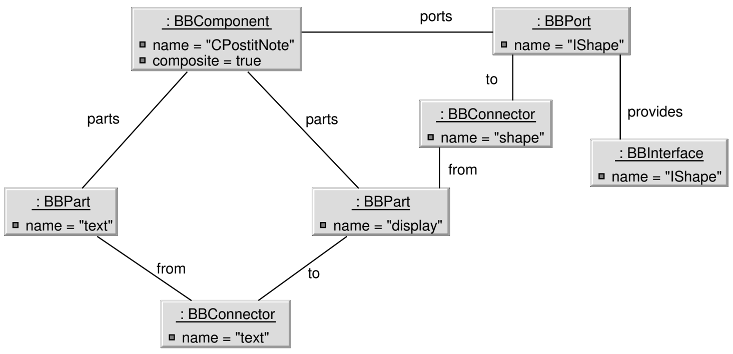


Figure 4: The model for CPostitNode

6 The Construction Stage

Construction uses the information in the model to instantiate the Java objects corresponding to the leaf component instances, and to connect these together as per the configuration. After construction, control is passed to a Java object and the program can commence execution.

As previously indicated, each leaf component must be associated with a Java class which implements its functionality. Each instantiated Java object holds a reference to a runtime object, which allows the Java object to invoke services from the Backbone runtime. An example of this is where an object calls the runtime to dynamically instantiate a new component instance.

Being a composite, CPostitNote does not have a Java representation. However, the CNoteText Java class looks as follows:

```
public class CNoteText extends BackboneRelatedComponent
{
  private BackboneProperty<String> text;
  private INoteText note = new INoteTextImpl();
  public INoteText getINoteText_of_Data()
  {
    return note;
  }

  private class INoteTextImpl implements INoteText
  {
    public string getText()
    {
      ....
    }
    public PartHandle getMyHandle()
    {
      return new PartHandle(me);
    }
  }
```

Of interest is the BackboneProperty called text. All attributes, which are settable from Backbone must be specified in this way. This allows the interpreter to set attributes to the specified state in the configuration.

This also provides a way to handle *environment* attributes, which allow the state of a part to be mapped onto the state of a parent. This provides a natural way to give the part access to its enclosing component / environment and is a crucial part of the way that Backbone facilitates component decomposition. Environment variables will be described in more detail in a future paper.

6.1 Connectors

Backbone connectors can be complex. This is primarily due to three language features: indexed ports, dynamic component instantiation and multiple connectors to a single port.

Indexed Ports

In the simple case, a connector simply joins two ports together. At the code level this involves setting an attribute in an object to the reference of another object.

However, indexed ports are allowed. An indexed port actually behaves like an array of ports. For instance, in the case of the indexed port A[0..9] we actually have A[0], A[1] and so on.

A delegate connector can connect between two indexed ports, but this can result in a complex scenario. Consider the situation in figure 5. The effect of delegateConnector between the two indexed ports can only be determined by examining the connectors on either side of the delegate connector. In this case, only indices [3] and [2] should be connected. The general case involves following the connector chain back to a direct connection between non-indexed ports. The logic to do this lives in the ConnectorChain class in the construction phase.

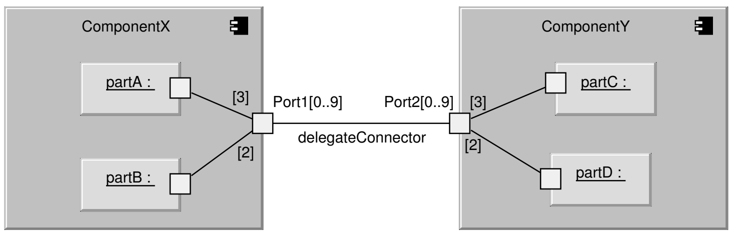


Figure 5: Delegate connectors can lead to complex connection effects

Dynamic Instantiation and Connectors

The situation becomes more complex in the presence of dynamically instantiated components. Consider that instantiation could add new delegate connectors or new basic connectors which could indirectly be joined to existing delegate connectors. Backbone handles these situations by following the connector chains back to a point where the precise connection effect is known.

Multiple Connectors to a Single Port

Two or more connectors can link to a single port. The effect that each connector has depends on the port on the other side of the connector, but this port may also have more than one connector linking to it and so on. In the future it is envisaged that this "feature" will be removed from the Backbone language to make reasoning about connectors simpler.

6.2 Factory Components

Dynamic instantiation occurs when a Java class asks the Backbone runtime to dynamically construct new component instances. The components to be instantiated are described by a factory component, which is modelled after the factory design pattern [1].

Consider figure 6 which shows ComponentA, which has a part partF which is an instance of ComponentF. ComponentF is a factory component, which means that its parts will not be created immediately, but only on request through the creation port. The factory has two attributes, “existing” and “name” which can both be set through the creation port. However, “existing” must be set to the reference of an existing component.

