Loosely Coupled Class Families

Topic: Characteristics and Design Decisions of ASoC models

Erik Ernst

Abstract

Families of mutually dependent classes that may be accessed polymorphically provide an advanced tool for separation of concerns, in that it enables client code to use a group of instances of related classes safely without depending on the exact classes involved. However, class families which are expressed using virtual classes seem to be very tightly coupled internally. While clients have achieved the freedom to dynamically use one or the other family, it seems that any given family contains a fixed set of classes and we will need to create an entire family of its own just in order to replace one of the members with another class. This paper shows how to express class families in such a manner that the classes in these families can be used in many different combinations, still enabling family polymorphism and ensuring type safety.

1 Introduction

In [4], the notion of family polymorphism is presented. Family polymorphism is a phenomenon that arises when virtual classes are available as attributes of objects, and it enables the type safe usage of groups of objects in a subtype polymorphic setting—similar to the subtype polymorphism which is a hallmark of object-orientation in general, but applied to a group of objects and not just a single object.

Traditional polymorphism supports the separation of client code from knowledge about variant or implementation of the polymorphically accessed object. A similar separation of client and server concerns breaks down when several objects which are instances of mutually dependent classes are to be used together. As explained in [4], we must either give up type safety or polymorphic access, if only traditional polymorphism (including parametric polymorphism) is available. Family polymorphism allows us to separate client code from knowledge about the exact classes of the group of objects and still ensures type safety.

However, this is done by means of virtual classes, and a group of mutually dependent virtual classes are tightly bound to each other. This makes it impossible to modify a class family by using most members from one family and then, e.g., one member of another class family. Virtual classes from different families

are type incompatible unless they are known exactly—i.e., unless they are used in such a way that they are reduced to ordinary, statically known classes (and they may of course be type incompatible even then).

This paper presents a way to express more loosely coupled class families. Section 2 presents the traditional solution using virtual classes. The more flexible version is presented in Sect. 3. Finally, Sect. 4 concludes.

2 Tightly Coupled Families

The observer design pattern [5] has been used a number of times [9, 1, 6, 3] to illustrate the challenges of working with mutually recursive classes, and in particular the challenges of creating such groups of classes incrementally and using them in a type safe manner.

Class families built by means of family polymorphism is one possible approach, and it excels in the area of polymorphic decoupling between the client and the class family. However, as it is argued in [8], the members of each class family are very intimately dependent on each other, and this makes it hard to create ad hoc families by mixing and matching classes.

In this section we present a traditional expression of the observer design pattern using virtual classes. It is inspired by the expression given in [8], but simplified because of space constraints.

```
Init: (# init:< (# do INNER exit this(Init)[] #) exit this()[] #);
SubObs:
  (# Subject:< Init
       (# obs: @list(# element::Observer #);
          addObserver: (# enter obs.append #);
          notify:<
            (# do obs.scan(# do this(Subject)->current.onNotify #)#)
       #);
     Observer:< Init
       (# onNotify:< (# s: ^Subject enter s[] do INNER #)#)
  #);
WinMan: SubObs
  (# Window: Subject; Manager: Observer;
     Subject::<
       (# move:< (# d: @integer enter d do pos+d->pos; INNER; notify #);
          draw:< (# do '\nPos='->puttext; pos->putint; INNER #);
          pos: @integer
       #);
     Observer::< (# onNotify::< (# do s.draw; INNER #)#)
  #)
```

Box 1

Box 1 shows the observer design pattern class family, expressed in the language gbeta [2, 3, 4], which is a generalized version of BETA [7]. It is our hope that this design pattern is so well-known that the BETA-style syntax will be readable even by people who do not know BETA or gbeta, but let us first give

a few basic directions as to how the syntax should be understood. Declarations are on the form Id : Mark Exp where Id is the name being declared, the colon distinguishes declarations, Mark is one or two non-letter characters specifying the kind of entity being declared, and Exp is an expression specifying how that entity should be created or looked up. For example, `pt: @Point` declares the name `pt` to be an object ("`@`" means object) which is an instance of the class `Point`. Similarly, `v:< Point` declares the name `v` to be a virtual class ("`<`" means virtual) whose value is the class `Point` or a subclass; virtuals can be extended using the marker `:<`, as in `v::< ColorPoint`.

Box 1 shows the basic family, `SubObs`, containing virtual, generic `Subject` and `Observer` classes, along with the necessary machinery for distribution of notifications. Briefly, whenever an instance of the virtual class `Subject` changes state it should invoke the method `notify`, thus enabling all its observers to learn about the change. Each `Subject` keeps its observers in a list, `obs`, and the members of this list will then be notified because `notify` iterates through `obs` and invokes `onNotify` on each element in the list (the `scan` method on `list` implements "scanning", or iterating through, the list). It is then up to the `Observer` to react in some way to the changes that occurred in the `Subject`. The `onNotify` method in `Observer` in `SubObs` just executes `do INNER`; this means that the method does nothing, but subclasses may add behavior as needed.

