References

1. D. Batory, The GenVoca Model of Software-System Generators. in *IEEE Software*, September 1994.
2. 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>.
3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.
4. I. Holland, The design and representation of object-oriented components, Ph.D. Dissertation, Northeastern University, Computer Science, 1993.
5. K. Lieberherr, Lorenz, and M. Mezini, Aspect-Oriented Components, College of Computer Science, Northeastern University, Technical Report, Boston, MA, 1999.
6. M. Mezini. Variational Object-Oriented Programming Beyond Classes and Inheritance. Kluwer Academic Publishers, 1998.
7. M. Mezini and K. Lieberherr, Adaptive Plug and Play Components for Evolutionary Software Development. Proc. OOPSLA, October 1998. ACM Press.
8. L. Seiter, M. Mezini, K. Lieberherr, Dynamic composite adapter design pattern code. Available at <http://www.cse.scu.edu/~ lseiter/cgse/codeexamples/>.
9. Y. Smaragdakis and D. Batory, Implementing Layered Designs with Mixin Layers, in *Proc. ECOOP'98*, 550-570. Springer Verlag, Lecture Notes in Computer Science.
10. L. Seiter, A. Gunawan, A Java architecture for dynamic object and framework customizations, Technical Report, Santa Clara University, 1998.
11. L. Seiter, J. Palsberg, and K. Lieberherr, Evolution of Object Behavior using Context Relations, in *IEEE Transactions on Software Engineering*, January 1998.
12. P. Tarr, H. Ossher, W. Harrison, S. Sutton Jr., *N* Degrees of Separation: Multi-Dimensional Separation of Concerns, In *ICSE'99*. May 1999.
13. M. VanHilst and D. Notkin, Using Role Components to Implement Collaboration-Based Designs, In *OOPLSA'96*. ACM Press.
14. Xerox PARC AspectJ Team, AspectJ, Xerox PARC Technical Report, January 1999, http://www.parc.xerox.com/spl/projects/aop/

Holland [4]. A *Contract* [4] 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 [11]. 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 [6]. 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 for 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 inner class 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 [2].

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* [14] and *HyperSpaces* (a new model of subject-oriented programming) [12]. AspectJ [14] 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 [5]. This improved form of AP&PC uses similar techniques as described in this paper, along with tool support.

Summary

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 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.

Acknowledgements

This work has been partially sponsored by the National Science Foundation under grant number CDA-972057.

the essence of the pricing framework deployment, using the new *adapter* scoping construct.

```
abstract adapter AbstractPricingScheme {
  abstract Quote extends LineItemParty {
    ItemParty item () { return product(); }
    CustomerParty customer() { return customer(); }
  }
  Customer extends CustomerParty {}
  HWProduct extends ItemParty {
    ChargerParty charge() { return tax(); }
  }
  Tax extends ChargerParty {
    public double cost(double unitPrice, ItemParty item)
    { return taxCharge(unitPrice, (HWProduct) item); }
  }
}

adapter RegPricingScheme extends AbstractPricingScheme {
  Quote extends AbstractPricingScheme.Quote {
    PricerParty pricer () { return product(); }
  }
  HWProduct extends PricerParty {
    public double basicPrice(ItemParty item) { return regPrice();}
    public double discount(CustomerParty cust, ItemParty item)
    {  return regDiscount( (Customer)cust );  }
  }
}
//example main program that applies the adapter to an application object
public class Client {
 static public void main(String[] args) {
  Quote q = new Quote();
  System.out.println(((RegPricingScheme)q).price());
}
```

Fig. 9. Pricing Component Adapter - Succinct Specification

4 Related Work

VanHilst and Notkin propose an approach for modeling collaborations based on templates and mixins as an alternative to using frameworks [13]. 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 [9]. However, their approach does not address the issue of dynamic customizations as described by

by all three pricing schemes; concrete adapter classes for each individual pricing scheme will subclass it. *AbstractPricingScheme* contains an inner class for each role in the pricing framework. Each inner class implements the abstract framework methods based on a specific pricing scheme, and also maps a part of the application composite object to the appropriate framework role. Note that *AbstractPricingScheme* does not define an inner class for the *PricerParty* role. The deployment for this role is specific for concrete schemes and will be provided by concrete subclasses *RegPricingScheme*, *NegPricingScheme*, and *SalesPricingScheme*. *RegPricingScheme* is given in Fig. 8 for illustration.

```
class RegPricingScheme extends AbstractPricingScheme {
  protected Hashtable tableHWProduct_PricerParty;
  RegPricingScheme(){super(); tableHWProduct_PricerParty=new Hashtable();}
  public double price(Quote q) {
    Object o = wrap(tableQuote_LineItemParty, q, new Quote_LineItemParty());
    return ((Quote_LineItemParty)o).price();
  }

  protected class Quote_LineItemParty
  extends AbstractPricingScheme.Quote_LineItemParty {
    protected PricerParty pricer() {
      Object o = wrap(tableHWProduct_PricerParty,
        adaptee.product(), new HWProduct_PricerParty());
      return (HWProduct_PricerParty)o;
    }
  }

  protected class HWProduct_PricerParty
  extends PricerParty implements AdapterObject {
    protected HWProduct adaptee;
    public void setAdaptee(Object o) { adaptee = (HWProduct)o; }
    public Object getAdaptee() { return adaptee; }
    public double basicPrice(ItemParty item) { return adaptee.regPrice();}
    public double discount(CustomerParty cust, ItemParty item) {
      Object o = ((AdapterObject)cust).getAdaptee();
      return adaptee.regDiscount( (Customer)o );
    }
  }
}
```

Fig. 8. Regular Pricing Adapter

Note that in Figs. 7 and 8 it is not obvious how the application classes fulfill the framework roles, as much of the code is related to the *dynamic composite adapter* implementation technique. Fig. 9 on the other hand clearly captures

```
abstract class AbstractPricingScheme extends ApplicationAdapter {
  protected Hashtable tableQuote_LineItemParty, tableCustomer_CustomerParty,
                      tableHWProduct_ItemParty, tableTax_ChargerParty;
  AbstractPricingScheme() {
    tableQuote_LineItemParty = new Hashtable();
    ...  //instantiate other hash tables
  }

  protected abstract class Quote_LineItemParty
  extends LineItemParty implements AdapterObject {
    protected Quote adaptee;
    public void setAdaptee(Object o) { adaptee = (Quote)o; }
    public Object getAdaptee() { return adaptee; }
    protected ItemParty item() {
      Object o = wrap(tableHWProduct_ItemParty,
        adaptee.product(), new HWProduct_ItemParty());
      return (HWProduct_ItemParty)o;
    }
    protected CustomerParty customer() {
      Object o = wrap(tableCustomer_CustomerParty,
        adaptee.customer(), new Customer_CustomerParty());
      return (Customer_CustomerParty)o;
    }
    //concrete pricing schemes will implement pricer()
  }

  protected class Customer_CustomerParty
  extends CustomerParty implements AdapterObject {
    protected Customer adaptee;
    public void setAdaptee(Object o) { adaptee = (Customer)o; }
    public Object getAdaptee() { return adaptee; }
  }

  protected class HWProduct_ItemParty
  extends ItemParty implements AdapterObject {
    protected HWProduct adaptee;
    public void setAdaptee(Object o) { adaptee = (HWProduct)o; }
    public Object getAdaptee() {return adaptee;}
    protected ChargerParty charge() {
      Object o = wrap(tableTax_ChargerParty,
        adaptee.tax(), new Tax_ChargerParty());
      return (Tax_ChargerParty)o;
    }
  }

  protected class Tax_ChargerParty
  extends ChargerParty implements AdapterObject {
    protected Tax adaptee;
    public void setAdaptee(Object o) { adaptee = (Tax)o; }
    public Object getAdaptee() { return adaptee; }
    public double cost(double unitPr, ItemParty item) {
      Object o = ((AdapterObject)item).getAdaptee();
      return adaptee.taxCharge(unitPr, (HWProduct) o);
    }
  }
} //end AbstractPricingScheme adapter class
```

Fig. 7. Abstract Pricing Adapter

- *Negotiated Pricing*: A customer may have negotiated certain item prices and discounts. *Quote* plays the *LineItemParty* role. *Customer* plays the *PricerParty* role, implementing *basicPrice, discount* for negotiated pricing. *HWProduct* plays the *ItemParty* role. Finally, *Tax* plays the *ChargerParty* role.
- *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.

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 *LineItemParty*, *HWProduct* to be redefined as a subclass of both *ItemParty* and *PricerParty*, etc. Note that each of the three pricing schemes requires multiple inheritance. As Java does not support multiple inheritance, some of the mappings would be established indirectly using e.g., the static, non-composite adapter design pattern [3]. 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.

Trying to accommodate all three pricing schemes and allow dynamic switching between them, e.g., by exploiting the strategy pattern [3], results in a proliferation of classes and spurious relations in which the original design gets lost [8]. It should be noted that our main concern is that deployment of the pricing functionality with the product application requires the modification of the application classes. Reuse of the original application class model becomes impossible and the resulting tangled code is very hard to maintain. This is especially true in a real-life scenario where many business objects and processes are involved. Hence, we conclude that traditional object-oriented frameworks do not properly support component technology. The core of the problem is that there is no direct way to establish some kind of "plays-the-role-of" wiring relationship between classes in different components independently of the component implementations.

3.1 Dynamic, Non-Invasive Framework Deployment

Let us now model the same example using the dynamic composite adapter technique. Each pricing scheme will be defined in a separate adapter class, which serves to dynamically augment an *existing* quote object with a new pricing scheme by mapping each object accessible from the quote object at hand to the appropriate pricing framework roles.

Fig. 7 shows the non-invasive deployment of the application to the framework. *AbstractPricingScheme* defines that part of the deployment that is shared

entry systems developed at IBM, which is subsequently customized in different applications to customer-specific pricing schemes.

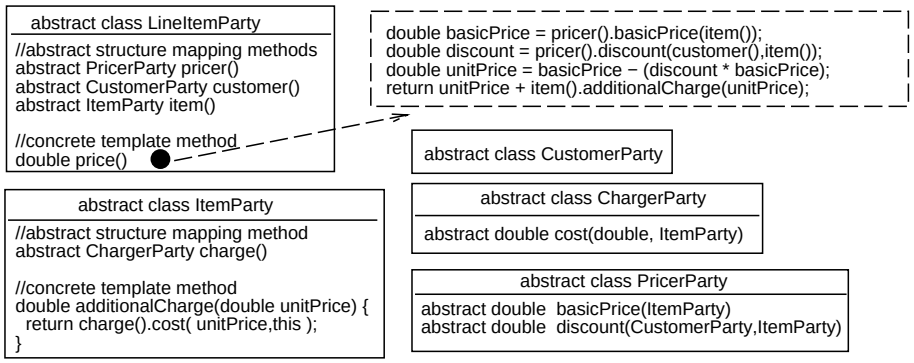


Fig. 5. Pricing Framework Class Model

Fig. 5 contains the pricing framework class model. The *LineItemParty* role is responsible for calculating the actual price of a line item purchased by a customer. The *PricerParty* role provides price and discount information for a given item and customer. The *ItemParty* role is responsible for calculating additional charges, defined in the *ChargerParty* role. Fig. 5 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.

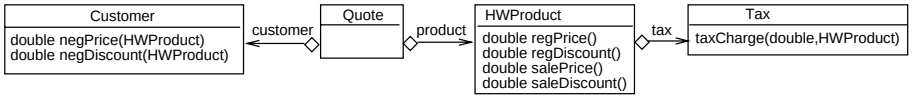


Fig. 6. Product Application Class Model

The class model of an example application domain, a product package, is shown in Fig. 6. Assume we wish to deploy the pricing 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 *LineItemParty* role. *HWProduct* plays the *ItemParty* and *PricerParty* roles, implementing *basicPrice, discount* for regular pricing. *Tax* plays the *ChargerParty* role, implementing the *cost* method. Finally, *Customer* maintains the customer role.

2.1 Succinct Specification of Dynamic Composite Adaptation

```
adapter AR_TO_FR {
  AppRoot extends FrameworkRoot {
    FrameworkChild frameworkChild() { return appChild(); }
    void primitiveop_1() { method_m(); }
    void primitiveop_2(FrameworkChild f) { method_n( (AppChild) f ); }
  }
  AppChild extends FrameworkChild {
    void primitiveop_3() { method_o(); }
  }
}
//example main program that applies the adapter to an application object
public class Client {
 static public void main(String[] args) {
  AppRoot ar = new AppRoot();
  System.out.println(((AR_to_FR)ar).templatemethod_1());
}
```

Fig. 4. Dynamic Component Adaptation - Succinct Specification

Fig. 4 defines the framework deployment using a scoping construct *adapter* for succinctly specifying dynamic composite adaptation. The *adapter* defines a set of dynamic connections between the application and framework components. Each connection represents the dynamic extension of an application class to a framework role. Note that the method bodies are simply written as if the application class was a static subclass of the framework role.

The adapter class of Fig. 2 can be generated from this specification. For each dynamic class extension, an inner class is generated based on the ternary relation between the application class (aggregation), framework role (extends), and *AdapterObject* interface (implements). A hashtable is also generated per dynamic class extension. Following the Hollywood principle of framework design (*don't call us, we'll call you*), the generated code in the primitive operations simply delegates to the application object. An application object that come into scope is automatically wrapped by an instantiation of its dynamic class extension, using the appropriate hash table. Conversely, application objects that are passed into application class methods are automatically unwrapped. Finally, the *AR_to_FR* cast in the main method in Fig. 4 is replaced by the invocation of the template method on a new adapter class instance.

3 Modeling Collaboration-Based Designs

We now demonstrate the technique with an example, namely the *Pricing* component described by Holland [4]. The component is part of a framework for order

