

Coding Patterns

See also: Chapter 25

Coding Patterns

CS193D, 3/13/06

What is a Coding Pattern?

A coding pattern is a reusable language specific technique to solve a common problem.

C++ Coding Patterns we've already covered:

- Smart Pointers
- Catching and rethrowing exceptions
- File reading loops

Double Dispatch

The Problem: Virtual methods in C++ only let you perform polymorphism in one dimension:

```
class Animal
{
    public:
        virtual void move() = 0;
};

class Kangaroo : public Animal
{
    public
        virtual void move() {
            cout << "hop!" << endl;
        }
};
```

Goal: Create a method called `eats()`, which returns true if the animal is a predator of the argument animal:

```
bool Animal::eats(const Animal& inAnimal)
```

Two factors for making the decision:

- The type of the animal doing the eating
- The type of the animal being eaten

A virtual method can only solve one of these decisions! C++ does not support *multi-methods*.

Attempt #1: Brute Force

```
class Animal
{
    public:
        virtual bool eats(const Animal& inPrey) const = 0;
};

bool Bear::eats(const Animal& inPrey) const
{
    if (typeid(inPrey) == typeid(Bear&)) {
        return false;
    } else if (typeid(inPrey) == typeid(Fish&)) {
        return true;
    } else if (typeid(inPrey) == typeid(Dinosaur&)) {
        return false;
    }
    return false;
}
```

```

bool Fish::eats(const Animal& inPrey) const
{
    if (typeid(inPrey) == typeid(Bear&)) {
        return false;
    } else if (typeid(inPrey) == typeid(Fish&)) {
        return true;
    } else if (typeid(inPrey) == typeid(Dinosaur&)) {
        return false;
    }
    return false;
}

bool Dinosaur::eats(const Animal& inPrey) const
{
    return true;
}

```

Each subclass is responsible for determining the return value of eats() by examining the typeid of the input argument.

Evaluating the Brute Force Approach

On the plus side:

- It's pretty straight-forward
- It uses virtual methods just as we always have

However...

- We have to query the type of an object at runtime manually
- Subclasses must reimplement `eats()`, or express in terms of the parent
- The code is repetitive (imagine 100 *Animal* subclasses)
- It doesn't force the implementer to be exhaustive

The brute force approach may be appropriate for a small number of classes, but isn't a good general solution.

Attempt #2: Single Polymorphism with Overloading

What if we try to have different versions of `eats()` for each parameter type? The compiler *should* call the right one, which can simply return true or false:

```
class Animal {
    public:
        virtual bool eats(const Bear& inPrey) const = 0;
        virtual bool eats(const Fish& inPrey) const = 0;
        virtual bool eats(const Dinosaur& inPrey) const = 0;
};

class Bear : public Animal {
    public:
        bool eats(const Bear& inBear) const { return false; }
        bool eats(const Fish& inFish) const { return true; }
        bool eats(const Dinosaur& d) const { return false; }
};
```

Why Attempt #2 Won't Always Work

Overloaded method binding happens at compile time. So this *will* work:

```
Bear myBear;  
Fish myFish;  
cout << myBear.eats(myFish) << endl;
```

And this *will* work:

```
Animal& animalRef = myBear;  
cout << animalRef.eats(myFish) << endl;
```

But this *won't* work:

```
Animal& animalRef = myFish;  
cout << myBear.eats(animalRef) << endl; // BUG! No such  
method Bear::eats(Animal&)
```

Attempt #3: Double Dispatch

In attempt #2, we were trying to do polymorphic overloading, which doesn't work in C++. Instead, let's use real polymorphism to determine the class on which to call a method, not the version of a method to call.

```
bool Bear::eats(const Animal& inPrey) const
{
    return inPrey.eatenBy(*this);
}
```

By calling a virtual method on the input argument, we can invoke polymorphism twice:

- 1) Determine who is doing the eating (polymorphic eats())
- 2) Determine who is being eating (inPrety.eatenBy())

Double Dispatch Class Definitions

```
class Fish;
class Bear;
class Dinosaur;
class Animal {
    public:
        virtual bool eats(const Animal& inPrey) const = 0;
        virtual bool eatenBy(const Bear& inBear) const = 0;
        virtual bool eatenBy(const Fish& inFish) const = 0;
        virtual bool eatenBy(const Dinosaur& d) const = 0;
};

class Bear : public Animal {
    public:
        virtual bool eats(const Animal& inPrey) const;
        virtual bool eatenBy(const Bear& inBear) const;
        virtual bool eatenBy(const Fish& inFish) const;
        virtual bool eatenBy(const Dinosaur& d) const;
};
```

```
class Fish : public Animal {
    public:
        virtual bool eats(const Animal& inPrey) const;
        virtual bool eatenBy(const Bear& inBear) const;
        virtual bool eatenBy(const Fish& inFish) const;
        virtual bool eatenBy(const Dinosaur& d) const;
};
```

```
class Dinosaur : public Animal {
    public:
        virtual bool eats(const Animal& inPrey) const;
        virtual bool eatenBy(const Bear& inBear) const;
        virtual bool eatenBy(const Fish& inFish) const;
        virtual bool eatenBy(const Dinosaur& d) const;
};
```

Each class implements the polymorphic eats() method, and all overloaded versions of eatenBy()

Double Dispatch Implementations

```
bool Bear::eats(const Animal& inPrey) const {  
    return inPrey.eatenBy(*this);  
}
```

```
bool Bear::eatenBy(const Bear& inBear) const {  
    return false;  
}
```

```
bool Bear::eatenBy(const Fish& inFish) const {  
    return false;  
}
```

```
bool Bear::eatenBy(const Dinosaur& inDinosaur) const {  
    return true;  
}
```

```
bool Fish::eats(const Animal& inPrey) const {
    return inPrey.eatenBy(*this);
}

bool Fish::eatenBy(const Bear& inBear) const {
    return true;
}

bool Fish::eatenBy(const Fish& inFish) const {
    return true;
}

bool Fish::eatenBy(const Dinosaur& inDinosaur) const {
    return true;
}

bool Dinosaur::eats(const Animal& inPrey) const {
    return inPrey.eatenBy(*this);
}
[etc]
```

Evaluating Double Dispatch

- All versions of eats() are the same, but we can't factor them up to the parent.
- We've achieved multi-methods functionality by invoking polymorphism twice.
- We don't need to query the type of the input argument.
- We don't have a bunch of if/else logic.
- Each subclass is forced to consider all of the possible argument types through pure virtual methods.

Mix-In Classes

A mix-in class answers the question *What else can this class do?*

Example: Some animals are pettable

```
class Cat : public Animal, public Pettable {  
    public:  
        [etc]  
};
```

Mix-in classes use multiple inheritance in a very straight-forward and useful way.

Interface Mix-In Classes

You can use a mix-in class just to require that a subclass implements a certain behavior (like Java's interfaces):

```
class Pettable {
    public:
        void bePet() = 0;
};

class Cat : public Animal, public Pettable {
    public:
        void bePet() { cout << "Purrrrrrr" << endl; }
        [etc]
};
```

Behavioral Mix-In Classes

You can also use mix-in classes to add side-functionality:





