6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

A generative approach for family polymorphism in C++

István Zólyomi, Zoltán Porkoláb

Department of Computer Science,
Eötvös Loránd University of Sciences
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
e-mail: {scamel, gsd}@inf.elte.hu

Abstract

The object-oriented paradigm provides safe and flexible use of objects of classes that can be arranged to inheritance hierarchies. Late binding ensures that we use the appropriate function body when we call a method on an actual object via polymorphic references. In the same time we have compile-time guarantees to use only valid calls.

The problem arises when we use two or more independent hierarchies of classes together. In this case the collaborating "families" may consist of similar but not interchangeable classes. Because there can be subtype relationship between classes in the different groups, it is not obvious to implement a constraint ensuring that only classes of the same family are used together. Traditional object-oriented languages are not able to handle this situation. Proposed solutions vary from run-time assertions to family polymorphism extensions of existing programming languages. Family polymorphism – strongly investigated by Erik Ernst and others – takes traditional polymorphism to the multi-object level.

In this paper we present a generative way to express such constraints in the C++ language using templates. We use only standard C++ language features in our implementation.

Categories and Subject Descriptors:
D.1.5 [Programming Techniques]: Object-oriented Programming;
D.1.m [Programming Techniques]: Miscellaneous

Key Words and Phrases: Family polymorphism, Generative programming

1. Introduction

Without doubt the most popular design and programming methodology of our days is object-oriented programming. We identify objects of the system, describe

similar objects with classes and organize an inheritance hierarchy of the classes expressing subtype relationships. Inclusion polymorphism and late binding of methods ensure execution of appropriate overriding methods corresponding to the dynamic type of the object. These features provide flexibility of design and their language support must be responsible for the safety of implementation.

Family polymorphism is a natural extension of conventional polymorphism: it takes polymorphism to the multi-object level. We bind several classes into a family, and provide polymorphism *between* such families. In the same time, classes in a family must be used together in any context, mixture of classes from different families is considered to be an error. This facility has language support in a few rarely used languages only, e.g. gbeta.

For easier understanding, we show an example for the problem, which we use consequently throughout the rest of the article. It is related to animals: imagine having a zoo with all kinds of common or exotic animals (horses, lions, penguins, or camels). Using the common design construct, all animals should share a common base class Animal (see figure 1). Because animals need to be fed sometimes, we have class Food and a member function of Animal which is eat(Food&). Different animal classes, like Horse and Lion, inherit from base class Animal. On the other side, we specialize several possible food types like Grass or Meat[1]. The problem arises with the Animal::eat() function: different animals have to be fed differently. Lions eat only meat, while horses eat grass.

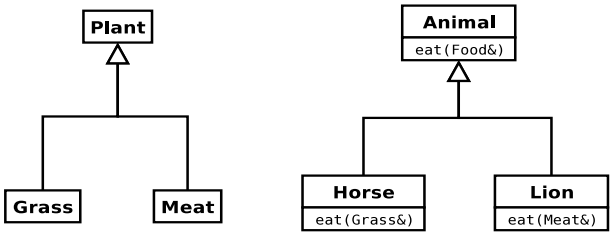

Figure 1: Class hierarchy for the Animal example

Different food classes are not freely interchangeable: a horse would starve to death if it was fed with meat; a lion would do the same near a bunch of flowers. In widely used object-oriented languages, we cannot express such constraints: it is a limitation of the object-oriented paradigm.

Note that this example is not too artificial: Almost the same problem and class hierarchies can be found taking a closer look at event based frameworks (e.g. java.awt): all you have to do is to substitute class Food with Event and Animal with some EventHandler class. These frameworks are based on different event families and event handler components which accept only a restricted set of events. For

[1]Let us now ignore the fact that meat consists of parts of an animal, what could be modeled deriving Animal from Food.

example, a MouseEventHandler class cannot process keyboard events, thus we have the very same situation as with animal and food types.

We can see the lack of sophisticated solutions for the problem in these frameworks. In jawa.awt, the solution is to simply avoid defining a common event handling function in the event handler base class.