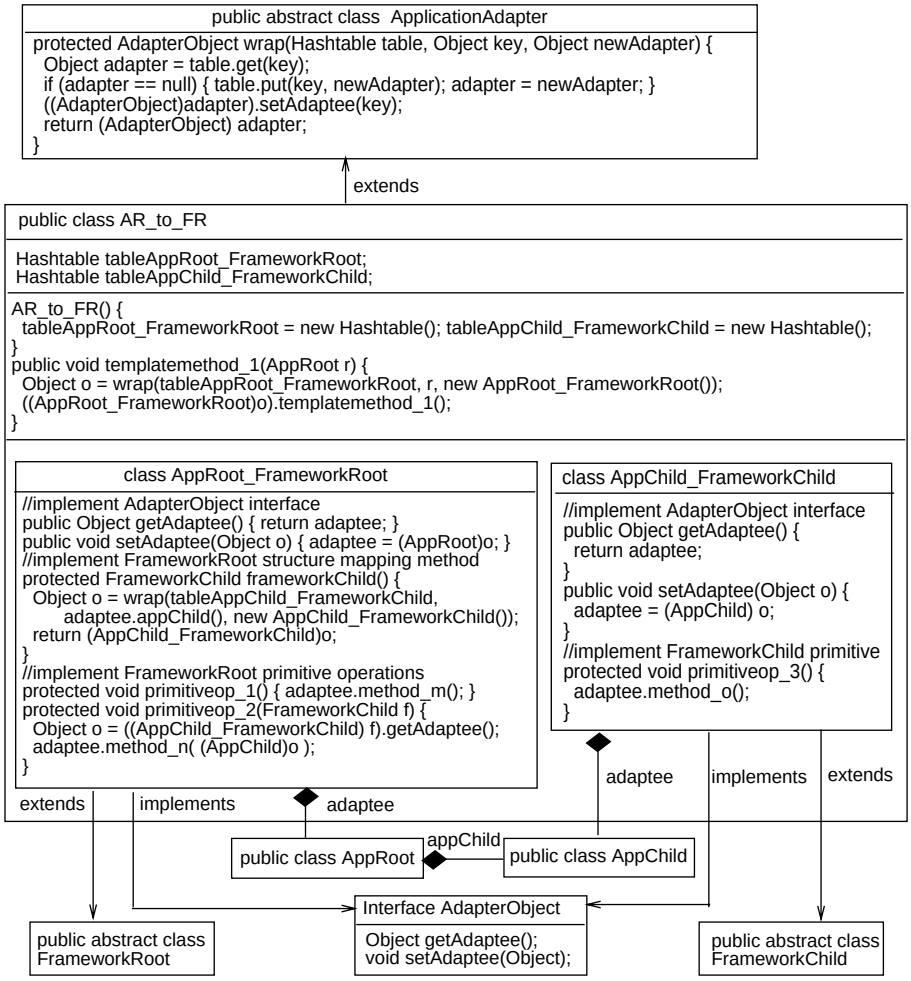


Fig. 2. Dynamic Composite Adapter - Java Inner Class Implementation

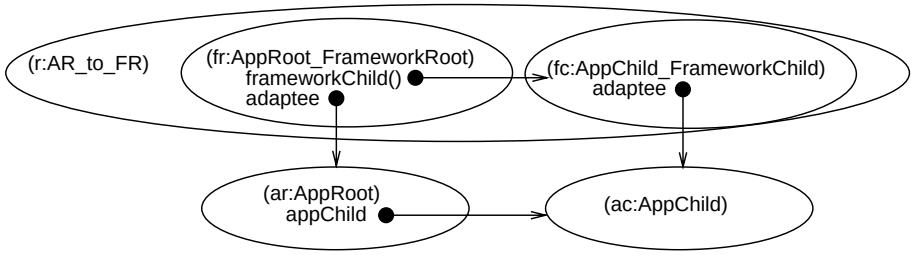


Fig. 3. Composite Object Structure

This is exactly what the *dynamic composite adapter* design pattern does. The structure of the pattern is shown in Fig. 2. The class *ApplicationAdapter* is the parent of all adapter classes, implementing a method *wrap* to maintain a hashtable for the mapping between an application object and the adapter object that wraps it. For each framework deployment, there will be one hashtable per dynamic class extension. It is important to reuse the same adapter once an application object is wrapped, as the framework may add local state to the application object. The adapter class *AR_to_FR* extends the *ApplicationAdapter* class and implements the framework deployment, defining an inner class for each framework role: *AppRoot_FrameworkRoot* and *AppChild_FrameworkChild*. Each inner class simulates the dynamic extension of the application class to implement the framework role, serving as both an *application object wrapper* and a *framework role implementor*. The inner class implements the *AdapterObject* interface (*getAdaptee, setAdaptee*) to maintain a reference to the application object that it wraps (its *adaptee*). Each inner class also implements a framework role interface by explicitly delegating to the wrapped application object.

Clients can then invoke the public template method *templatemethod_1()* on an *AR_to_FR* object, passing the root of the application composite along as a parameter. This is where the root of the application composite gets wrapped with the corresponding *AppRoot_FrameworkRoot* adapter. Each application object that comes into the scope, either through direct instantiation or as the result of invoking an application operation, is wrapped by the dynamic extension 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. The resulting structure of an *AR_to_FR* adapter object is shown in Fig. 3. Conversely, an application object is unwrapped before it can be passed into an application method, such as the parameter to *primitive_op2* that is passed to *method_n()*.

