Component Integration with Plug & Play Adapters

Mira Mezini (mira@informatik.uni-siegen.de)
Univeristy of Siegen

Linda Seiter (lseiter@scu.edu)*
Santa Clara Univeristy

Karl Lieberherr (lieber@ccs.neu.edu)†
Northeastern University

Abstract. ...

1. Introduction

Component software, i.e., software that is an assembly of individual, independently developed parts, is becoming the predominant architecture. We consider two factors as the driving force behind this development. First, as indicated in [21] component software represents a middle path between the two extremes that predominate traditional software development: (a) custom-made software, i.e., software that is developed from scratch with only the help of tools/libraries and (b) standard software, i.e., prefabricated complete solutions that can only be parameterized to get close enough to what is needed in a particular scenario [21].

The first approach has the advantage that the resulting software optimally supports the specific needs of the particular customer, i.e. important, competitive strategic decisions, as well as changes in the business process. However, it has severe cost, along with maintainance and evolution problems that make it practically obsolete. The opposite is true for the second approach. While it is certainly the more effective approach, it results in solutions that totally ignore the needs of particular customers and their strategic decisions. Component software combines the advantages of the traditional approaches while avoiding their problems: most of the individual components can be standard solutions with all the advantages that this brings, while on the other side customer-specific strategic decisison can be individually satisfied by integrating custom-made parts and adapting the standard components during the assembly process.

The second factor that favorizes component software is that most of the effort in developing an enterprise application goes actually into the development of business logic that is shared across the application's vertical domain. Feedback from companies that plan to deliver applications based on IBM's SanFrancisco framework – a Java-based collection of components that allows developers to assemble server-side business applications from existing parts

* Partially support ...
† Partially support ...

– indicates that as much as 80% of their development cost is spent writing and supporting the basic, non-competitive functions that are essentially the same for any application solution offered in a specific domain [19]. In such conditions, the benefits of concentrating the fabrication of the shared functions in a few components and reusing them across domains is obvious. This observation has actually driven the SanFrancisco project – one of the few server-based component frameworks today.

Now, if we consider component-based development to predominate server-based solutions, the architecture of server-side software resulting from this development will have more or less the layered structure in Fig. 1, which shows the architecture for Java server-side software. Actually, the figure is a slightly modified version of a figure that represents the structure of SanFrancisco-based servers-based systems found in [19]. However, the architecture is language independent, provided that server-side component models for component transaction monitors similar to Enterprise JavaBeans [20] are available for these languages.

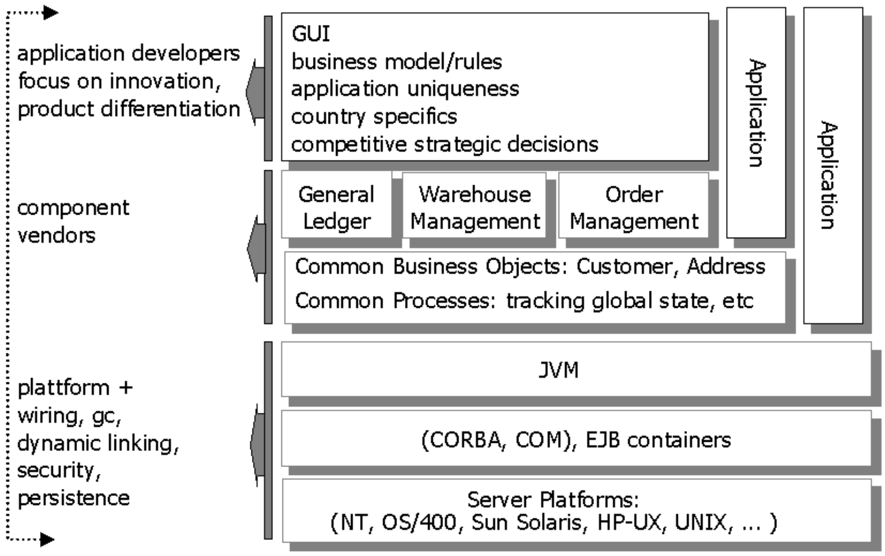

Figure 1. Server-Side Component Software

The interest of our work is in the boundary between the upper two layers. Software parts in the core business processes layer (EJB components in a Java setting) are provided by component vendors. They model partial solutions while leaving open application-specific details. Software building blocks in the top layer are developed by application developers, preferrably independently from components in the lower layer. Obviously, in an end-product the elements from these two layers need to be coded into a single solution. In order to ensure that full-advantage is taken from the component-based architecture, it is important that the integration technique satisfy the following requirements:

- *The integration process should work with independently developed components.* Pre-fabricated components from the core business processes layer may in general be the product of different vendors. Furthermore, independence between components in the core business processes layer and application-specific components in the upper-most layer should also be supported. The application developer might have legacy code (e.g., class libraries) that he/she wants to integrate into the end-product. This requirement implies that the integration process should tolerate incompatible interfaces among the components to be integrated.

- *The integration technique should enable flexible (dynamic) reconfiguration of the integrated system.* This implies easy integration of new components into an integrated setting to reflect changes in the business structure, as well as dynamic reconfiguration to enable dynamic adaptation of the system's behavior. Individual components might come in different implementations and it might be necessary to dynamically switch between these different flavors. Furthermore, it might make sense to plug certain components in and out depending on runtime conditions.

Cross domain business logic will most likely be modeled as object-oriented frameworks: business processes in the SanFrancisco project are indeed modeled as object-oriented application frameworks. Thus, the end-product of component-based application development will typically include several frameworks, as well as class libraries containing either legacy code or application-specific functionality, e.g., encoding the business structure of the particular application. However, traditional framework-based development is based on *reuse by extension* [11]. Frameworks are generally developed to be deployed by newly written application specific code. For instance, the techniques used for building SanFrancisco framework-based applications [19] are based on a mixture of subclassing and aggregation (class- and object-based composition).

We argue that traditional framework-based development does not properly support the requirements that we posed above for component integration techniques. Similar observations about problems with framework composition are made in [11]. We argue that given the importance of the integration process, object-oriented languages should provide explicit constructs for capturing abstractions in this process. We propose to extend object-oriented languages with an explict scoping construct for component integration, called a *Plug & Play Adapter*. Having a dedicated construct for capturing the component integration apart from the code that models business logic improves the modularity of the end-product. Integration of new components into the system is equivalent to defining a new adapter: the implementation of the existing parts is not affected by the integration. Furthermore, adapters are first-class objects in this model thus supporting the dynamic reconfiguration of the system. By being themselves components, adapters can be aggregated into more complex building blocks. Furthermore, refinement techniques á la inheritance apply. We have developed a prototype implementation of the *Plug & Play Adapter* in Java.

The remainder of the paper is organized as follows. In Sec. 2, we present a running example to demonstrate the shortcommings of traditional component composition techniques. The *plug& play adapter* model is presented in Sec. 3. Sec. 4 shortly presents the current prototype realization of the model in Java and outlines possible alternative realisations. Related work is discussed in Sec. 5 Finally, Sec. 5 summarizes the paper and outlines areas of future work.

2. Traditional Object-Oriented Composition Techniques

In this section we discuss problems with traditional component integration techniques in terms of a running example. A more general discussion of problems with framework composition can be found in [11].

Assume we are given a component that encodes the process of price calculation in the domain of order processing[1]. The component is part of a component framework for order processing systems. It is intended to be customized by different applications to customer-specific pricing schemes.

