adapter AR_TU_FR {
    class AppRoot_FrameworkRoot adapts AppRoot extends FrameworkRoot {
        FrameworkChild frameworkChildO { return adaptee.appChildO; }
        void primitiveOp_1O { adaptee.method_mO(); }
        void primitiveOp_2O(FrameworkChild f) { adaptee.method_n( AppChild f ); }
    }
    class AppChild_FrameworkChild adapts AppChild extends FrameworkChild {
        void primitiveOp_3O { adaptee.method_o(); }
    }
}
//example main program that applies the adapter to an application object
public class Client {
    static public void main(String[] args) {
        (AR_to_FR new AppRoot()).templateMethod1O;
    }
}

Fig. 4. Dynamic Component Gluing - Succinct Specification

adapter defines a set of dynamic connections between the application and framework components. Each connection represents the dynamic adaptation of an application class to a framework role. The adapter class of Fig. 2 can be generated from this specification.

A key theme of the work described here is separation of concerns to avoid software tangling. This is also the motivation behind both HyperSpaces [4] and Aspect-Oriented Programming [5]. HyperJ [4] and AspectJ [5] are extensions of Java that allow one to program different concerns separately. Mezini and Lieberherr proposed Adaptive Plug and Play Components, or AP&PCs, which define a slice of behavior for a set of classes, and can be parameterized to allow reuse with different class models. An enhanced form of AP&PCs that decreases tangling of connectors and aspects is described in [1]. This improved form of AP&PC uses similar techniques as we describe here, along with tool support. Example code of the adapter design pattern is available at www.cse.scu.edu/~lseiter/classes/gcse/.

References

Fig. 2 shows that no modification of the framework or application is required and that all gluing code is encapsulated within the adapter class, including code for traversing the application composite as well as code for mapping types from one domain to the other due to incompatible signatures. New adaptations can be dynamically added because adapters are implemented as separate objects.

```
public class GeneralAdapter

HashTable factoryRepository;

GeneralAdapter() {
    factoryRepository = new HashTable();
    //create a GeneralWrapperFactory for each inner class
    ....
}  
public Object wrap(String adapterClass, Object obj) {
    //get the factory for the given adapter class
    GeneralWrapperFactory factory = (GeneralWrapperFactory)factoryRepository.get(adapterClass);
    return factory.wrap(obj); //wrap the application object with an adapter (inner class instance)
}
public Object unwrap(Object obj) {
    ....
    return factory.unwrap(obj); //unwrap the application object from the adapter
}
```

```
public class GeneralWrapperFactory

HashTable table;

GeneralAdapter() { table = new HashTable(); }
public Object wrap(Object key) {
    //if the application object is not in the hashtable, create adapter for it and add to table
    ....
}  
public Object unwrap(Object obj) {
    //get the application object from the hashtable, where obj is the adapter object
    ....
}  
//create method is implemented by an anonymous class, to instantiate the appropriate
//adapter object (inner class instance)
public abstract Object create();
```

**Fig. 3.** Adapter Package - General Utility Classes

There are two main drawbacks of the technique. First, the delegation model requires additional method invocations for each invocation of a framework method. However, the source of this problem is the attempt to simulate dynamic modification of object behavior in a static language like Java. The second drawback is the complexity of the technique, especially when several frameworks are deployed within an application. Our experience with using the technique shows that it is best suited for code generators that would implement high-level language constructs for framework composition, such as *Adaptive Plug and Play Components* (APE/PCs) [2], *adapters* [3], as well as Hyper/J hyperspaces [4].

Fig. 4 defines the framework deployment using a dedicated sooping construct called *adapter* for succinctly specifying dynamic composite adaptation[3]. The
classes to simulate each dynamic class adaptation: AppRoot_FrameworkRoot and AppChild_FrameworkChild. Each inner class serves as both an application object wrapper and a framework role implementor. It is important to reuse the same adapter (inner class instance) once an application object is wrapped, as the framework may add local state. Each inner class implements a framework role interface by explicitly delegating to the wrapped application object.

```
public class AR_to_FR extends GeneralAdapter

public class AppChild_FrameworkChild extends FrameworkChild
```

**Fig. 2.** Dynamic Composite Adapter - Java Implementation

The pattern relies on the utility classes shown in Fig. 3. GeneralAdapter serves as the superclass of all toplevel adapter classes, its constructor uses Java's reflection API to generate one GeneralWrapperFactory object per inner class. The GeneralWrapperFactory class maintains a collection of adapter objects that wrap application objects in order to dynamically extend them. The wrap method of the GeneralWrapperFactory class maintains a hashtable for the mapping between an application object and the adapter that wraps it, while the unwrap method retrieves the application object wrapped by a given adapter. The wrap and unwrap methods of the GeneralAdapter class simply delegate to the appropriate GeneralWrapperFactory object.

Clients can then invoke templatemethod1() on an AR_to_FR object, passing the root of the application composite as a parameter. The root is immediately wrapped with the corresponding AppRoot_FrameworkRoot adapter. Each application object that comes into the scope of an inner class method, either through direct instantiation or as the result of calling an application method, is wrapped by the dynamic adaptation of its class. For example, in the frameworkChild() method the application object returned from the appChild() method is wrapped by a AppChild_FrameworkChild adapter. Subsequent framework method invocations are sent to the adapter. Conversely, an application object is unwrapped from its adapter before an application method can be invoked upon it, as in the frameworkChild() method body which invokes the appChild() method.
an application model in the Application package. Assume we wish to deploy the framework, with AppRoot playing the FrameworkRoot role and AppChild playing the FrameworkChild role. The dashed enclosures in Fig. 1 represent dynamic class adaptations, each depicting how an application class needs to be adapted to fulfill a framework role. Our goal is to utilize the framework without “physically” extending the application classes. Rather, we make the application objects appear to acquire the types encoded by the dynamic class adaptations of Fig. 1. That is, given the root of an application object $o$: AppRoot, we need to:

1. Wrap the object $o$ with the code in the AppRoot dynamic class adaptation.
2. Each application object that comes into the scope of the dynamic adaptation code must be wrapped with the dynamic adaptation of its class.
3. A wrapped application object should be unwrapped before it can leave the scope of the dynamic adaptation code.

This is exactly what the dynamic composite adapter design pattern does. The structure of the pattern is shown in Fig. 2. The toplevel adapter class $ARToFR$ implements the framework deployment, defining two inner adapter
Dynamic Component Gluing

Linda Seiter\textsuperscript{1}, Mira Mezini\textsuperscript{2}, and Karl Lieberherr\textsuperscript{3}

\textsuperscript{1} College of Engineering, Santa Clara University, Santa Clara, CA, USA. 
  lseiter@scu.edu, www.cs.scu.edu/ \textasciitilde seiter
\textsuperscript{2} College of Engineering and Computer Science, University of Siegen, Germany. 
  mira@informatik.uni-siegen.de, www.cs.neu.edu/home/mira
\textsuperscript{3} College of Computer Science, Northeastern University, Boston, MA, USA. 
  lieber@ccs.neu.edu, www.ccs.neu.edu/home/lieber

Abstract. We present the \textit{dynamic composite adapter} design pattern to achieve \textit{modular, dynamic, non-invasive} component adaptation. The pattern allows a clean separation between an abstract framework component and a concrete application component, while supporting the dynamic “gluing” of the two. This allows the different system concerns to be carved into separate components, which may then be dynamically composed. We also present a scoping construct for succinctly defining the dynamic gluing of Java components.

A collaboration can be implemented as a white-box framework, i.e. a set of abstract classes, a set of abstract primitive operations, and a set of concrete template methods that define the collaboration skeleton. The abstract framework model is easily customized by an application through static subclassing. However, this solution is \textit{invasive} in that it requires modification of the application classes. It is also not \textit{modular}, as the framework deployment is scattered among the application classes, its code tangled among the existing application code. It is also \textit{static}, the framework deployment is fixed prior to runtime. It is often desirable for an application to dynamically customize a framework, especially with Java’s runtime architecture. A running application may wish to apply a newly loaded collaboration scheme to a set of previously loaded classes. As component-oriented programming emphasizes the gluing of pre-existing binary components, we assume that the application model and the collaborative designs that model business processes represent concerns that are independently developed by different component vendors, and then “glued together” at the customer site. We propose the \textit{dynamic composite adapter} design pattern as a model for dynamic, non-invasive component adaptation \cite{3}.

Consider the \textit{Framework} package in Fig. 1, i.e. the \textit{FrameworkRoot} and \textit{FrameworkChild} classes. The composite structure between parent and child is modeled through the structure mapping method \textit{frameworkChild()} rather than through an aggregation relation to allow the abstract framework structure to be subsequently implemented in terms of a concrete application structure. The template methods define the collaboration skeleton, invoking abstract primitive operations that will be customized by individual applications. Fig. 1 also defines