Box 1 also shows a subfamily of `SubObs`, `WinMan`. In this family we first create two aliases—`Window` which is the same as `Subject`, and `Manager` which is the same as `Observer`. Then `Subject` is furtherbound (virtually-extended) into a window class—crudely illustrated by adding a `move` and a `draw` method. Similarly, `Observer` is furtherbound into a window manager class, capable of using the extended `Subject` class. The idea is that each window has a number of managers, taking care of various decorations or interactions, and every manager needs to be notified when something happens to one of its windows. The `Manager` will actually simply invoke `s.draw` upon notification (i.e., in its furtherbinding of the method `onNotify`), thereby "redrawing" the `Window` `s`.

Finally, box 1 shows the auxiliary class `Init` which is used to equip other classes with an `init` initialization method, and to allow an expression denoting an object to evaluate to a reference to that object (in gbeta and BETA, evaluation semantics must be specified explicitly).

We may now use these class families as follows:

```
(# f: @WinMan; w: ^f.Window                    (* declarations *)
do f.Manager->(&w).init.addObserver;           (* statement *)
   1->w.move                                   (* statement *)
#)
```

Box 2

This declares a class family `f` and a reference `w` to a `Window` in that family. Then it creates a `Window`, makes `w` refer to that `Window`, and initializes it (`(&w).init`). It creates a `Manager` in `f`, and adds that `Manager` as an observer to the window (`..Manager->..addObserver`). Finally, it moves `w` by `1` (`1->w.move`), whereby the `Manager` gets notified and invokes `draw` on `w`, which prints `Pos=1`.

This works, and it is useful. But the recursive structure of a class family is rather closed. It seems to be hard to use several different kinds of observers in such a family—if we want to furtherbind `Observer` in two different ways then we end up having two different class families. As we shall see in the next section, there are ways out of this dilemma.

3 Decoupling

In order to create class families with a more flexible inner structure we do not aim to allow ad-hoc combinations of arbitrary classes. The freedom of dynamic polymorphism together with the safety of static type checking is achieved in family polymorphism because of the support for networks of virtual classes referring to each other.

What we can do, however, is to create ordinary non-virtual classes *containing usages of virtual classes*. This lets us create hierarchies of ordinary, statically known classes, except that each class family will have its own copy of these static[1] hierarchies, and for each class family the static class hierarchy will be implicitly *customized*, in the sense that its usages of virtual classes will be associated with the virtuals in exactly that family. Let us consider an example:

```
SubObsD:
  (# <<SLOT solib:attributes>>;
     Subject:< Init
       (# obs: @list(# element::Observer #);
          addObserver: (# enter obs.append #);
          notify:<
            (# do obs.scan(# do this(Subject)[]->current.onNotify #)#)
       #);
     Observer:< BaseObserver;
     BaseObserver: Init
       (# onNotify:< (# s:^Subject enter s[] do INNER #)#)
  #);
WinObsD: SubObsD
  (# <<SLOT wolib:attributes>>;
     Subject::<
       (# move:<(# d: @integer enter d do pos+d->pos; INNER; notify #);
          draw:<(# do '\nPos='->puttext; pos->putint; INNER #);
          pos: @integer
       #)
  #)
```

Box 3

In box 3 we have an alternative expression of the class family, `SubObsD`, and a subfamily `WinObsD` ("D" for decoupled). The explanation of the functionality in box 1 applies unchanged to box 3, except that `WinObsD` only furtherbinds

[1] Note that the word "static" in this paper refers to static knowledge and has nothing to do with the "per-class" meaning of the word in connection with declarations in, e.g., Java

`Subject` (where `WinMan` in box 1 gave furtherbindings for both `Subject` and `Observer`).

The important difference is that the virtual class `Observer` in box 3 is defined to be the static class `BaseObserver`. We may furtherbind `Observer` in subclasses of `SubObsD` as usual, but now we also have a name which denotes the statically known value of the virtual class `Observer` in `SubObsD`, and that allows us to create an ordinary static class hierarchy, rooted in `BaseObserver`.

In the definition of `BaseObserver`, it is crucial that the argument `s` of `onNotify` is typed as a `Subject` and not, say, as some static `BaseSubject` class that we might introduce in a similar role as `BaseObserver`. Generally, the static class hierarchies should be written in such a way that each static class refers to the virtual classes as its peers, not other static classes. Only then will the static hierarchies be implicitly customized in such a way that they can be used in subfamilies. Note that this customization is an inherent property of virtual types, and not e.g. some hidden preprocessing mechanism.