In the rest of the paper, we discuss the possibilities of solving the family polymorphism problem using features of different, widely used object-oriented languages. Our goal is to provide compile time guarantees for the forementioned situations, similarly to checking types for conventional polymorphism. Weakly typed script languages are not considered in the article, because they provide runtime errors. However, we are not restricted to object-oriented features only: we also keep the tools of generative programming in mind: we analyse multiparadigm methods to solve the problem. Because all of the object-oriented and generative features we will discuss are present only in C++, we concentrate on that language.

2. Dynamic verification

A basic pillar for object orientation is the use of virtual functions. However, they do not provide a solution in our case: we have to narrow the parameter types of our eat() function, what is not allowed when using dynamic binding for the function.

Hence a conventional object-oriented solution may be the verification of the dynamic type of parameters objects. Most languages have support for this kind of verifications, e.g. instanceof in Java or dynamic_cast in C++. If the dynamic type of the parameter is inappropriate, we may raise an error after the type check. A possible solution may look like the following in C++:

```
// --- General family
struct Food { ... };
struct Animal { virtual void eat(Food&); };

// --- Carnivore family, herbivores are implemented similarly
struct Meat: public Food { ... };
struct Lion: public Animal {
    void eat(Food& f) {
        if (! dynamic_cast<Meat*>(&f) ) {
            // --- RAISE ERROR
        }
    }
};
```

Firstly we have base classes Food and Animal. Later, there are two different inheritance branches created from these bases. One is the carnivore family, with class Lion and its food: class Meat. The other family is of herbivores, with class

Horse and its appropriate food class grass. (Let us now ignore the fact that lions do not eat *all kind* of animals and horses do not eat *everything* that is green: in real life, such further contraints usually must be introduced to express exact conditions). When an animal is not fed with its appropriate food type, an error is raised when checking the parameter type dynamically.

The use of these classes may look like the following in our program:

```
// --- Animals and their food
Lion Simba;      /*eats*/ Meat meat;
Horse Shadowfax; /*eats*/ Grass green;

// --- Proper calls
Shadowfax.eat(green);   Simba.eat(meat);

// --- Calls resulting runtime errors
Simba.eat(green);       Shadowfax.eat(meat);
```

Though the method above is the conventional object-oriented solution, it has several serious drawbacks. One is the cost of verifications done during runtime: using a number of dynamic checks may need a noticable amount of work. The lack of safety in our program can be considered much more seriuosly in most cases. For unaccepted parameters this solution raises runtime errors instead of compile time errors, which are preferred whenever possible. Generally, compile time checks largely improve the safety of the implementation. They have two great advantages: we can avoid surprising program termination in cases of unexpected runtime properties and may also avoid many error handling cases, what can be overwhelming when we rely on runtime errors. Realization of static verification leads us to usage of generative tools.

3. Generative Programming

The generative programming paradigm is a further extension of the traditional object oriented methodology. Besides conventional object oriented tools, it provides many additional features that enable creation of more general components and a more automated development process for applications. Generative programming is a collection of several subparadigms, it includes several different approaches.

Aspect oriented programming is very much like a "programmable debugger": it can trigger arbitrary actions to specific program events, e.g. function calls or variable access. Generic programming allows creation of more general programming components by parameterizing classes with element types and algorithmic strategies. Template metaprogramming transfers certain kinds of calculations from runtime to compile time, thus making the running program more effective.

Since our goal is to allow compile time inspection of combined classes, we do not further investigate aspect oriented programming (it provides mostly runtime tools): we concentrate on the latter two subparadigms of generative programming.

3.1. Traits

The main goal of generic programming is to create the most general components possible that later can be configured to conform to any required behavior. This can be realized using so-called generics or templates which enable compile time type parameters of program elements besides traditional runtime parameter objects. From now on, we use the term templates corresponding to C++ language conventions. Templates lead us to another form of polymorphism, which is called parametric polymorphism: it means the possibility of parameterizing a class with type parameters. Thus vector<int> and vector<string> are two different polymorphic forms of the same class vector.

