Software Development (CS2500)

Lecture 51: The Strategy Pattern

M. R. C. van Dongen

February 28, 2011

1 Outline

These lectures studies the famous *Strategy Design Pattern*. The pattern defines a class of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithms vary independently from clients using it [Gamma *et al.*, 2008]. In addition it explores three design principles: (1) encapsulate what varies, (2) program to an interface, not to an implementation, and (3) favour composition over inheritance. This lecture is based on [Freeman and Freeman, 2005; Gamma *et al.*, 2008].

2 Introduction

Joe works at SimuDuck™. SimuDuck™ specialises in duck pond simulation games. These games involves lots of quacking and swimming ducks. Joe is in charge of SimuDuck™'s most popular game. The game is written in `Java` and is based on inheritance.

3 Initial Design

Figure 1 depicts some of the design. In reality, there are an impressive 50 or so more `Duck` subclasses. All subclasses inherit the `quack( )` and `swim( )` behaviour, except for the `RubbrDuck` class which overrides the `quack( )` behaviour. Each subclass overrides the `display( )` method so subclass instances display the right picture.

4 Enters Mr Change

The economy was in a dip and competition was tough. This explains why Joe's boss decided that SimuDuck™'s application needed something extra. The next time he saw Joe he told him he wanted the ducks to fly.

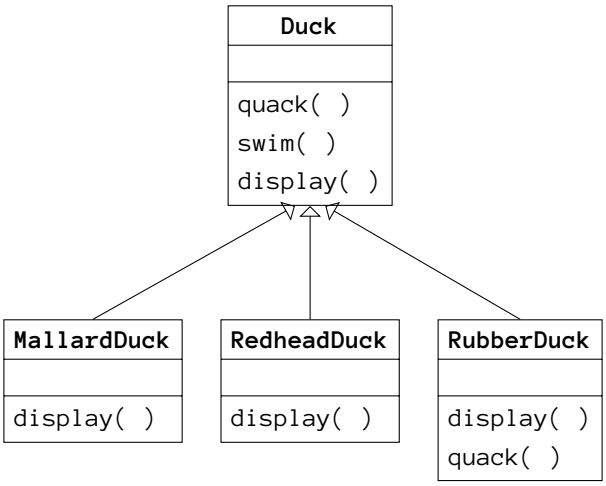

Figure 1: Initial design of game.

5 Inheritance Issues

Figure 2 shows how Joe fixed the problem. He simply added a `fly( )` method to the `Duck` class. All subclasses in this design immediately inherit the default behaviour. He handed in his solution and it was included in the next release of the duck simulator program.

The first day after the release, Joe was called in to his boss' office. His boss had just received a phone call from one of the top clients. It turned out they had `RubberDuck`s flying all over the screen. Thanks to Joe's class design *all* subclasses of the `Duck` class inherited the default `fly( )` behaviour. This included the `RubberDuck` class.... Really, this should have never happened.

Needless to say, Joe's boss wasn't impressed. Joe was sent back to his office to fix the problem.

6 Design Options

So, what should Joe do? Should he override `fly( )` in the `RubberDuck` class? If he did that he might have to duplicate code later. For example what if a `WoodenDecoyDuck` had to be added to the app. `RubberDuck` and `WoodenDecoyDuck` would be almost identical, yet share no code.... Of course he could introduce a common superclass for the `RubberDuck` and the `WoodenDecoyDuck`. But that would mean much work. Also there was no guarantee that work would stop there.

Hmmm, ... Should Joe consider getting rid of overriding? For example, what about using an interface? Figure 3 shows Joe's design with interfaces. Clearly, this really cannot be the solution. All ducks have to implement the interfaces `Flyable` and `Quackable`. For most subclasses the implementations of `fly( )` and `quack( )` are identical, which causes lots of code duplication. Surely, this cannot be the way to add `fly( )` behaviour.

Joe really wants software that doesn't change. After all, the *only* constant thing in software development is change. (No matter how well he designs the software, his boss or a customer will eventually change the specs or ask for new features.) So if code changes this should have no impact on existing code. That

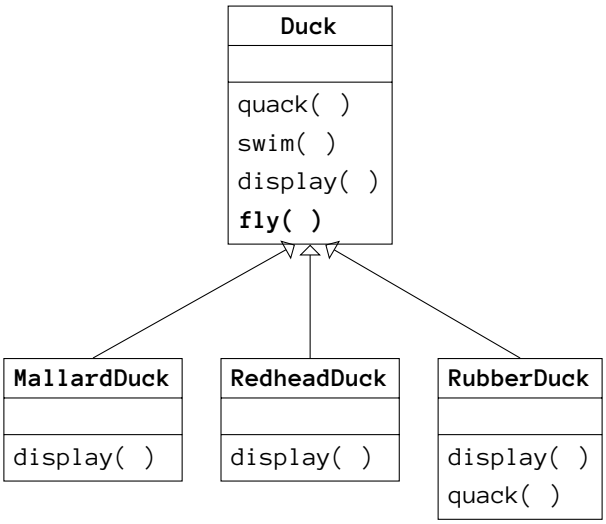

Figure 2: Game design after adding the `fly( )` behaviour.