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 (e.g. the invocation of *adaptee.appChild()* within *frameworkChild()*) as well as code for mapping types from one domain to the other due to incompatible signatures (wrap/unwrap plus the necessary type casts). New adaptations can be dynamically added because adapters are implemented as separate objects.

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*, (*AP&PC*s) [7], as well as *adapters* [10].

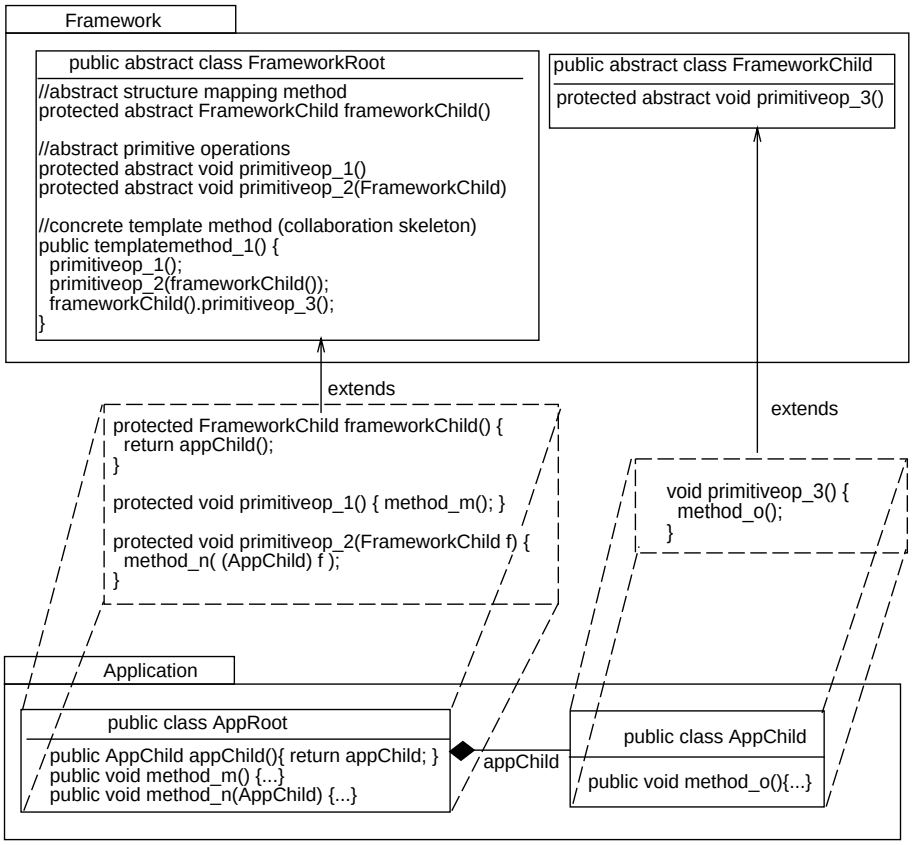


Fig. 1. Extension of Application Class Model to Framework Roles

Our goal is to utilize the framework's collaboration scheme within the application without "physically" extending the application classes. Rather, we make the application objects appear to acquire the types encoded by the dynamic class extensions of Fig. 1. That is, given the root of an application object *ar: AppRoot* on which we want to apply the framework's collaboration scheme, we need to:

1. Wrap the object *ar* with the code in the *AppRoot* dynamic class extension.
2. While executing the code defined in a dynamic class extension, application objects may come into scope as the result of either (a) invoking an application operation that returns an object, or (b) directly instantiating an application class. All such application objects will be wrapped with the dynamic extension of its class.
3. Any wrapped application object should be unwrapped before it can leave the scope of a dynamic class extension. This occurs when an application object (a) is passed as a parameter into an application operation, or (b) has an application operation invoked on it.

