# Design Patterns

[Skrien §7.1] O-o design is more than deciding which classes work together to solve a problem.

It is more than deciding on suitable public interfaces for these classes.

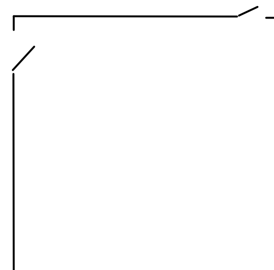
It also has to do with the study of how objects "fit together" to solve common programming problems.

The ways these objects fit together in program after program are called *design patterns*.

Design patterns in programming draw their inspiration from design patterns in architecture.

Back in the 1970s, an architect named Christopher Alexander asked, "Is quality objective?" Or is it just in the eye of the beholder?[1]

If an architect is going to place the doors into a room, can (s)he choose arbitrary locations? Or are some locations better than others?



He came up with the pattern of corner doors:

In his book *A Timeless Way of Building*, Alexander said,

In the same way, a **courtyard**, which is properly formed, helps people come to life in it.

Consider the forces at work in a courtyard. Most fundamental of all, people seek some kind of private outdoor space, where they can sit under the sky, see the stars, enjoy the sun, perhaps plant flowers. This is obvious.

---

[1] Much of this lecture is taken from *Design Patterns Explained: A New Perspective on Object-Oriented Design*, by Alan Shalloway and James Trott, © 2002 Addison-Wesley

But there are subtle forces too. For instance, when a courtyard is too tightly enclosed, has no view out, people feel uncomfortable, and tend to stay away … they need to see out into some larger and more distant space.

Or again, people are creatures of habit. If they pass in and out of the courtyard, every day, in the course of their normal lives, the courtyard becomes familiar, a natural place to go … and it is used.

But a courtyard with only one way in, a place you go only when you "want" to go there, is an unfamiliar place, tends to stay unused … people go more often to places that are familiar.

A pattern, according to Alexander, is a "solution to a problem in a context."

Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

The description of a pattern involves …

- The name of the pattern.
- The problem the pattern solves.
- How the pattern can be implemented.
- The constraints we have to consider in order to implement it.

In the early '90s, some software developers, including Ward Cunningham and Ralph Johnson, happened on Alexander's work. They applied it to programming.

In 1995, *Design Patterns: Elements of Reusable Object-Oriented Software* was published … and the world has never been the same.

## Why study design patterns?

There are several reasons to study design patterns:

- To reuse solutions.
- To establish common terminology.

- To give you a higher-level perspective on the problem.
- They facilitate restructuring a system.

Several features of Ruby facilitate using design patterns. In fact, building blocks for some of the patterns are available as modules in the library.

## Singleton

The Singleton pattern is used to ensure that only one object of a particular class is instantiated.

Can you think of reasons that you might want to put a class in your program, and insure that it is only instantiated once?

The Singleton pattern is available as a mixin in the Ruby library. Including it in the code makes the new method private and provides an instance method used to create or access the single instance.

```
require 'singleton'
class Registry
  include Singleton
  attr_accessor :val
end
r = Registry.new #throws a NoMethodError
r = Registry.instance
r.val = 5
s = Registry.instance
puts s.val   >> 5
s.val = 6
puts r.val  >> 6
s.dup  >> TypeError: can't duplicate instance of
singleton Registry
```

What's the difference between `require` and `include`?

Let's take a look at how this might be implemented.

```
class Single
   def initialize
   # Initialize an instance of the class
  end

  def self.instance
   return @@instance if defined? @@instance
   @@instance = new
  end
  private_class_method  :new
end
```

Actually, the `Singleton` module is more complicated than this. Can you identify one or two additional things it needs to do? Consider these issues.

Also, notice that `initialize` has no arguments. Why do you think this is?

*Exercise:* Write code that uses Single to define an attribute or method, and then print out the attribute, or invoke the method, from code outside the Single class.

One example:

The Singleton pattern can't be implemented this easily in Java. Why not?

## Adapter Pattern

An *adapter* allows classes to work together that normally could not because of incompatible interfaces.

- It "wraps" its own interface around the interface of a pre-existing class. What does this mean?
- It may also translate data formats from the caller to a form needed by the callee.

Can you think of some examples where you would need to do this? Suggest method signatures of the original class, and method signatures for the new class.

One can implement the Adapter Pattern using delegation in Ruby.

Consider the following contrived example.