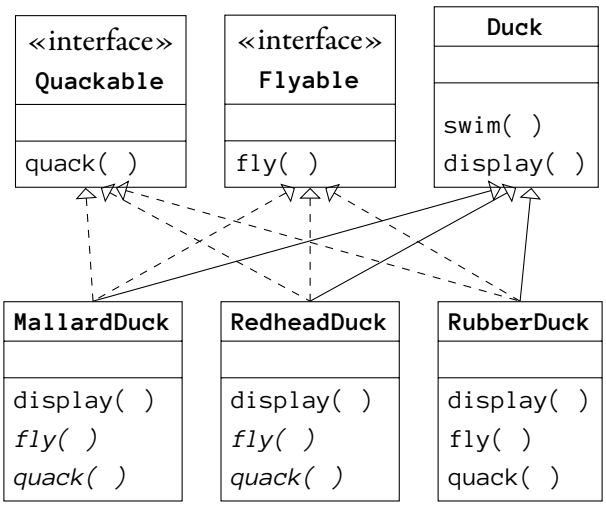

Figure 3: Design after adding interfaces for `fly( )` and `quack( )` behaviour.

would save much time rewriting existing code.

7 Encapsulate what Varies

We've seen that inheritance hasn't worked out for Joe. When the `Duck` class changes this affects all subclasses. The interfaces look nice, but they have no implementation: no reuse of code. The following design principle may help Joe:

Design Principle 1 (Identify what Varies). *Identify the aspects of your application that vary and separate them from what stays the same.*

Applying the principle, the implementation (of what varies) won't affect the remaining code. This reduces the number of potential code change locations. It increases the flexibility of the software.

In Joe's last design most classes implemented `Flyable` and `Quackable` (because the behaviour of these interfaces is what varies). This is what caused the code duplication. What we're going to do is *encapsulate* what varies:

- We separate what varies: `fly( )` and `quack( )` behaviour.
- We define a `Flyable` and a `Quackable` interface. We encapsulate each specific `fly( )` behaviour as a separate *concrete class* that implements the interface `Flyable`. We encapsulate each specific `quack( )` behaviour as a separate *concrete class* that implements the interface `Quackable`.
- We reuse the behaviour in the actual `Duck` subclasses. This is done using *delegation*. (It involves a design pattern.)

8 Program to an Interface

We need to design (lots of) classes that implement  behaviour. We want to *assign* the behaviour to specific `Duck` instance attributes. Assiging behaviour could even be done at runtime. The following design principle is exactly what we need:

Design Principle 2 (Program to an Interface). *Program to an interface, not an implementation.*[1]

We use an interface for each behaviour: `Flyable`, `Quackable`, ... Specific classes implement specific behaviours. We use instances of these classes to use the behaviour. Before we depended on an *implementation*: the implementation of the default behaviour in the `Duck` class or the implementation of the overridden methods. Now we depend on an interface, an object with a type. Client classes are now completely unaware of the actual type and class of the object. This greatly reduces subsystem dependencies.

We've come a long way. After all the hard work we're finally ready to integrate the `Duck` behaviour. Figure 4 depicts the new `Duck` class design.

[1] Here the word "interface" should be interpreted as "supertype".

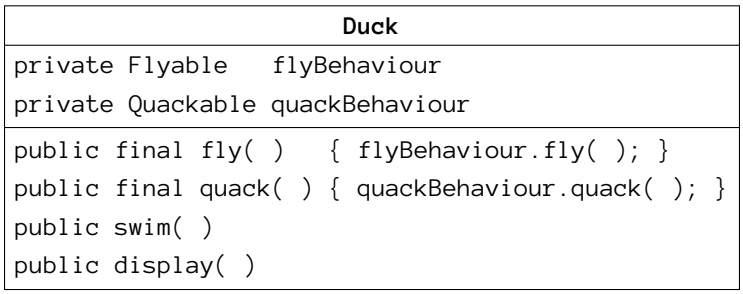

Figure 4: The new `Duck` class.

There are three important changes. First each `Duck` instance has two *attributes* which determine the instance's actual behaviour.

The second change is that the implementations of `fly( )` and `quack( )` now simply *delegate* the behaviour to the new attributes. Of course the methods `fly( )` and `quack( )` are made final, so they cannot be overridden.

The third difference depends on the current design. Notice that the attributes of the `Duck` are not `final`. As a consequence we can now even *change* `Duck` behaviour at runtime. This may be useful for a `MutableDuck`.

