Inheritance vs. delegation

Delegation can often achieve the same effect as inheritance. Let’s look at another example.

Consider the java.util.Stack class. How many operations does it have?

Suppose in a program you want a “pure” stack class—one that can only be manipulated via `push(...)` and `pop()`.

Why would you want such a class, when Java already gives you that and more?

What is the “simplest” way to get a pure Stack class?

Or you could create Stack class “from scratch.” What’s wrong with doing this?

Another option is to create your own Stack class, but have it include a java.util.Stack.

What is the name for the approach are we using here?

Here’s what this class might look like.

```java
public class MyStack
{
    private java.util.Stack stack;
    public MyStack(){stack = new java.util.Stack();}
}
```
public void push(Object o) { stack.push(o); }
public Object pop() { return stack.pop(); }
public object peek() { return stack.peek(); }
public boolean isEmpty(){return stack.empty();}

Let’s consider inheritance and delegation, from these perspectives.

- Polymorphism.
- Interface.
- Efficiency.
- Amount of code.

**Exercise: Decorator pattern**

In the online lectures for this week, we considered the Decorator pattern.

Recall that Decorator has add-ons, each of which implement the same interface, which the basic object also implements.

Our example uses this hierarchy.

```
«Interface Computer»

BasicComputer

HardwareCustomizer

Laptop GamingComputer
```

Fill in the blanks to complete the pattern.

**Exercise: Strategy pattern**

In Week 6, we considered the Strategy pattern, using quadrature as an example:
Each time you calculate the area of a region, you need to know what the function is that you are calculating the region underneath.

Or consider converting different file formats, e.g., .jpeg, .gif, .eps.

You could write a case statement whenever you needed to invoke one of the algorithms. But this is not a good idea.

Another situation might be where you are manipulating several geometric shapes, e.g., circles, squares, and composites of circles and squares. You need to—

- draw the shapes on a display
- move them to a different location
- rotate them by a certain number of degrees.

These tasks will be performed differently for each shape. You could use a case statement everywhere you need to make the decision.

Another common situation is when you are working with various kinds of files. You need to open, close, and access them differently depending on the file type.

Our example looks like this.

```
«Interface Data»
  ▼
 ▼
Database   CSVData   TSVData

«Interface Extraction»
  ▼
 ▼
DatabaseExtraction   CSVExtraction   TSVExtraction
```

Fill in the blanks to complete the pattern.
Exercise: Command pattern

Another pattern we considered in Week 6 was the Command pattern. We motivated it like this:

A program needs to issue requests to objects. The code that is doing the requesting doesn’t know what the receiver will be, or what operation will be requested.

One example is a check at a restaurant.¹

- The waiter/waitress takes an order from a customer and writes it on a check.
- The check is then queued for the cook, who prepares the food as requested by the customer.
- The check is later returned to the server, who uses it to bill the customer.

Note that the check has nothing to do with the menu; in principle, the same checks could be used at any restaurant.

Now we want to work on an exercise implementing this pattern in Ruby. The class hierarchy looks like the second part of the hierarchy above, but the implementation is different, because the main program passes entire commands to the extraction tools.

Fill in the blanks to complete the pattern.

¹ Michael Duell, “Non-software examples of software design patterns”, Object Magazine, Jul 1997, p54