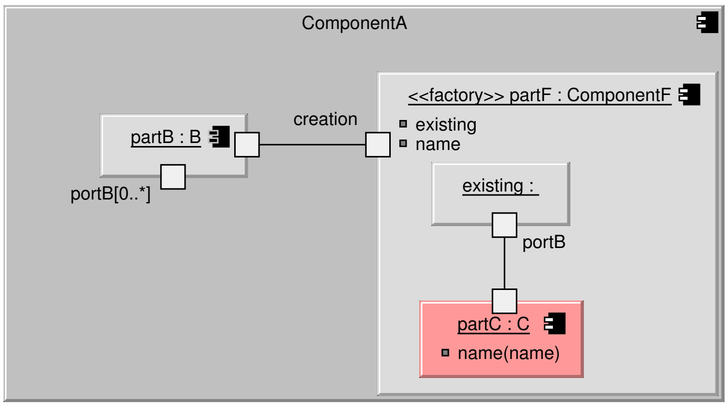


Figure 6: Use of a factory for dynamic instantiation

The effect of using the factory twice and setting “existing” to reference partB is shown in figure 7.

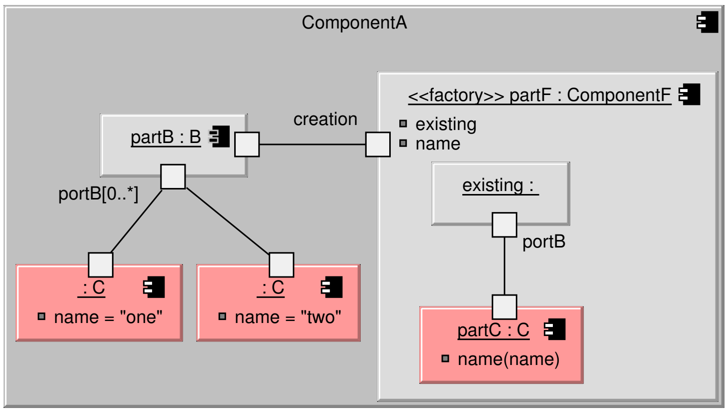


Figure 7: After using the factory twice

Factories are phrased as components in Backbone so that they can be redefined and transformed in the same way as other components.

Instance deletion is not currently supported.

7 Current Status, Limitations and Future Direction

Status

The interpreter currently supports all three stages of interpretation as discussed in this paper. It also fully supports delegate connectors and indexed ports, environment variables and factory components.

The specification of message protocols at a component level is supported, along with the translation of these specifications into a simple FSP equivalent. This will be used in future phases to check that a redefined component is still compatible with existing usage.

Limitations

Currently instance deletion is not supported. Furthermore, although redefinition is supported, the [previous] construct outlined in [3] is not yet implemented.

Redefinition is currently handled in the creation stage, and only has a representation in the AST and not the model or the runtime objects. It is envisaged that this will need to change in order to support more robust analysis of redefinitions than is currently being done.

Future Direction

A key aim of the interpreter is to fully support redefinition as the research into this area matures. Another main aim is to generate configurations directly from the jUMbLe UML2 CASE tool. This will also be upgraded to graphically depict and allow the modelling of redefined components.

Finally, connectors will be simplified and a form of type inferencing will be added so that composite components need not specify the interfaces that their ports provide and require. A construct will also be added so that it is possible to indicate the type relationships between ports in a leaf component.

8 Summary

The Backbone interpreter is a Java program that includes parsing, creation and construction stages. Its main purpose is to instantiate the Java classes that correspond to leaf components and connect them together in the way specified in the configuration. This allows a Backbone configuration to be executed.

The implementation is functional but has only been tested on example and test scenarios. The interpreter will undergo significant changes to accommodate more robust analysis of redefined components.

A Implementation Notes

This section explains some implementation details of the various stages. This section is not necessary for understanding the high-level runtime processing of the interpreter.

A.1 Generating the Parse Node Classes

The parse node classes are used by the parser to hold the AST.

The JavaCC parser generator is used to create the parser and associated node classes. The input grammar file is generated from a UML class diagram which allows the grammar to be represented visually. This is then fed into JavaCC which produces the parse node classes. See figure 8. The grammar consists of around 65 productions.