- We want to put a SquarePeg into a RoundHole by passing it to the hole's peg_fits? method.
- The peg_fits? method checks the radius attribute of the peg, but a SquarePeg does not have a radius.
- Therefore we need to adapt the interface of the SquarePeg to meet the requirements of the RoundHole.

```
class SquarePeg
   attr_reader :width
   def initialize(width)
      @width = width
   end
end
```

```
class RoundPeg
   attr_reader :radius
   def initialize(radius)
      @radius = radius
   end
end
```

```
class RoundHole
    attr_reader :radius

    def initialize(r)
        @radius = r
    end

    def peg_fits?(peg)
        peg.radius <= radius
    end
```

```
end
```

Here is the Adapter class:

```
class SquarePegAdapter
    def initialize(square_peg)
        @peg = square_peg
    end

    def radius
        Math.sqrt(((@peg.width/2) ** 2)*2)
    end
end

hole = RoundHole.new(4.0)
4.upto(7) do |i|
    peg = SquarePegAdapter.new( SquarePeg.new(i.to_f)
)
    if hole.peg_fits?( peg )
        puts "peg #{peg} fits in hole #{hole}"
    else
        puts "peg #{peg} does not fit in hole
#{hole}"
    end
end

>>peg #<SquarePegAdapter:0xa038b10> fits in hole
#<RoundHole:0xa038bd0>
>>peg #<SquarePegAdapter:0xa038990> fits in hole
#<RoundHole:0xa038bd0>
>>peg #<SquarePegAdapter:0xa0388a0> does not fit in
hole #<RoundHole:0xa038bd0>
>>peg #<SquarePegAdapter:0xa038720> does not fit in
hole #<RoundHole:0xa038bd0>
```

*Exercise:* Write a class SquarePegDemo that uses the SquarePegAdapter class to print out, for holes of size 1 to 5, whether pegs of size 1 to 10 fit in each of them.

## Closures and Patterns

Several patterns can be implemented elegantly using closures.

A *closure* is a block of code[2] that has these three properties

- It can be passed around as a value.
- It can be executed on demand by any procedure or method that has that value.
- It can refer to variables from the context in which it was created (it is “closed” with respect to variable access).

When implementing design patterns, it is often useful to pass around code that needs to be executed later.

## Command Pattern

The Command pattern solves this problem:

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.[3]

- 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.

---

[2] Definition adapted from http://innig.net/software/ruby/closures-in-ruby.rb

[3] Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 1997, p54

- 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.

Another use of the Command pattern is undo/redo in an editor. Keeping a history of commands executed allows one to undo them and, if necessary, redo them later.

Can you think of other examples of the Command pattern?

A Command class holds some subset of the following: an object, a method to be applied to the object, and the arguments to be passed when the method is applied.

Ruby's `call` method then causes the pieces to come together.

The Command pattern can be implemented using Proc objects.

A Proc object represents a callable block of code that closes over the variables in scope when it was created.

This leads to a much more concise implementation of Command Pattern than in many other programming languages.

```
count = 0

commands = []
(1..10).each do |i|
    commands << proc { count += i }
end

puts "Count is initially #{count}"
commands.each { |cmd| cmd.call }
puts "Performed all commands.  count is #{count}"
```

```
>>Count is initially 0
>>Performed all commands.  count is 55
```

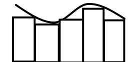
The jukebox example on pp. 55–56 of *Programming Ruby* is another example of this pattern.

*Exercise:* Modify the code sequence above to add other commands to the array, such as instances of AdderGen that we used in Lecture 4. Submit your code here.

## Strategy Pattern

A related pattern is Strategy. This pattern helps when you need to choose an algorithm for a task depending on some "parameter" of the situation.

For example, consider quadrature (numerical integration) again. 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. Is this a good idea?

Consider extensibility and maintainability.

But suppose there is only one case statement. Is it OK then?

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.

But it is better to use Proc objects to implement the Strategy Pattern.

```
class RoutePlanner
 attr_accessor :strategy
 def go(pointA, pointB)
  strategy.call(*args)
 end
 def fastest … end
 def shortest … end
end

ctx = RoutePlanner.new
if on_foot
    ctx.strategy = RoutePlanner.method(:shortest)
    else ctx.strategy = RoutePlanner.method(:fastest)
ctx.go(pointA, pointB)
```

In some cases (like the ones described at the start of this section), polymorphism could be used to achieve the same benefits as the Strategy pattern.

In a lot of cases, it is more straightforward to use polymorphism.

But in this last example, the correct "strategy" for going between two points depends on whether the traveler is driving or on foot. If driving, it may also depend on the price of gasoline vs. the value of your time.

A lot of factors can affect which strategy to use. And these factors can change dynamically. It would be very difficult to capture this by inheritance.

Can you think of other examples of the Strategy pattern?

Now let's consider the difference between Command and Strategy.

## The best explanation I have found is here.

"Command encapsulates a single action. It therefore tends to have a single method with a rather generic signature. It often is intended to be stored for a longer time and to be executed later — or it is used to provide undo functionality (in which case it will have at least two methods, of course).

"Strategy, in contrast, is used to customize an algorithm. A strategy might have a number of methods specific to the algorithm. Most often strategies will be instantiated immediately before executing the algorithm, and discarded afterwards."