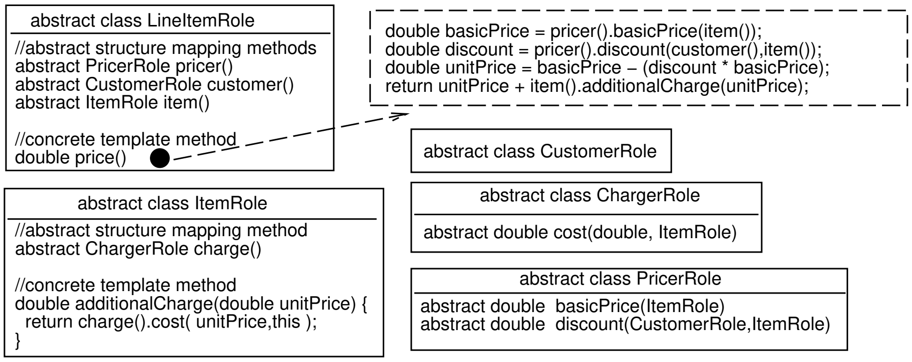

Figure 2. Pricing Framework Class Model

Fig. 2 contains the pricing framework class model. LineItemRole is responsible for calculating the actual price of a line item purchased by a customer. *PricerRole* provides price and discount information for a given item and customer. ItemRole is responsible for calculating additional charges, defined in ChargerRole. Fig. 2 also defines the collaboration required to compute the price of a line item. The *price* and additionalCharge methods are template methods that define the message and data flow of the collaboration. The primitive operations (basicPrice, discount, etc.) are abstract, to be filled in with an application-specific implementation.

The class diagram modeling the business model of an example application, a product package, is shown in Fig. 3. Assume we wish to deploy the pricing

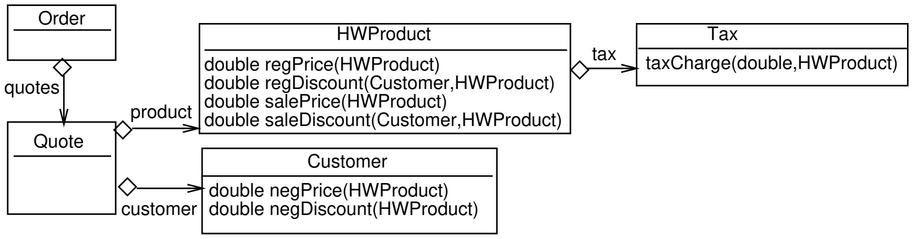

Figure 3. Product Application Class Model

component with the product application according to three pricing schemes. Each scheme requires the application model to conform to the framework model in a different way.

- *Regular Pricing*: Each product has a base price that may be discounted based on quantity ordered. Quote plays the LineItemRole, while HWProduct plays the ItemRole. In this pricing scheme HWProduct will also play PricerRole, implementing basicPrice, discount for regular pricing by calling regPrice and regDiscount, respectively. Tax plays the ChargerRole, implementing the cost method. Finally, Customer maintains the customer role.

- *Negotiated Pricing*: A customer may have negotiated certain item prices and discounts. Quote plays the LineItemRole. Customer plays the PricerRole, implementing basicPrice, discount for negotiated pricing. *HWProduct* plays the ItemRole. Finally, Tax plays the ChargerRole.

- *Sale Pricing*: Each product has a designated sale price and no discounting is allowed. Quote, HWProduct, Tax and Customer play the same roles as they do with the regular pricing scheme. However, HWProduct will implement *basicPrice* and discount for sales pricing rather than regular pricing, i.e., by calling salePrice and saleDiscount respectively.

The traditional framework deployment technique uses static inheritance to model *plays-the-role-of* mappings. For example, the regular pricing scheme would require Quote to be redefined as a subclass of LineItemRole, HWProduct to be redefined as a subclass of both ItemRole and PricerRole, etc. Note that each of the three pricing schemes requires multiple inheritance. In languages that do not support multiple inheritance, e.g., Java, some of the mappings would be established indirectly using e.g., the adapter design pattern [5].

Framework deployment using static inheritance has three drawbacks. First, it is invasive in that it requires modification of the application classes to encode the customization and the inheritance relationships. Second, it does not encapsulate the multiple roles of the pricing scheme into a single construct, as the roles would be spread out among the various application classes. Third, it is static in that it restricts the product application to a particular pricing implementation at the point of Quote class instantiation.

Accommodating all three pricing schemes and allowing an application to dynamically switch between them by applying design patterns [5] results in the design given in Fig. 4. Classes that were present in the original application model are represented in Fig. 4 by the non-shaded rectangles. The adapter pattern [5] is used in two places to adapt the interfaces of HWProduct and Customer to their PricerRole (the adapter classes HWProduct_PricerRole and Customer_PricerRole, respectively). The strategy pattern is used to model the three pricing variants (regular, negotiated, and sale) of the LineItemRole played by Quote. Each of these classes is parameterized with a different PricerRole adapter (HWProduct_RegPricerRole, HWProduct_NegPricerRole, and Customer_PricerRole, respectively). Finally, there will be *facade* classes for the price calculation according to each scheme. Each of these classes would implement the method price(Quote q) by setting the appropriate strategy and adapter objects for the quote argument, and then invoke price() on it. The factories are not shown in Fig. 4 due to lack of space.

The figure indicates several shortcomings of traditional object-oriented composition techniques. First, deployment of the pricing functionality with the product application requires the modification of the application classes. That is, it is not possible to apply newly acquired business processes to existing persistent objects. Second, the integration of the pricing component results in a proliferation of classes and spurious relations in which the original design gets lost. The clarity of the design is damaged because the code responsible for one customization is not localized in one place, but rather spread around several new adapter, strategy, or facade classes as well as in the original classes. The resulting tangled code is difficult to understand, maintain and extend with new components. This is especially true in a real-life scenario where many business objects and processes can be expected to be involved. To summarize, one could say that the design in Fig. 4 suffers from poor modularity.

As far as the requirement is concerned that the integration of new components should be possible without modifying an existing application, in such a way that the new integrated functionality can be used with existing objects, this can actually be realized by advanced design and implementation techniques that combine several simple design patterns. We have presented such a design technique and an elegant implementation of it in [14]. The technique, called the *composite adapter design pattern*, is a generalization of the original adapter pattern [5] that supports the adaptation of collaborative functionality. The original adapter pattern, whose structure is given in Fig. 5, solves the problem of adapting the interface of a single pre-defined class (Adaptee in Fig. 5) to match the interface of a single class in a framework (Target in Fig. 5).

However, the pattern does not deal with collaborations (composite object adaptation) and the typing issues that arise when one needs to map a set of target classes to a set of adaptee classes. Note that the sample interface used in the structure of the pattern has no return values or parameters. What if the signature of specificRequest was SpecificReturnType specificRequest(SpecificArgType$_1$, ... SpecificArgType$_n$), where SpecificReturn-