components; hence, it is natural to assume that the application model and the collaborative designs that model business processes are independently developed by different component vendors, and are then "glued together" at the customer site. Thus, we outline several requirements for dynamic component adaptation:

- *Framework/Application independence:* the framework and application components should be independent of each other, allowing the framework to be reused by many applications, and an application to dynamically deploy many frameworks.
- *Adapter independence:* framework deployment must be *non-invasive*, in that the existing framework and application classes need not be modified.
- *Tolerance to interface incompatibility:* an application interface may not correspond to a framework interface due to conflicts in name, argument type and cardinality. Additional control flow may be required when gluing together the application and the framework. Thus, a simple mapping among method names (parameterization) may not be sufficient.

To manage these requirements, we introduce the *dynamic composite adapter* pattern. The technique is described in general terms in Section 2 and illustrated with an example in Section 3. Related work is discussed in Section 4, followed by a brief summary of the paper in Section 5.

2 The Dynamic Composite Adapter Pattern

Consider the *Framework* package in Fig. 1, defining the abstract classes *FrameworkRoot* and *FrameworkChild*. The composite structure between parent and child is modeled through the structure mapping method *frameworkChild()* rather than through an aggregation relation. This allows 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. As the primitive operations may unexpectedly modify the structure of the application composite object, it is important that template methods always use structure mapping methods to reference the elements of the application composite. Thus, framework methods should not maintain local references to the elements of the composite structure. However, it is acceptable for a framework method to define a local variable to reference an object that is not part of the current composite object, for example an object that results from a computation.

Fig. 1 also defines an application model in the *Application* package, consisting of the composite class structure *AppRoot* and *AppChild*. Assume we wish to deploy the framework using the application, with *AppRoot* playing the *FrameworkRoot* role and *AppChild* playing the *FrameworkChild* role. The dashed enclosures in Fig. 1 represent *dynamic class extensions*. Each depicts how an application class needs to be extended to fulfill a framework role, with the abstract framework methods implemented in terms of the concrete application methods.

Inner Classes For Component-Based Programming

Linda Seiter[1], Mira Mezini[2], and Karl Lieberherr[3]

[1] College of Engineering, Santa Clara University, Santa Clara, CA, USA.
lseiter@scu.edu
[2] College of Engineering and Computer Science, University of Siegen, Germany.
mira@informatik.uni-siegen.de
[3] College of Computer Science, Northeastern University, Boston, MA, USA.
lieber@ccs.neu.edu

Abstract. Frameworks elevate encapsulation and reuse to the level of large-grained components, namely groups of collaborating classes. The abstract collaboration defined in a framework is easily customized by an application through static subclassing. However, this implies non-independent development of the application and framework models and excludes the possibility of *dynamically* deploying the framework.
We propose the *dynamic composite adapter* design pattern, which employs the use of Java inner classes to achieve dynamic, non-invasive, composite object customization. We also present a new scoping construct for succinctly defining dynamic adaptation of Java components.

1 Introduction

Component software is becoming an increasingly popular choice for system development, its goal being the development of highly reusable, customizable software components. In this paper we focus on a type of component referred to as *collaboration-based* design [7, 9]. A collaboration may be viewed as a slice of a class model, specifying the structural and behavioral relations required to accomplish a specific task. A collaboration can be implemented as a white-box framework, i.e. a set of abstract classes (representing roles in the collaboration) along with their structural relationships, a set of abstract primitive operations, and a set of concrete template methods that define the collaboration skeleton by invoking the primitive operations. The abstract model of a framework is easily customized through static subclassing. This works well when the assignment of framework roles to application classes is statically fixed prior to runtime. However, it is often desirable for an application to dynamically customize a framework, especially with Java's runtime architecture in which classes are dynamically loaded. A running application may wish to apply a newly loaded collaboration scheme to a set of previously loaded classes. Static subclassing will not support such customization. Even if the customization is known to be static, it is not feasible to require that the application be "written" to the framework. Component-oriented programming emphasizes the gluing of pre-existing binary