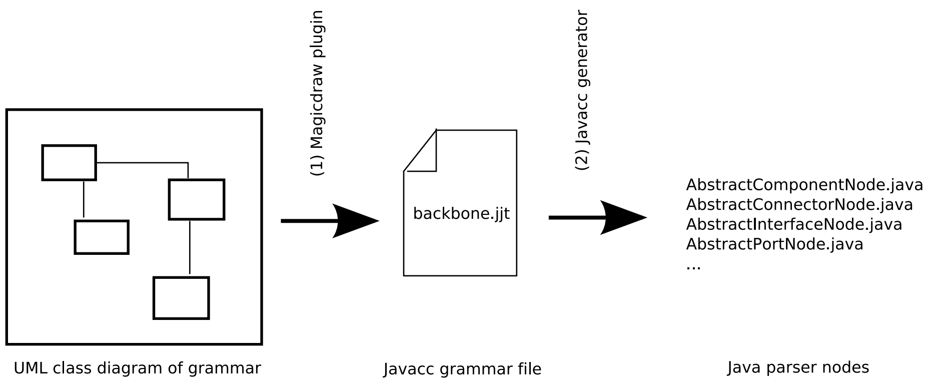


Figure 8: Generating the Parse Node Classes

A.2 Generating the Model Classes

In a similar process to above, the model classes are also generated directly from a UML diagram. The model classes represent a simplified variant of the UML2 composite structure meta-model. As such, composite structure diagrams can be used to represent a Backbone configuration, giving a convenient graphical representation.

Figure 9 shows the full set of model classes.

A.3 The Generation Gap Pattern

This pattern allows functionality to be added to generated classes such that re-generation will not overwite the added methods.

The generated parser node classes and the generated model classes use the generation gap pattern [6]. This pattern works by generating classes with names

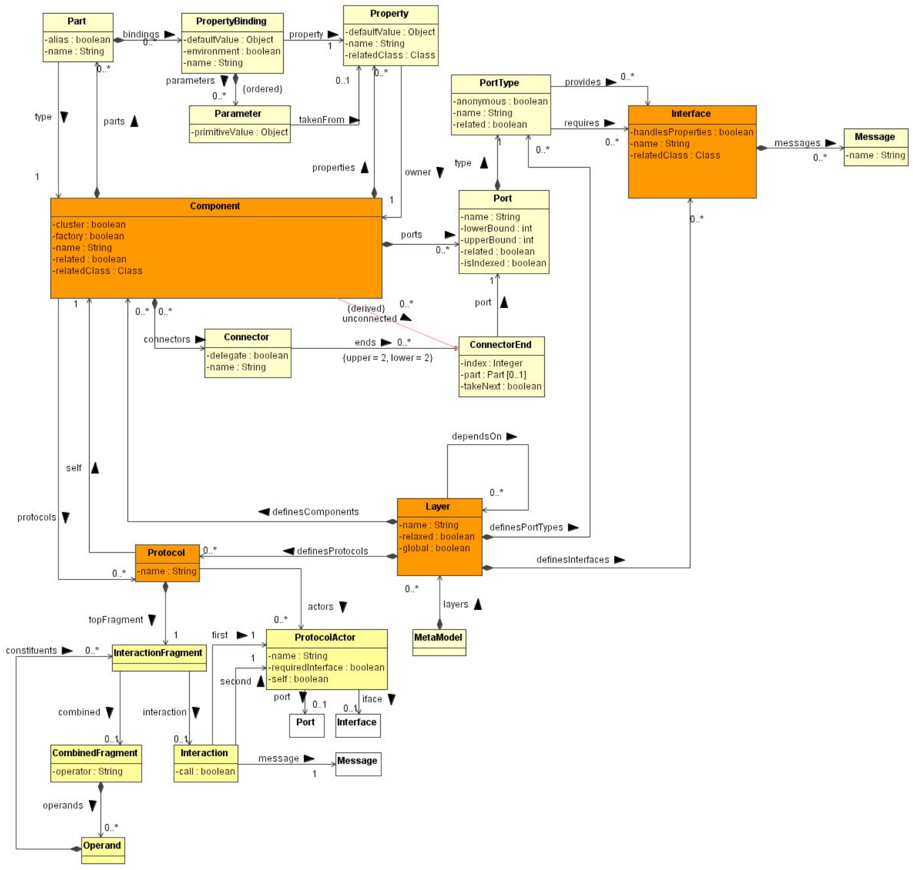


Figure 9: The Backbone model classes

prefixed by "Abstract". Another class is then created which inherits from the generated class. This class contains the functionality added by the user. This is shown in figure 10. The "Abstract" classes can be regenerated without overwriting the added methods.

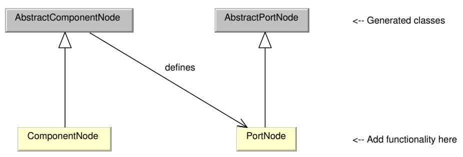


Figure 10: Example of the generation gap pattern

References

[1] E. Gamma, R. Helm, R. Johnson, and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] 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.

[3] 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.

[4] SUN Microsystems. *Website*, https://javacc.dev.java.net/, 2006.

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

[6] John. Vlissides. *Pattern Hatching: Design Patterns Applied.* Addison Wesley Professional, 1998.