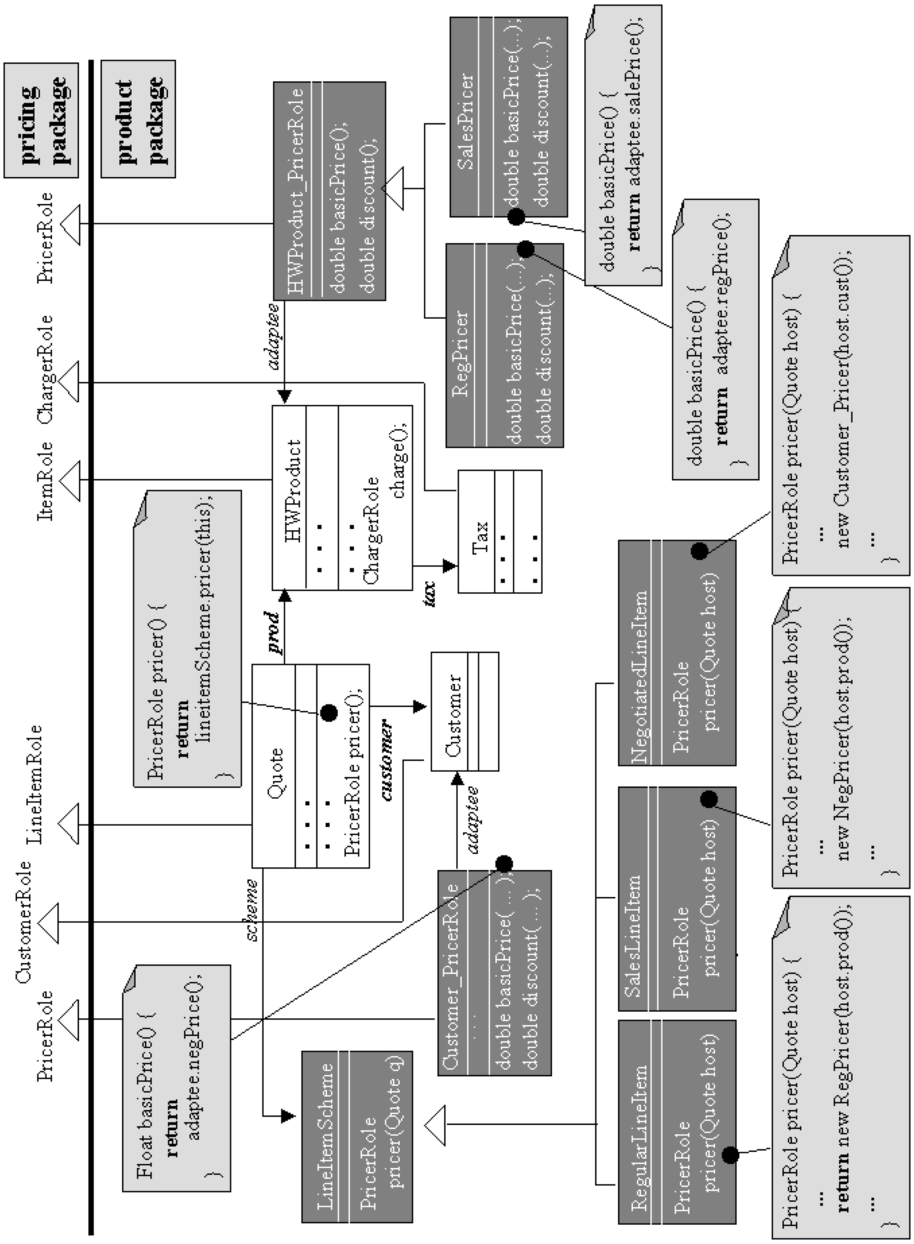

Figure 4. Traditional Framework Deployment

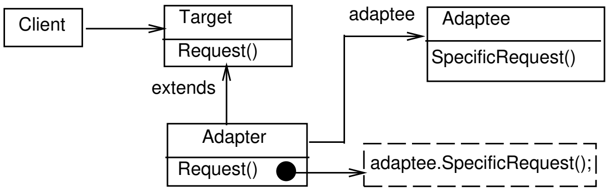

Figure 5. The Structure of the Original GOF Adapter Design Pattern

Type and $\mathsf{SpecificArgType}_1$, ..., $\mathsf{SpecificArgType}_n$ are types in the domain of the class model that defines Adaptee? These issues are not considered by the original adapter pattern, however they are solved by the composite adapter technique described in [14]. The composite adapter technique provides a means for integrating collaborative functionality and business processes into existing applications.

Fig. 6, shows the technique applied to the integration of sample Java framework and application components. We summarize the technique as follows. Assume we want to integrate the functionality implemented in the Framework package in Fig. 6 into the application class model given in the Application package. There will be a composite adapter responsible for this integration, the class (App_Frm_CompositeAdapter in Fig. 6, which will have a nested class adapter defined for each framework class (RootAdapter, ChildAdapter in Fig. 6). Each nested adapter class adapts the interface of an application class (referenced by *adaptee*) to the interface of the framework class that it extends.

The nested adapter classes RootAdapter and ChildAdapter are essentially adapters in the sense of the original adapter pattern [5]. However, the nested adapter classes must also take care that application objects (AppRoot, AppChild) that come into the scope of the adapter code must be wrapped by the corresponding class adaptation (RootAdapter, ChildAdapter) before being operated on. The class adaptations represent dynamic extensions of the application classes, thus the application class instances should appear to acquire their dynamic types while they are referenced within the composite adapter implementations. The application objects acquire their dynamic types by being wrapped by the adapter objects. The creation of adapter objects is done by factories that memoize the established adapter-adaptee relations, (RootAdapter_Factory and ChildAdapter_Factory shown in the dashed rectangles in Fig. 6 to indicate that they are created within App_Frm_CompositeAdapter as anonymous inner classes of the predefined AdapterFactory).

The solution in Fig 6 does indeed satisfy the requirement that the application can be developed independently from the framework[2]. However, the structure of the adapter is complex and the technique is too advanced to expect it to become a common tool in the toolbox of an average programmer. One should not forget that patterns are conventions that need to be followed, as such they are not enforced by the language and cannot simply be assumed by average programmers [8, 2]. The core of the problem is that there is no

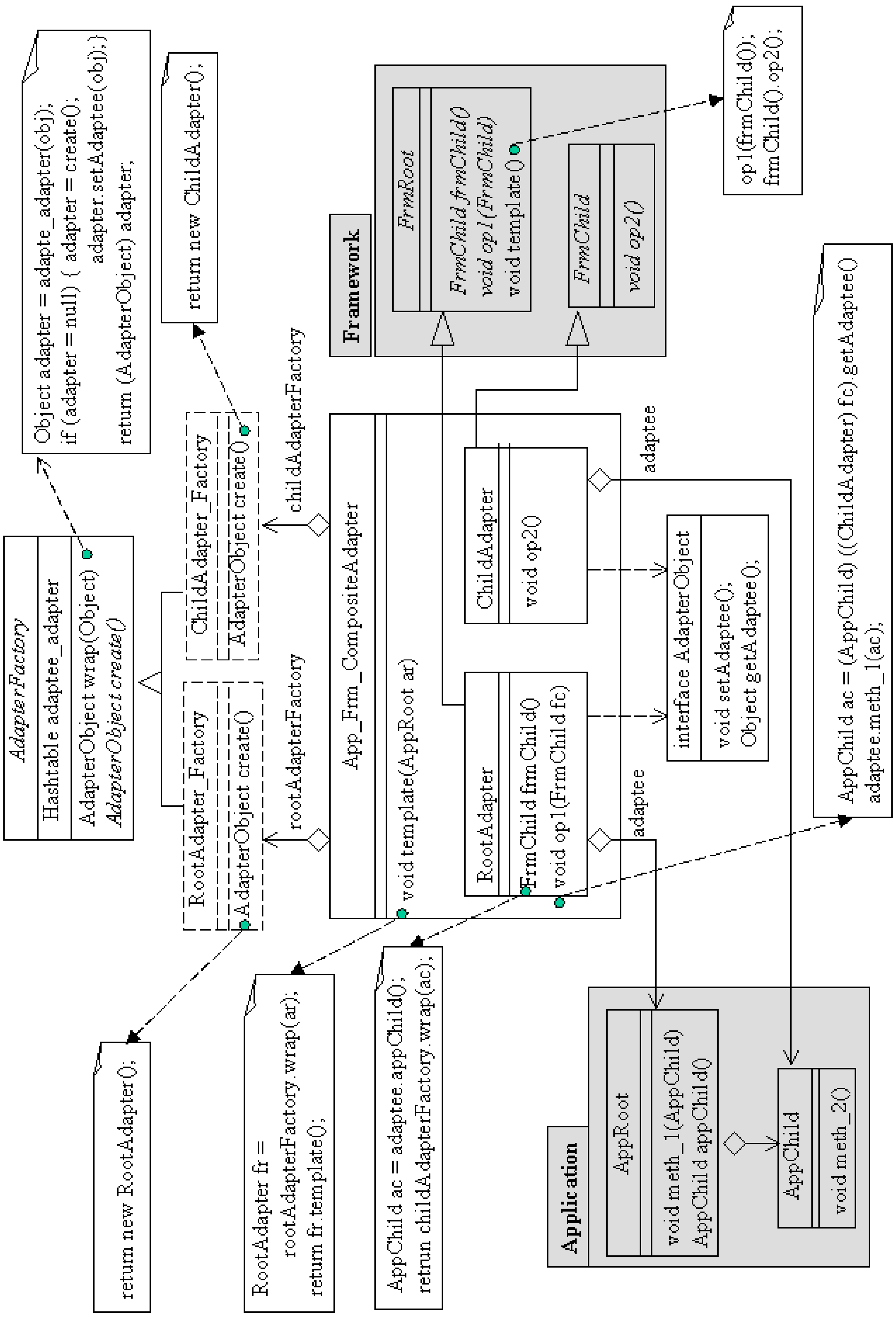

Figure 6. Framework Deployment with the Composite Adapter

direct way to establish some kind of "plays-the-role-of" wiring relationship between classes in different components independently of the component implementations. It is exactly here that the *Plug & Play Adapter* construct comes into play.