A common generative solution for a wide scale of problems is the use of type traits. Traits are a special kind of type parameters for a class: they make it possible to parameterize member data construction or algorithmic strategies of a class. E.g. for the class std::basic_string from the standard library of C++, we can set the character type for a string (member data type), and specify comparision or ordering between such characters (algorithmic strategies). In our case, we intend to specify the types accepted by member function eat(). We separate this property in a trait parameter of class Animal, which we use later at the definition of function eat().

```
struct Food { ... };
struct Grass: public Food { ... };
struct Meat: public Food { ... };

struct HorseTraits { typedef Grass FoodType; };
struct LionTraits { typedef Meat FoodType; };

template <class Traits> struct Animal {
    void eat(typename Traits::FoodType f) { ... }
};

typedef Animal<LionTraits> Lion;
typedef Animal<HorseTraits> Horse;
```

In the example above, we do not create an inheritance hierarchy of animals, we used templates instead: we ignore inclusion polymorphism and choose parametric polymorphism instead.

Type Traits is supposed to have a nested type FoodType. We utilize this nested type at the definition of the food parameter type of function eat(). This construction results in compile time errors for unaccepted parameters (we used the same naming conventions for our animal and food objects as before):

```
// --- Proper calls
Simba.eat(meat);    Shadowfax.eat(green);
```

```
// --- Calls resulting compile errors
Simba.eat(green);   Shadowfax.eat(meat);
```

Unfortunately, this approach still has serious limitations. One can be seen at the definition of class LionTraits: since we have to define the exact type of the food (it cannot be a template still unparameterized), we lose some flexibility. Considering the case having no class Meat, but having class Animal derived from Food instead (hence modelling that lions simply eat other animals), we must specify an exact animal type as the food of lions (e.g. Animal<HorseTraits>), therefore lions may eat only a special kind of animal instead of a wide scale of animals.

More seriously, as a result of parametric polymorphism, we lose the possibility to handle all animals through a common interface, e.g. an abstract base class.

3.2. Template metaprogramming

As we could see, generic programming tools provide solutions with undesired side effects and limitations, thus we change to the template metaprogramming paradigm. This paradigm is based on the possibility of specializing templates for certain cases (e.g. specific types for type parameters or concrete values for integer parameters). The implementation of a specialized template can be completely different from the general one, therefore we have no restrictions on writing specializations. This enables a strange programming approach: instantiating template specializations according to recent calculation results, we can run algorithms inside the compiler during compilation time. If a calculation is done during compilation, we can simply store its result in an initialized program variable, thus we need no runtime computations afterwards. This way we can make our programs more efficient.

Let us clarify this with a simple example calculating the factorial of an integer. It can be implemented as follows:

```
// --- General recursive case for factorial template
template <int i> struct Fact {
    // --- Store result as nested enum to save memory
    enum { Result = i * Fact<i-1>::Result };
};

// --- Specialization for zero
template <> struct Fact<0> {
    // --- Stop recursion providing constant result
    enum { Result = 1 };
};

// --- Compute result in compile time
int num = Fact<7>::Result;
```

3.3. Type maps

Based on template metaprogramming, we can further improve our previous solution using a metaprogramming construction: type maps. Instead of traits, we can try to create a collection of allowed pairs: we create maps of types. Note that type maps are similar to conventional maps, but instead of data, they contain types. The key of a map is the type to which the function call is delegated (e.g. Animal with its eat() function), while the value is a collection of classes marking the required parameter types of the function call (e.g. class Meat for key Lion). We have similar basic classes as before, but to show the expressive power of this solution, we also create an omnivore class Monkey:

```
struct Food { ... };
struct Grass: public Food { ... };
struct Meat: public Food { ... };

struct Animal { virtual void eat(Food&) {}; };

struct Lion: public Animal { ... };
struct Horse: public Animal { ... };
struct Monkey: public Animal { ... };
```

Now we construct our type map. By default, if no specific rule is given to an animal, it is omnivore (it can eat anything), what is expressed using a dummy food class Any. Based on this class, we can create our TypeMap as follows:

```
// --- Dummy class meaning any food
struct Any {};

// --- General case: animals are omnivore by default
template <class> struct TypeMap { typedef Any Type; };

// --- Specializations for certain animal types
template <> struct TypeMap<Lion> { typedef Meat Type; };
template <> struct TypeMap<Horse> { typedef Grass Type; };
```

While animals (like monkeys) are considered to be omnivores by default, there are two specializations of TypeMap for our two classes having special food consumption behavior: lions and horses. For them, we define the specific kind of food in the type map that they can eat.