We have placed two "`SLOT`" applications named `solib` and `wolib` in the class family. The effect of this is that we may add attributes to the families externally, e.g., in other files. This again makes it possible for us to extend the abovementioned static class hierarchies in an open-ended manner. Here are some additions:

```
-- solib:attributes --
TraceObserver: BaseObserver
  (# onNotify::< ! (# do '\nNotify!'->puttext; INNER #)#)

-- wolib:attributes --
Window: Subject;
ManagerObserver: BaseObserver
  (# onNotify::< (# do s.draw; INNER #)#)
```

Box 4

Box 4 defines `TraceObserver` in `solib`, i.e., in context of `SubObsD`. This is a subclass of `BaseObserver` that reacts on all notifications by printing a message, thus making it possible to watch notifications as they are received. The `INNER` statement will then invoke other contributions to `onNotify`.

Moreover, box 4 defines the usual alias `Window` as well as the static class `ManagerObserver` in `wolib`, i.e., in context of `WinObsD`. We could not have defined `ManagerObserver` in context of `SubObsD`, because it uses the `draw` method of an object typed as a `Subject`, and it is only in `WinObsD` that a `Subject` is guaranteed to have such a method. We may now use all the different kinds of observers together:

```
   (# f: @WinObsD; w: ^f.Window
   do f.BaseObserver->(&w).init.addObserver;
      f.TraceObserver->w.addObserver;
      f.ManagerObserver->w.addObserver;
      1->sub.move
   #)
```

Box 5

This example is similar to the example in box 2, but this time we add three different observers to the window, namely a `BaseObserver` which will do nothing upon notification, a `TraceObserver` which will print a message when notified, and a `ManagerObserver` which will invoke `draw` on its `Subject`.

As long as it is known statically that we are creating observers in a family that is an instance of `WinObsD`, there is no problem in verifying statically that an `f.TraceObserver` is really a subclass of an `f.Observer`. However, if we are using a class family polymorphically, we can still use the static class hierarchies, only now as mixin classes. Assume that `f` denotes a class family which is an instance of some (not statically known) subpattern of `SubObsD`. Then we can do the following:

```
(# (* given: f is an instance of some subclass of SubObsD *)
   w: ^f.Window
do f.TraceObserver & f.Observer -> (&w).init.addObserver;
   1->sub.move
#)
```

Box 6

The expression `f.TraceObserver&f.Observer` will create an instance of a class which is created by combining the classes `TraceObserver` and `Observer` in `f`. That instance is known statically to be an acceptable argument to the method `w.addObserver`, because it is an instance of some subclass of `f.Observer`, and `w.addObserver` expects an argument which is an instance of `f.Observer` or a subclass.

Such combinations will generally be useful since the class combination mechanism is propagating [2], which among other things means that the behavior of a method of the combined class will be a combination of the behaviors of that method in the classes being combined. E.g., `onNotify` will in this case generate output like a `TraceObserver`, and it will also perform all the actions that `onNotify` of an `f.Observer` does.

In other words, we may use the static class hierarchies in class families as toolboxes for direct use in the static case, and as mixin classes in the polymorphic case, thereby providing and safely using a wealth of variants of each member of a class family.

4 Conclusion

This paper described the approach to creation and management of mutually recursive classes based on virtual classes. It was noted that this approach yields separation of client code concerns from class family concerns, but it seems that this separation is achieved in return for a tight coupling between the members of any given class family. Since virtual classes have an existential type, an inspection of their (statically known) structure is not a sufficient or a sound basis for determining type equivalence. Hence, we cannot freely mix and match classes which are members of different class families, and it seems that we must

create a multitude of families, just to be able to use various selections of family member variants.

However, a technique was presented that allows us to create, inherit, and openly extend static class hierarchies in context of a hierarchy of class families. The crucial idea is to write each static class in such a way that it uses as its peers the virtual classes that constitute the class family, as opposed to using other static classes. These static class hierarchies can then deliver concrete variants and extensions to the virtual classes at any given level, and they may be used to create, e.g., many different kinds of `Observer`s in a `Subject/Observer` class family.

References

[1] K. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. *Lecture Notes in Computer Science*, 1445:523–549, 1998.

[2] Erik Ernst. *gbeta– a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, DEVISE, Department of Computer Science, University of Århus, Denmark, 1999.

[3] Erik Ernst. Propagating class and method combination. In Rachid Guerraoui, editor, *Proceedings ECOOP'99*, LNCS 1628, pages 67–91, Lisboa, Portugal, June 1999. Springer-Verlag.

[4] Erik Ernst. Family polymorphism. In *Proceedings ECOOP 2001*, LNCS (no. not yet known), page (page numbers not yet known), Budapest, June 2001. Springer-Verlag.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[6] Vassily Litvinov. Constraint-based polymorphism in cecil: Towards a practical and static type system. In Craig Chambers, editor, *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98), Special Issue of SIGPLAN Notices*, volume 33, 10, Vancouver, October 1998. ACM Press.

[7] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, MA, USA, 1993.

[8] Didier Rémy and Jérôme Vouillon. On the (un)reality of virtual types. Work in progress, available from `http://pauillac.inria.fr/~remy/`, 2001.

[9] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings ECOOP'97*, LNCS 1241, pages 444–471, Jyväskylä, June 1997. Springer-Verlag.