3. Plug & Play Adapters

Given the importance of the integration process in component-based development, we argue that it should be supported by dedicated language constructs. We propose to extend object-oriented languages with the *plug & play adapter* construct to serve this need.

- In its simplest form, a *plug & play adapter* defines how to *dynamically extend* a component C_1 with a new concrete collaboration. "Dynamically" means: C_1 objects are adapted on the fly to play the roles in the collaboration without physically changing C_1's classes. For this reason, this use of the *plug & play adapter* is also called *dynamic component adaptation*.

- In addition, a *plug & play adapter* may define how to glue together two independently developed components C_1 and C_2, where C_2 is an abstract collaboration. The gluing results in the component C_1 being dynamically extended with a concrete realization of the abstract collaboration C_2 in terms of the functionality provided by C_1. The adapter thus defines an "adaptation" layer that makes a component and a collaboration fit together. We call this use of the *plug & play adapter* construct *dynamic component gluing*.

In both cases the adapter is dynamically applied – it is dynamically plugged in and out –, hence the attribute *plug & play*.

A collaboration involves a set of objects that work together to implement a task, where each object plays a particular role. Consequently, dynamically extending a component with a collaboration will require the adaptation of several component classes to the appropriate collaboration roles. New variable, method, and class definitions might be needed in order to realize the component adaptation. Hence, the structure of a *plug & play adapter* as presented in Fig. 7.

The nested adapters (i.e. adapter R adapts X ... in Fig. 7) are called *class adapters* because they specify the adaptation of a single class X to its dynamic role R. The enclosing adapter A is called a *composite adapter*, as it nests a set of class adapters. We refer to the component C_1 being adapted as the ground component. In the case of component gluing, the abstract collaboration C_2 that the ground component is being adapted to is referred to as the super component. The composite adapter will serve to dynamically lift (code), resp. lower (decode), from, resp. to, instances of classes in the ground component to, resp. from, the types defined in the nested class adapters (hence, also to types in the super component in the case of component gluing, since the latter will be supertypes of the nested class adapters) whenever this is needed. In other words, the composite adapter serves as a "simulation" of the abstract model it defines, resp. extends, in terms of the concrete model of C_1.

The *extends* clause is put within brackets in Fig. 7 to indicate that it is optional. In the following, we first present the main concepts of the *plug & play adapter* model by considering first the case when there is no *extends* clause, i.e. the case of dynamic component adaptation in which the adapter directly implements a concrete collaboration. We will subsequently consider dynamic component gluing in which the *extends* clause will be used to support the concrete realization of the abstract collaboration given in the super component.

```
adapter A adapts C1 [extends C2] {
    Field_Method_Defs
    Helper_Class_Defs
    { adapter R adapts X [ extends Y ] adaptation_body } *
}
```

Figure 7. The Structure of the Composite Adapter Construct

3.0.1. *Dynamic component adaptation*

Given the component definition in Fig. 7, the class adapter R adapts X adaptation_body implies an adaptation of the functionality of component class C_1.X by the code in the adaptation_body. The adaptation_body encodes the delta by which we would have to enhance the definition of C_1.X to play the given collaboration role R if we used static subclassing, or if we modified the implementation of C_1.X in-place. However, the *adapts* relation implies neither an in-place modification of X, nor does it create a new subclass of C_1.X. Via the *adapts* relation we define a "dynamic" extension of C_1.X, meaning that (existing) objects x: C_1.X appear to acquire the extension given in adaptation_body. The adaptation_body can be thought of as written in terms of three "*self* variables": *A.this* (the composite adapter environment), *R.this* (the role environment), and *X.this* (the adaptee environment). That is, three kinds of definitions are within the scope of the adaptation_body: (a) the definitions in the nested class adapter R, (b) the definitions in component class C_1.X, and (c) the fields, methods and type definitions (classes and class adapters) in the enclosing composite adapter A.

For illustrating the discussion so far, Fig. 8 shows an example of component adaptation in which the regular pricing collaboration is being defined for the Product component. In this example, we assume the abstract pricing framework of Fig. 2 has not yet been developed. Thus, we are intentionally hardwiring the regular pricing algorithm to the product application class model. The regular pricing collaboration is defined in the composite adapter Reg_Pricing_Product, which implements a class adapter for each role in the collaboration, namely LineItemRole, CustomerRole, ItemRole, ChargerRole and

```
adapter Reg_Pricing_Product adapts Product {
  adapter LineItemRole adapts Quote  {
    protected ItemRole item () { return product(); }
    protected CustomerRole customer() { return customer(); }
    protected PricerRole pricer () { return product(); }
    public double price() {
      double basicPrice = pricer().basicPrice(item());
      double discount = pricer().discount(customer(),item());
      double unitPrice = basicPrice - (discount * basicPrice);
      return unitPrice + item().additionalCharge(unitPrice);
    }
  }

  adapter CustomerRole adapts Customer { }

  adapter ItemRole adapts HWProduct {
    protected ChargerRole charge() { return tax(); }
    double additionalCharge(double unitPrice){
      return charge().cost( unitPrice,this );
    }
  }
  adapter ChargerRole adapts Tax  {
    public double cost(double unitPrice, ItemRole item) {
      return taxCharge( unitPrice, item );
    }
  }
  adapter PricerRole adapts HWProduct  {
    public double basicPrice(ItemRole item) {
      return regPrice( item );
    }
    public double discount(CustomerRole cust, ItemRole item) {
      return regDiscount(cust, item );
    }
  }
}
```

Figure 8. Regular Pricing Component Adapter - Dynamic Component Adaptation

PricerRole. Each class adapter implements its role in terms of a Product component class, specified using the *adapts* relation.

The adaptation body within each class adapter is written in terms of three implicit "*self* variables" – the composite adapter, the role adapter and the component adaptee. For instance, the definition of price() in the class adapter LineItemRole calls operations defined in LineItemRole itself, e.g., pricer(), as well as operations defined in Quote, e.g., basicPrice(). The price() method thus has three implicit self references: Reg_Pricing_Product.this, LineItemRole.this, Quote.this. Note also the implicit rebinding (lifting/lowering) of types within the adapter scope. For example, consider the implementation of item() in the LineItemRole class adapter. One might expect a type mismatch between the result of invoking product() – assuming it to be HWProduct – and the return type of item(), which is ItemRole. However, there is no such conflict because the type of this within the definitions of the LineItemRole class adapter is not Quote, but rather Reg_Pricing_Product.LineItemRole – the adaptation of Quote defined within the scope of Reg_Pricing_Product. As a consequence, the ground object returned by product() will automatically be lifted to ItemRole – the adaptation of HWProduct within Reg_Pricing_Product which corresponds to the type that is expected at the place where the HWProduct object at hand came into scope withinReg_Pricing_Product .

The key point in understanding type rebindings is that individual class adapters make sense only within the scope of the enclosing adapter, which is not merely a syntactic construct for encapsulating adaptations for the classes involved in a collaboration (in general, a business process). It has an important semantical function: it defines a "functor" that converts types from the ground component's domain to the types defined within the nested class adapters (and thus to the types in super component's domain in the case of component gluing using the extends clause) and vice versa. The composite adapter thus *lifts* objects from the ground component's domain as they come into its scope, and subsequently lowers them as they leave its scope.

Recall that the definitions in an adaptation body are also written in terms of an implicit composite adapter self reference A.this. Besides binding composite adapter level variable and method definitions, composite adapter self reference A.this also provides a type environment in which type references within the composite adapter are resolved, i.e., playing a similar role for type references to that played by the pseudo-variable this for solving variable references.

adaptationsOf: $C_1\_K \rightharpoonup \mathcal{P}(A\_CA)$

$$adaptationsOf_A(X) = \begin{cases} \{X_1, ..., X_n\} & (\{X, X_1\}, ..., \{X, X_n\} \in adaptedTo_A) \ \wedge \\ & (\nexists X' \in A\_CA \backslash \{X_1, ..., X_n\}, \{X, X'\} \in adaptedTo_A) \\ \\ \perp & \text{otherwise} \end{cases}$$

adaptedTo$_A$ is a relation in $C_1\_K \times A\_CA$, where C_1_K is the domain of classes defined in C_1, and A_CA is the domain of class adaptations defined in A.