So far we have defined our families, but still did not considered the check itself. We have a separate template function checkedEat()[2] to do the job:

[2]Note that function checkEat() could be implemented as a member function of Animal or any of its subtypes, what we avoided for easier understanding of our examples. For member functions, we could use a more natural form of function calls as Simba.checkedEat(meat);

```
template <class AnimalType, class FoodType>
void checkedEat (AnimalType &a, FoodType& f)
{
    // --- Get allowed food type for animal (may be Any)
    typedef typename TypeMap<AnimalType>::Type AllowedFood;

    static const bool valid =
        // --- Food argument matches specific rule
        SUPERSUBCLASS(AllowedFood, FoodType) ||
        // --- OR there is no special rule (Any)
        SUPERSUBCLASS(AllowedFood, Any);

    STATIC_CHECK(valid, ANIMAL_IS_FED_WITH_INAPPROPRIATE_FOOD);

    a.eat(f);
}
```

First of all, it has more than one type parameter: it has a parameter for every function argument and an additional one for the delegated object. Real magic lies in the body of the function: it uses macros on templates from library Loki[3]. For space considerations, we do not give a detailed explanation of these macros, we concentrate on their usage in this article.

Macro SUPERSUBCLASS is used to check the inheritance relation between two classes: its parameters are the suspected super and subtype respectively. The result is a boolean value marking if the first type is really a supertype for the second.

Macro STATIC_CHECK provides compile time error messages: if the condition given as its first parameter is false, a compile error is created using the custom error message specified as the second argument.

The implementation of function checkedEat() is much more simple than it seems. Variable valid contains the result of the type verification: it is true if the animal is actually fed with its specialized food (first macro call) *or* there is no special rule on its food (second call). Based on this result, a static assertion is called to raise an error message for unadequate arguments. The function call is finally delegated to the animal object.

After defining this function, we can use function calls in the following form:

```
// --- Proper calls
checkedEat(Simba, meat);   checkedEat(Shadowfax, green);

// --- Compile errors
checkedEat(Simba, green);  checkedEat(Shadowfax, meat);
```

Note that we do not have to (though we may) explicitly specify the template (type) parameters of our function, they are deducted automatically by the compiler.

[3]For a detailed explaination of library features, see [4]).

This solution has many advantages. Most importantly, it is non-intrusive. Type constraints are separately defined, for every new animal class, we need only a new type map specialization "entry"what does not interferes with any existing code. Contrary to the solution with traits, we can have subtype relationship between appropriate classes (e.g. Food and Grass). Furthermore, we are able to express more complex constraints on our types, e.g. introducing omnivore animals. This is a result of using arbitrary logical operations writing our conditions, e.g. valid is a result of connecting two conditions with a *logical or*.

As a logical consequence of using compile time features, this solution still has a drawback: the *dynamic type* of parameters cannot be checked in compile time. We are able to inspect only the static type of function parameters, what is insufficient having polymorphic parameter objects. In these cases, the only possible way is to have dynamic checks in the code, what we intended to avoid.

References

[1] Ernst, E.: Family Polymorphism. In Proceedings ECOOP 2001, LNCS 2072, pages 303–326, Budapest, Hungary 2001.

[2] Kim B. Bruce: Foundations of Object-Oriented Languages. The MIT Press, Cambridge, Massachusetts (2002)

[3] Bjarne Stroustrup: The C++ Programming Language Special Edition. Addison-Wesley (2000)

[4] Andrei Alexandrescu: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)

[5] Herb Sutter: Exceptional C++. Addison-Wesley (2000)

[6] Carlo Pescio: Multiple Dispatch: A new approach using templates and RTTI. in C++ Report, June 1998.