Design Patterns

Topics

- SOLID
- Design Patterns
  - Observer
  - Decorator
  - Factory
  - Singleton

SOLID

Single responsibility principle

Open/closed principle

Liskov substitution principle

Injection of dependencies

Demeter principle

Single Responsibility Principle

- A class should have **one and only one** reason to change.
- ex: a module that compiles and prints a report. Can be changed by **content** and **format**.
- Increases robustness of class.

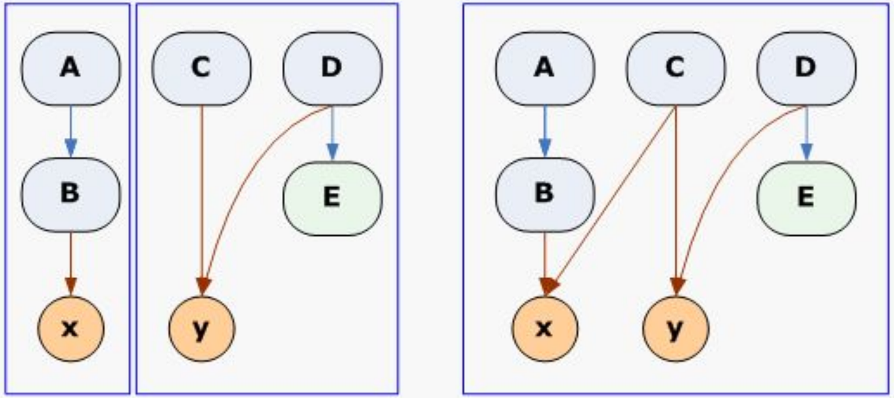
Lack of Cohesion of Methods

- LCOM = 1- (sum(MV_i) / M*V) (between 0 and 1)
- M = # instance methods
- V = # instance variables
- MV_i = # instance methods that access the *i*'th instance variable (excluding "trivial" getters/setters)

Lack of Cohesion of Methods

- LCOM4 counts # of connected components in graph where related methods are connected by an edge
- LCOM4 = 1 indicates a cohesive class, which is the "good" class.
- LCOM4 >= 2 indicates a problem. The class should be split into so many smaller classes.
- LCOM4 = 0 happens when there are no methods in a class. This is also a "bad" class.

Lack of Cohesion of Methods



LCOM4 = 2 LCOM4 = 1

The example on the left shows a class consisting of methods A through E and variables x and y. A calls B and B accesses x. Both C and D access y. D calls E, but E doesn't access any variables.

This class consists of 2 unrelated components (LCOM4=2). You could split it as {A, B, x} and {C, D, E, y}.

In the example on the right, we made C access x to increase cohesion.

Now the class consists of a single component (LCOM4=1). It is a cohesive class.

LCOM4 (Bad Example)

```
class Person
   @first_name
   @last_name
   @address_1
   @city
   @zip_code
   def get_name
         puts @first_name + @last_name
   end
   def get_address
        puts @adress _1 + @city + @zip_code
   end
end
```

It's bad because you have 2 unrelated instance methods.

get_name doesn't reference @adress _1 @city @zip_code while get_address doesn't reference @first_name and @last_name.

LCOM4 (Refactored)

```
class Person
   @first_name
   @last_name
   def get_name
         puts @first_name + @last_name
   end
   def get_full_address    # equivalent to C
in the diagram from the previous slide
        @address = Address.new
        puts @address.get_address()
   end
end
```

```
class Address
   @address_1
   @city
   @zip_code
   def get_address
         puts @address_1 + @city + @zip_code
   end
end
```

Open/Closed Principle

- Classes should be **open for extension** but **closed for source modification**.
- ex: inheritance from abstract base classes
- Valuable in production (why?)