$\mathbf{adaptedTo_A} \subseteq C_1\_K \times A\_CA:$
$\forall\ C_1.X \in C_1\_K,\ A.R \in A\_CA,\ C_1.X\ adaptedTo_A\ A.R \iff$
$adapter\ R\ adapts\ C_1.X\ [extends\ C_2.Y]\ adaptation\_body \in A$

Figure 9. The composite adapter type environment

The environment $A.this$ is defined by the $\mathsf{adaptationsOf}_A$ function in Fig. 9, which maps types from the ground component C_1's domain to sets of types in composite adapter A's domain. For instance, given the composite adapter Reg_Pricing_Product in Fig. 8 (RPP for short), the expression $\mathsf{adaptationsOf}_{RPP}(\mathsf{HWProduct})$ is equivalent to the set { ItemRole, PricerRole } Given the type environment $A.this$ of a composite adapter A, type conversions within A obey the rules given in the following paragraphs about type lifting and type lowering.

Type lifting

An object x of type $\mathsf{C}_1.\mathsf{X}$ will automatically be lifted to the type $\mathsf{A.R} \in \mathsf{adaptationsOf}_A(\mathsf{C}_1.\mathsf{X})$ when x comes into the scope of an adaptation_body nested within adapter A in a context where an object of type A.R (or of the supertype of A.R in C_2, if A.R is defined by an *adapts extends* construct) is expected. Such an object comes into the scope of a class adaptation body as the result of either (a) invoking an operation that returns an object of type $\mathsf{C}_1.\mathsf{X}$, or (b) directly instantiating $\mathsf{C}_1.\mathsf{X}$.

Lifting x: $\mathsf{C}_1.\mathsf{X}$ to A.R means (a) creating an instance of R, i.e, an environment *R.this*, and (b) establishing the relations between the newly created adapter object *R.this*, its corresponding ground object (*X.this*), and the enclosing composite adapter within which the lifting takes place ($A.this$). Hence, at this point all three *"self variables"* implicitly referred to within the adaptation body of R are bound. The relationship between $A.this$ and *R.this* is essentially the conceptual relation between a Java inner class instance and the enclosing toplevel class instance. The relationship beteen *R.this* and *X.this* is an aggregation in the current realization of the adapter construct. Other alternatives are discussed in the following section.

The adapter construct "remembers" the result of lifting a component class x: $\mathsf{C}_1.\mathsf{X}$ to its role A.R. That is, if component object x goes in and out of the scope of composite adapter A several times during the execution of operations defined in A, it will not be lifted to different A.R role adapter objects, i.e., the invocation history in the context of A does not get lost. This is crucial as the class adapter may add role-based state to the component object.

For illustration, consider the methods item() and pricer() in Fig. 11. They both are implemented by calling the product() operation from Quote, thus both obtain a HWProduct object when called on a Quote object. Each call to product() causes a HWProduct object to come into the scope of the Reg_Pricing_Product environment. However, the item() method requires the HWProduct object to be lifted to the ItemRole type since it comes into scope in a context where an ItemRole instance is expected, while the pricer() method requires the HWProduct object to be lifted to PricerRole since it comes into scope in a context where a PricerRole object is expected.

Type lowering

Any object x of type $\mathsf{C}_1.X$ that was lifted to type A.R should be lowered back to type $\mathsf{C}_1.X$ before it can leave the adapter scope. This occurs when within the adaptation body of some nested class adaptation the object x is either (a) passed as a parameter into an operation defined in any of the classes in ground component C_1, or (b) an operation defined in class C_1.X is directly invoked on the object x. Lowering the lifted version of x back to type $C_1.X$ results in x.

For illustration, consider the implementation of the PricerRole method discount(CustomerRole cust, ItemRole item) in Fig. 8. Passing the cust argument as the first actual parameter to HWProduct.regDiscount(Customer, HWProduct) implies an automatic lowering of cust: CustomerRole to its ground component Product.Customer object. Similarly, the item argument will be lowered to the ground component Product.HWProduct.

The commutative diagram in Fig. 10, recapitulates the discussion about the automatic type lifting and lowering performed within the scope of a composite adapter. ASR (adapter-level structural relation) stands for a structural relation between two types defined in a composite adapter, A.R and A.R', whereby the relation is realised via a structure mapping method. For instance, $item : LineItemRole \rightarrow ItemRole,\ item(li) = li.item()$, is an example for an ASR between Reg_Pricing_Product.LineItemRole and Reg_Pricing_Product.ItemRole (cf. adapter in Fig. 8). On the other hand, BSR (base-level structural relation) is the (computed) structural relation between two base types, C_1.X and C_1.X, where $R \in adaptationsOf_A(X)$ and $R' \in adaptationsOf_A(X')$, that implements ASR. For instance, $product : Quote \rightarrow HWProduct,\ product(q) = q.product()$ is the base structural relation that implements the adapter structural relation item, above.

$$\begin{tikzcd} \mathbf{A.R} \arrow[r, "ASR"] \arrow[d, "lower"] & \mathbf{A.R'} \\ \mathbf{C1.X} \arrow[r, "CSR"] & \mathbf{C1.X'} \arrow[u, "lift"'] \end{tikzcd}$$

Figure 10. The relation between type lifting and lowering operations

Plugging adapters in at runtime

Given a composite adapter A that adds a collaboration to a component C_1 and assuming that the root of the collaboration is a class adapter RR (for root role), the adapter is dynamically plugged in by applying it to an object x: $C_1.X$, where RR $\in$ adaptationsOf$_A$(C_1). This is done by the adapter application operator ($< A >$ x). Note that we are assuming there is a single adaptation class for C_1.X in A. If there are several class adaptations for C_1.X in A, the application operator would explicitly specify which one to use. As the result of the adapter application, a new composite adapter instance get created, together with a new class adapter instance for RR.

After being applied to x the adapter A transforms the object aggregate of component C_1 starting at the composite root x: $C_1.X$, into a lifted object aggregate, starting at an object of RR. This is done in a "lazy way": new class adapter instances are created when new children in x's object graph come into A's scope as the result of invoking methods on already lifted parts of x's aggregate. All class adapter instances created this way share the same scope represented by the composite adapter *A.this*.

For illustration, Fig. 11 contains sample client code that dynamically plugs in the functionality of the Reg_Pricing_Product adapter. This allows the regular pricing functionality to be available to the quote object q.

```
public class Client {
   static public void main(String[] args) {
      Quote q = new Quote( ... );
      System.out.println(<Reg_Pricing_Product> q).price());
}
```

Figure 11. Applying a Class Adaptation

3.0.2. *Dynamic component gluing*

Given components C_1 and C_2, the existence of an *extends* clause in a nested class adaptation R adapts X extends Y puts the classes C_1.X and C_2.Y in a target-adaptee relation in terms of the adapter design pattern [5]. The nested class adaptation R expresses the relationship C_1.X plays-the-role-of C_2.Y. The adaptation_body defined in class adapter R encodes the delta by which we would have to enhance the definition of C_1.X if we used static subclassing to express the C_1.X plays-the-role-of C_2.Y relationship.

For illustration, the composite adapter for gluing the concrete product application and the abstract pricing framework component according to the regular pricing scheme is given in Fig. 12. While the composite adapter in Fig. 8 directly implemented the regular pricing collaboration, the adapter in Fig. 12 implements the regular pricing collaboration by extending the abstract pricing framework. Note again the implicit rebinding of types within the adapter scope, e.g., the return type ItemRole of the item() method and the return type HWProduct of the product() method that is called within item(). There is no conflict among these types because the type of this within the definitions of the LineItemRole class adapter is not Quote, but rather Reg_Pricing_Product.LineItemRole. Also, the return type of calling product() on the rebound this is not HWProduct; it is rather the extension of the latter within the scope of the enclosing adapter, Reg_Pricing_Product.HWProduct, which is a subtype of ItemRole. Hence, the return type of item() is substitutable for the return type of product().