Implementing specific `Duck` subclasses is straightforward. The following demonstrates the `MallardDuck` class.

```Java
public class MallardDuck extends Duck {
    public MallardDuck( ) {
        quackBehaviour = new SqueekQuack( );
        flyBehaviour   = new FlyWithWings( );
    }

    @Override
    public void display( ) {
        System.out.println( "MallardDuck here...." );
    }
}
```

Implementing the `MutableDuck` class is not much different.

```java
public class MutableDuck extends Duck {
    public MutableDuck( ) {
        quackBehaviour = new SqueekQuack( );
        flyBehaviour   = new FlyWithWings( );
    }

    public void setQuackBehaviour( Quackable behaviour ) {
        quackBehaviour = behaviour;
    }

    public void setFlyBehaviour( Flyable behaviour ) {
        flyBehaviour = behaviour;
    }

    @Override
    public void display( ) {
        System.out.println( "MutableDuck here...." );
    }
}
```

9 Favour Composition

Inheritance: Lets us create subclasses: *white-box reuse*

- Each subclass automatically inherits superclass behaviour.
- Subclasses can override superclass behaviour.
- You get code reuse for free.
- You cannot change behaviour at runtime.
- Breaks encapsulation. Subclasses may start to rely on superclass implementation. The subclass may break when the superclass is changed.

Composition: Lets you *compose* classes: *black-box reuse*

- A client class can use an object.
- You get code reuse but it takes more effort.
- Lets you change behaviour at runtime.
- Respects encapsulation. This helps encapsulated classes focus on a single task.

In our new design we rely on Has-A (more then on Is-A): each `Duck` has-a `flyBehaviour`, and each `Duck` has-a `quackBehaviour`. The "Has-A" relationship allows us to obtain behaviour by *composing* classes. (As opposed to obtaining it by inheritance — because each subclass instance *is-a* `Duck`.) The result is a more flexible design: It let's us encapsulate behaviour. We can change behaviour at runtime. This brings us to our third design principle:

Design Principle 3 (Favour Composition over Inheritance). *Favour Composition over Inheritance*

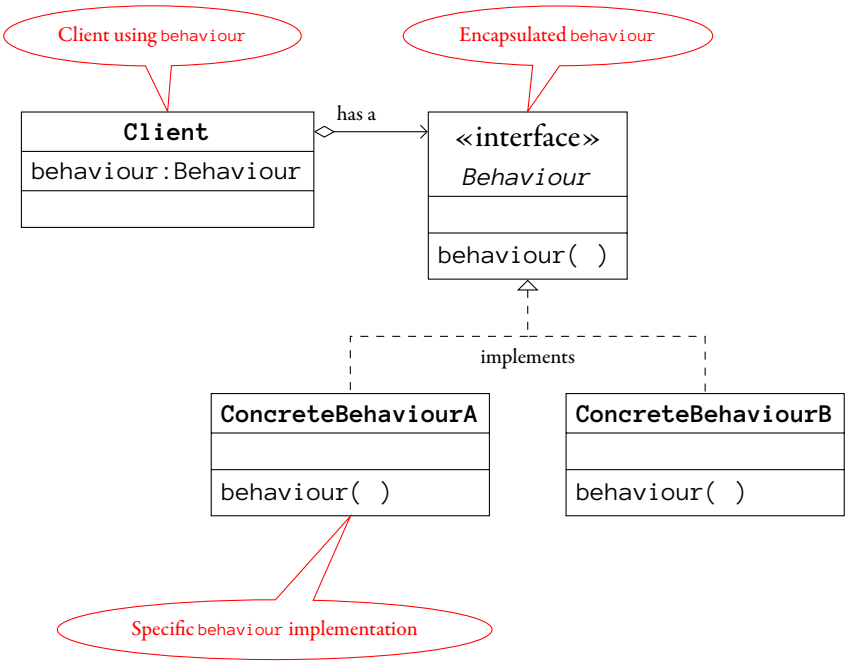

Figure 5: Strategy Pattern in UML. Based on [Gamma *et al.*, 2008, Page 316], which uses `Context` for `Client`, `Strategy` for `Behaviour`, `ConcreteStrategyA–B` for `ConcreteBehaviourA–B`, and `AlgorithmInterface( )` for `behaviour( )`.

10 The Strategy Pattern

The technique which we've used to implement `fly( )` behaviour flexibly is a commonly used design pattern which is known as the *Strategy Pattern*.

Design Principle 4 (Strategy Pattern). *The* Strategy Pattern *defines a class of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithms vary independently from clients using it [Gamma* et al.*, 2008].*

Figure 5 depicts the design pattern graphically.

11 For Wednesday

Study the lecture notes.

12 Acknowledgements

This lecture is based on [Freeman and Freeman, 2005, Chaper 1] and [Gamma *et al.*, 2008].

References

[Freeman and Freeman, 2005] Eric Freeman and Elisabeth Freeman. Head First *Design Patterns*. O'Reilly, 2005.

[Gamma *et al.*, 2008] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns* Elements of Reusable Object-Oriented Software. Addison–Wesley, 2008. 36th Printing.