3.0.3. *The adapter construct at work*

In the following, we show how the adapter construct improves the modularity and extensibility of systems that result from assembling several components

```
adapter Reg_Pricing_Product adapts  Product extends Pricing {
  adapter Quote_LineItemRole adapts Quote
              extends LineItemRole {
    protected ItemRole item () { return product(); }
    protected CustomerRole customer() { return customer(); }
    protected PricerRole pricer () { return product(); }
  }
  adapter Customer_CustomerRole adapts Customer
                 extends CustomerRole {}
  adapter HWProduct_ItemRole adapts HWProduct
                           extends ItemRole {
    protected ChargerRole charge() { return tax(); }
  }
  adapter Tax_ChargerRole adapts Tax extends ChargerRole {
    public double cost(double unitPrice, ItemRole item) {
           return taxCharge(unitPrice, item); }
  }
  adapter HWProduct_PricerRole adapts HWProduct
                             extends PricerRole {
    public double basicPrice(ItemRole item) {
      return regPrice( item );
    }
    public double discount(CustomerRole cust, ItemRole item) {
      return regDiscount( cust, item);
    }
  }
}
```

Figure 12. Regular Pricing Component Adapter

(business processes) in terms of our running example. Along the way, new features of the *plug & play adapter* model will be introduced as needed.

The main advantage of the adapter construct presented so far is that it facilitates the separation of concerns (functional ingredients of a system are separated from each other as well as the gluing concerns). This leads to improved modularity, hence, readability, maintainance, and more flexible extensibility. For illustrating these claims let us compare how easy it is to integrate a new pricing scheme (in general a new business process or multi-object collaborative functionality) when using traditional composition techniques versus an adapter construct. Assume that we have integrated the product and pricing components from the previous section (cf. see the class models in Fig. 2 and 3) according to the regular pricing scheme. During this integration we have not anticipated integrations of alternative schemes. Thus, the class model will include only the original product classes modified as subclasses of the corresponding framework classes, along with the single adapter class HWProduct_PricerRole (cf. see Fig. 4).

```
adapter Sale_Pricing_Product extends Reg_Pricing_Product {
   adapter HWProduct_PricerRole adapts HWproduct
                                extends PricerRole {
       public double basicPrice(Pricing.ItemRole item) {
            return salePrice();
       }
       public double discount(CustomerRole cust, ItemRole item) {
            return saleDiscount(cust);
       }
   }
 }
```

Figure 13. Sale Pricing Component Adapter

Now, assume we want to extend the system to support sale pricing. This extension will affect several places in the design in Fig. 4, requiring a subclass of HWProduct_PricerRole to be added, Quote to be modified to use a strategy object, a strategy class for PricingScheme to be added, along with the appropriate subclasses. Alternatively, we can simply define a new composite adapter Sale_Pricing_Product by incrementally refining the existing composite adapter Reg_Pricing_Product that was given in Fig. 12. The new adapter is shown in Fig. 13.

Adapers as extensions of other adapters

The meaning of the extends relationship between adapters is similar to the extends relation between classes in Java. Defining an adapter SA as a subadapter of adapter A means that SA (a) inherits the ground and super components declarations from A, (b) inherits all nested class adaptations from A, (c) can define new nested class adaptations, and (d) can incrementally refine class adaptations nested within A. For instance, Sale_Pricing_Product inherits the Quote_LineItemRole, Customer_CustomerRole, Tax_ChargerRole and HWProduct_ItemRole class adaptations defined in the Reg_Pricing_Product, while overriding the HWProduct_PricerRole class adaptation.

To illustrate how adapters can incrementally refine definitions from their super-adapters as well as add new adaptations, consider the following "re-engineering" of our adaptation code. One can factor out the commonalities of all pricing schemes in an abstract adapter – Pricing_Product in Fig. 14. Reg_Pricing_Product is then defined as a subadapter of it, by (a) incrementally refining the Quote_LineItemRole adapter with an implementation for pricer() and (b) defining a new class adaptation, HWProduct_PricerRole.

A composite adapter adapter A adapts Ground extends Super ... is abstract if (a) one of its class adapters is abstract, or (b) there is an abstract class in the Super collaboration for which there is no class adaptation in the composite adapter. For instance, Pricing_Product in Fig. 14 is abstract because the class

```
abstract adapter Pricing_Product adapts Product
                                 extends Pricing {
    abstract adapter Quote_LineItemRole adapts Quote
                                        extends LineItemRole {
        protected ItemRole item () { return product(); }
        protected CustomerRole customer() { return customer();}
     }
     adapter Customer_CustomerRole adapts Customer
                                   extends CustomerRole {
                 /* as in original Reg_Pricing_Product */ }
     adapter HWProduct_ItemRole adapts HWProduct
                                extends ItemRole {
                 /* as in original Reg_Pricing_Product */ }
     adapter Tax_ChargerRole adapts Tax extends ChargerRole {
                 /* as in original Reg_Pricing_Product */ }
}


adapter Reg_Pricing_Product extends Pricing_Product {
     adapter Quote_LineItem extends Quote_LineItem {
          protected PricerRole pricer () {return product();}
     }
     adapter HWProduct_PricerRole adapts HWProduct
                                  extends PricerRole {
                /* as in previous Reg_Pricing_Product */ }
  }
```

Figure 14. Extension of Abstract Pricing Component Adapter

adapter Quote_LineItemRole is still abstract (the abstract method pricer() from Pricing.LineItemRole remains unimplemented).

Adapting an adapter

Adapters can adapt other adapters as well, a feature that becomes very handy when business processes are layered on top of each other. A sample scenario for illustrating this is the following. Assume that after we have extended the bare business model of our HWProduct supplier with the pricing functionality, the integrated system needs to be further extended with the ability to calculate the total of a given Order. Furthermore, assume that some generic component (a mini-framework) is available for calculating the sum of a certain value in a composite object structure; the class model for this component is given in Fig. 15. Summing is modeled as the template method sum() in Composite. What is summed is left open (the method value() in Elements is left abstract), as it will vary in different contexts where the summing component might get used. The integration of the summing component into the product application is specified by the Summing_Pricing_Product composite adapter in Fig. 16 with the composite adapter Pricing_Product and Summing as its ground and super components.

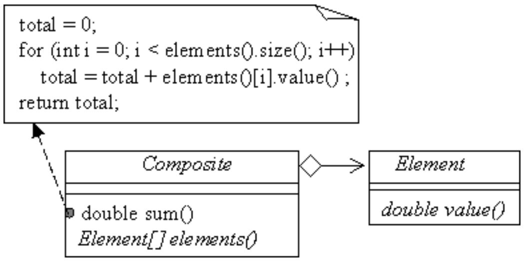

Figure 15. Summing Component

An adapter A, whose ground component is another adapter GA, extends GA dynamically with new class adapters adapting either (a) GA's ground component's classes, or (b) GA's class adapters themselves. For instance, Summing_Pricing_Product defines two class adapters. Order_Composite adapts the Order class in the ground component of the Pricing_Product composite adapter, while Quote_Element adapts the class adapter Quote_LineItemRole defined in Pricing_Product.

Let us consider the type lifting for the return value of the call to quotes() in the implementation of elements() in Order_Composite. Two type liftings will happen at this place in order to rebind the type of the elements of the array returned by quotes() (the base type Quote) to Quote_Element – the subtype of Summing.Element defined in Summing_Pricing_Product. The type environment of a composite adapter A that has another composite adapter as its ground component, GA, is a composition of the type environment *GA.this* and the delta type environment made up of the local adaptations in the body of A. If we denote this delta environment as $A\_GA.this$, we have $A.this = GA.this * A\_GA.this$, or $\mathsf{adaptationsOf}_A(\mathsf{X}) = \mathsf{adaptationsOf}_{GA}(\mathsf{X})$

```
adapter Summing_Pricing_Product adapts Pricing_Product
                                extends Summing {
       adapter Order_Composite adapts Product.Order
                                             extends Composite {
              Elements[] elements() { return quotes(); }
        }
        adapter Quote_Element adapts Quote_LineItemRole
                                           extends Element {
             double value() { return price(); }
        }
}
```

Figure 16. Integrating Summing with Pricing and Product (1)

* $\mathsf{adaptationsOf}_{A\_GA}(\mathsf{X})$. The semantics of the composition operator * in a simplified form is as follows. If (a) there is a path GA.R $\in$ $\mathsf{adaptationsOf}_{GA}(\mathsf{X})$ $\wedge$ A.R $\in$ $\mathsf{adaptationsOf}_{A\_GA}(\mathsf{GA.R})$, and (b) A.R is the type expected where an sf x: X comes into scope of A, then x will be first lifted to GA.R and the result of this lifting will be lifted to A.R.

In our example, the objects that come into scope are of type Quote and the expected type is Quote_Element. There is an adaptation path: Quote $\rightarrow$ Pricing_Product.Quote_LineItemRole $\rightarrow$ Summing_Pricing_Product.Quote_Element that connects these types. Hence, each Quote object q contained in the array returned by quotes() will be lifted to a ql: Pricing_Product.Quote_LineItemParty and the latter will be lifted to a qe: Summing_Pricing_Product.Quote_Element.

We could have as well used the extends rather than the adapts relationship between adapters in order to define Summing_Pricing_Product. However, adapts is in this case preferrable in order to avoid subclassing all Pricing_Product subclasses. Static subclassing results in (a) a proliferation of classes: Summing_Reg_Pricing_Product, Summing_Neg_Pricing_Product, Summing_Sale_Pricing_Product, and (b) duplication of the code in the Summing_Pricing_Product adaptation body in Fig. 16. Using the adapts relation rather than extends allows us to reuse the adaptation encoded within the body of Summing_Pricing_Product adapter with instances of all subadapters of Pricing_Product. This is illustrated by the sample client code in Fig. 17.

```
public class Client {
  static public void main(String[] args) {
      Order o = new Order();
      ...
      System.out.println( (<Total_Pricing_Product>
                          (<Reg_Pricing_Product> o)).total());
      System.out.println( (<Total_Pricing_Product>
                          (<Sale_Pricing_Product> o)).total());
  }
}
```

Figure 17. Using the Summing_Pricing_Product Adapter

In the discussion concerning dynamic component gluing so far the ground component has been a concrete application. Adapters can also be used to glue together two abstract collaborations, as illustrated in Fig 18, where we first glue together the abstract collaborations Summing and Pricing, resulting in the abstract composed collaboration Total_Pricing and then glue the latter with Product by specifying Total_Pricing_Product and its subadapters.

The last examples show that class adapters can adapt classes as well as other class adapters, and composite adapters can adapt components that are composed of simple (non-adapter) classes or other composite adapters. Actually, this is a natural consequence of the fact that class adapters are special kinds of classes and composite adapters are special kinds of components. Fig. 19 clearly depicts this, showing a meta-model for programming constructs

```
abstract adapter Total_Pricing adapts Pricing extends Summing {
  abstract adapter Pricing_Total extends Composite {
      Elements[] elements() { return LineItemRole[] lineItems(); }
      abstract LineItemRole[] lineItems();
  }
  abstract adapter LineItemRole adapts Pricing.LineItemRole
          extends Element {
      double value() { return price(); }
  }
}

abstract adapter Total_Pricing_Product adapts Product
                                       extends Total_Pricing {
    adapter Order adapts Product.Order extends Pricing_Total {
        LineItemRole[] lineItems() { return quotes(); }
    }
  class Quote_LineItemRole adapts Quote extends LineItemRole {
     // ... /* as in Pricing_Product  */
  }

  class HWProduct_ItemRole adapts HWProduct extends ItemRole {
    // ... /* as in Pricing_Product  */
  }
  // ... as in Pricing_Product
}

adapter Reg_Total_Pricing_Product extends Total_Pricing_Product {
    ... /* as in Reg_Pricing_Product */
}
```

Figure 18. Integrating Summing with Pricing and Product (2)

including *plug & play adapters*. Note that in this model (and throughout our discussion), we have made the silent assumption that components (i.e., collaborations, business processes) are modeled in dedicated constructs. This implies either the use of special models, e.g., *Adaptive Plug & Play Components* [13], or the use of appropriate language constructs in mainstream languages, such as Java inner classes.

4. Alternative Realizations of the Adapter Construct

Alternative realisations of the adapter construct can be classified in two main groups:

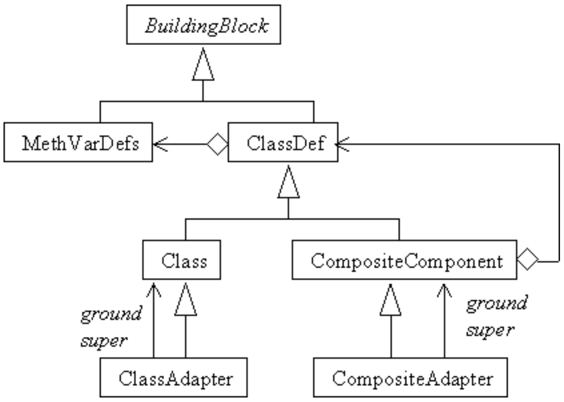

Figure 19. Metamodel of Constructs in the Plug & Play Adapter Model

(a) Global scope:
The modifications specified by the adapter are globally visible in the sense that after compiling adapter A { ... adapter R adapts X extends Y adaptation_body ... } only an in-place modified X is visible with the new definition being equivalent to class X extends Y { X-def adaptation_body}. Assuming only single inheritance is supported and X already uses the sole inheritance link before the adaptation defined in A, one should think of the resulting definition of X as being equivalent to class X { X_defs Y_defs adaptation_body}, i.e., Y's definition after inheritance flattening gets "copied" within X. The client's use of the integrated functionality is as in the sample code for our pricing example in Fig. 20. This is the generative case that applies the adapter statically, i.e., at adapter compilation time or during class loading. Either source code generators or binary component adaptation tools [6, 3] can be used for this purpose.

```
public class Client {
 static public void main(String[] args) {
    Quote q = new Quote();
    System.out.println(q.regular_price());
 }
}
```

Figure 20. Using a Globally Scoped Regular Pricing

The primary advantage of static, invasive integration is efficiency, as it will avoid the need for excessive delegation to an adapter object. The primary disadvantage is loss of flexibility. Enforcing the component adaptation to occur at either compile or load time restricts a runtime object from acquiring the services of new components that become available after its instantiation. Thus only anticipated collaborations are supported. As it modifies the class, all class instances are affected. Thus, it is not possible to be selective as to which objects acquire the new class semantics. Furthermore, objects become very heavy, in that they must support the state required for all of the possible collaborations in which they might participate. This also contrasts the Java philosophy of loading program constructs as they are needed, and not before.

(b) Local scope

The adapter introduces a new name space, which means that there will be two definitions of X after the adapter is compiled: the original X as in the component C_1 and its role in a given collabaration, A.R. Two subcases can be distinguished here:

```
public class Client {
 static public void main(String[] args) {
    Quote q = new Quote();
    System.out.println((<Reg_Pricing_Scheme> q).price());
 }
}
```

Figure 21. Using a Locally Scoped Regular Pricing Adapter

1. Transparent scope: Given adapter A adapts C_1 extends C_2 ... adapter R adapts X extends Y adaptation_body ... , we have that A.R is substitutable for X. This is the case if A.R is generated as class A.R extends Y { X-defs adaptation-code}. That means, if x is an instance of X, than <A>x provides the original interface of X plus the interface of A.R.

2. Opaque scope: A.R is a "view" over X, meaning that for object x we can call methods in the original interface of X, while we have to convert to $< A > x$ in order to call methods from Y.

The composite adapter pattern presented in [14] (the structure of which was shown in Sec. 2) provides (Java) programmers with a language idom to simulate the adapter construct based on the approach (b.2). A simulation of adapters with transparent scope (b.1) requires extended models that support object-based inheritance as e.g., the *Darwin* model and its Java based realization called *Lava* presented in [7]. Assuming *Lava* as the the underlying language, modifying the technique in [14] to support transparent scope essentially means replacing the *adaptee* relation between role and ground objects (i.e., forwarding semantics) in Fig. 4 with a *delegatee* relation (i.e., true delegation semantics with late-binding of self). The rest of the pattern remains the same.

However, as already mentioned, the pattern presented in [14] is too low-level. For experimental purposes, we have developed a more abstract version of the technique [resented in [14] making use of the reflective facilities of Java. In few words, the technique is as follows. The programmer defines all composite adapter classes as direct or indirect subclasses of a predefined class

CompositeAdapter. On the first call of its constructor a CompositeAdapter makes use of the reflective API of Java in order to automatically establish the infrastructure of adapter factories (cf. Fig. 4) as well the infrastructure needed for managing the adapter–adaptee relations. In this way the work of the programmer is facilitated, in that he/she does not need to (a) implement the AdapterObject interface from Fig. 4 for each inner class adapter, and (b) take care of defining and intializing the factory objects for each class adapter. However, he/she still needs to explicitly call lifting/lowering operations at the appropriate places. Further details about this realization are left out of the scope of this paper. The interested reader can find more information and sample code for the examples in the paper in [**?**]

The prototype outlined above allows us to experiment with the adapter within the context of a mainstream language such as Java. In a long run, however, a preprocessor will be needed for static transformation of the adapter bodies to generate the corresponding composite adapter class as it would be hand-written by the programmer, if he/she used the pattern decsribed in [14].

5. Related Work

VanHilst and Notkin propose an approach for modeling collaborations based on templates and mixins as an alternative to using frameworks [18]. However, this approach may result in complex parameterizations and scalability problems. Smaragdakis and Batory solve this by elevating the concept of a mixin to multiple class granularity, using C++ parameterized nested classes [15]. However, their approach does not address the issue of dynamic customizations as described by Holland [9]. A *Contract* [9] allows multiple, potentially conflicting component customizations to exist in a single application. However, contracts do not allow conflicting customizations to be simultaneously active. Thus, it is not possible to allow different instances of a class to follow different collaboration schemes.

Seiter et al. proposed a *context relation* to link the static and dynamic aspects of a class [16]. While supporting multiple dynamic collaboration schemes, the approach is based on dynamically altering a class definition for the duration of a method invocation, thus affecting all class instances. Multiple dynamic variations of an object's behavior are also supported in the RONDO model [12]. However, RONDO does not provide explicit support for collaborations. In this paper we propose a model for scoping the different collaboration schemes, thus we can be selective as to which objects are affected.

Batory proposed the *GenVoca* architecture to define parameterized, plug-compatible, interchangeable and interoperable components [1]. The *GenVoca* model is based on the notion of *realm, interface*, *component* and *layer*. Layers represent encapsulations of composite-object decorators, which could be dynamically composed. The technique we have presented can be used as an elegant Java implementation for *GenVoca* layers. Einarson and Hedin also suggest the use of inner classes as alternative implementations of several design patterns [4].

Patersson et al. [11] also indicate the problems with framewrok composition, analyze reasons for these problems, and investigate the state of the art of available solutions. Bosch argues that language support should be provided for explicitly describing design patterns in object-oriented programs [2]. Among supporting other patterns, he also provides a language construct for specifying a class as the adapter of another class, i.e., for explicit expression of the adapter pattern [5]. The adapter construct as proposed in [2] has two main restrictions. First, it does not support adaptation of entire collaborative functionality. Second, as indicated in [2], it does not allow interface incompatibility.

A key theme of the work described in this paper is separation of concerns to avoid software tangling. This is also the motivation behind both *Aspect-Oriented Programming* [22] and *HyperSpaces* (a new model of subject-oriented programming) [17]. AspectJ [22] is an extension of Java that allows one to program different aspects 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 in AspectJ is described in [10]. This improved form of AP&PC uses similar techniques as described in this paper, along with tool support.

Summary and Future Work

This paper studied traditional framework customization techniques and concluded that they are inappropriate for component-based programming since they lack support for non-invasive, encapsulated, dynamic customization. We proposed a new language construct for expressing component gluing explicitly and discussed an implementation technique for dynamic framework customization, based on Java inner classes. The technique allows the separation of customization code from application and framework implementations, thus supporting the gluing of pre-existing components by third-parties.

Notes

[1] The definition of this component is first described in [9].

[2] The technique has been originally formulated for framework-application integration, but it can be easily extended to serve the case when two fully-implemented components are to be integrated (replacing the inheritance links by usage links).

References

1. D. Batory, The GenVoca Model of Software-System Generators. in *IEEE Software*, September 1994.
2. J. Bosch. Design Patterns as Language Constructs. In *Journal of Object-Oriented Programming*, 1997.

3. G. Cohen, J. Chase, and D. Kaminsky. Automatic Program Transformation with JOIE. In *USENIX 1998 Annual Technical Conference*, pp. 167–178, June 1998.
4. D. Einarson, G. Hedin, Using Inner Classes in Design Patterns, Code examples available at <http://www.dna.lth.se/home/daniel/patterns_inner_classes.html>.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
6. U. Hölzle and R. Keller. Binary Component Adaptation In *Proceedings of ECOOP '98*, LNCS XXX, pp. 307–329, Springer Verlag, 1998.
7. G. Kniesel Type-Safe Delegation for Run-Time Component Adaptation In *Proc. of ECOOP '99*, LNCS XXX, pp. xxx–xxx, Springer Verlag 1999.
8. J. Gil and D. Lorenz. Design Patterns and Language Design. In *IEEE Computer*, pp. 118–120, March 1998.
9. I. Holland, The Design and Representation of Object-Oriented Components, Ph.D. Dissertation, Northeastern University, Computer Science, 1993.
10. K. Lieberherr, D. Lorenz, and M. Mezini, Modeling Aspects with Adaptive Plug & Play Components, College of Computer Science, Northeastern University, Technical Report, Boston, MA, 1999.
11. M. Mattson and J. Bosch and M. Fayad Framework Integration Problems, Causes, Solutions. *Communications of ACM*, Vol. 42, No. 10, pp. 80-87, October 1999
12. M. Mezini. Variational Object-Oriented Programming Beyond Classes and Inheritance. Kluwer Academic Publishers, 1998.
13. M. Mezini and K. Lieberherr, Adaptive Plug and Play Components for Evolutionary Software Development. In *Proc. of OOPSLA '98*, ACM Sigplan Notes, Vol. xxx, No. xxx, ACM Press, October 1998.
14. L. Seiter, M. Mezini, K. Lieberherr, Dynamic Component Gluing. Proc. First Symposium on Generative and Component-Based Software Engineering, September 1999. Springer Verlag, Lecture Notes in Computer Science. adapter design pattern code. Available at <http://www.cse.scu.edu/˜ lseiter/cgse/codeexamples/>.
15. Y. Smaragdakis and D. Batory, Implementing Layered Designs with Mixin Layers, in *Proc. ECOOP'98*, 550-570. Springer Verlag, Lecture Notes in Computer Science.
16. L. Seiter, J. Palsberg, and K. Lieberherr, Evolution of Object Behavior using Context Relations, in *IEEE Transactions on Software Engineering*, January 1998.
17. P. Tarr, H. Ossher, W. Harrison, S. Sutton Jr., *N* Degrees of Separation: Multi-Dimensional Separation of Concerns, In *ICSE'99*. May 1999.
18. M. VanHilst and D. Notkin, Using Role Components to Implement Collaboration-Based Designs, In *OOPLSA'96*. ACM Press.
19. IBM San Francisco Technical Summary http://www.software.ibm.com/ad/sanfrancisco/about.html
20. Enterprise Java Beans MIRA which citation did you want here?
21. Clemens Szyperski Component Software: Beyond Object-Oriented Programming. Addison-Wesley 1998
22. Xerox PARC AspectJ Team, AspectJ, Xerox PARC Technical Report, January 1999, http://www.parc.xerox.com/spl/projects/aop/