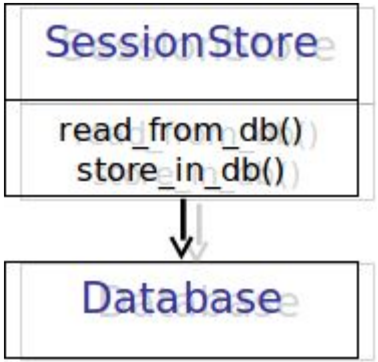
Liskov Substitution Principle

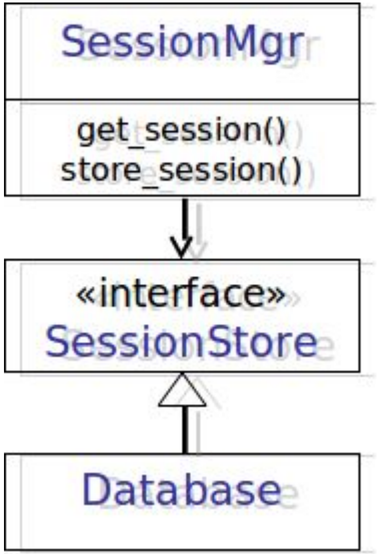
- If **S** is a subtype of **T**, then objects of type **T** can be replaced by objects of type **S**.
- i.e. Objects of type **S** can be <u>substituted</u> for objects of type **T**
- avoid “change to subclass requires change to superclass” scenario
- Inheritance!

Injection of Dependencies

- Problem: **A** depends on **B**, but what if **B**'s implementation and interface change?
- Solution: "inject" an abstract interface that **A** and **B** depend on.
- Ruby: extract module to isolate the interface.

Injection example





Demeter’s Principle

- You can call methods on yourself and your own instance variables, but **not** on the results returned by them.
- i.e. if an object **A** is calling a method of object **B**, object **A** can’t “reach through” **B** to access yet another object **C** because this requires greater knowledge of **B**’s internal structure.

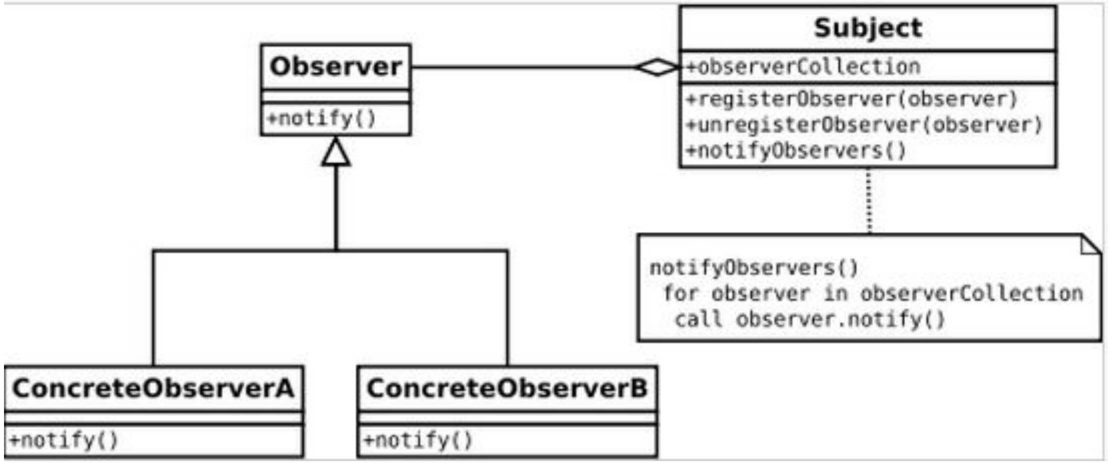
Why SOLID?

- Five basic principles of OOP and design.
- Create a system that is easy to maintain and extend over time.

Design Patterns

Observer

- One subject, many observers who register with subject and are notified when subject changes. Mainly used to implement distributed event handling systems.



Observer - example

Let's consider an `Employee` object that has a `salary` property.

We'd like to be able to change their salary and keep the payroll system informed about any modifications.

The simplest way to achieve this is passing a reference to payroll and inform it whenever we modify the employee `salary`:

```
class Employee
  attr_reader :name, :title
  attr_reader :salary

  def initialize( name, title, salary, payroll)
    @name = name
    @title = title
    @salary = salary
    @payroll = payroll
  end

  def salary=(new_salary)
    @salary = new_salary
    @payroll.update(self)
  end
end
```

Observer - refactored

```ruby
class Employee
  attr_reader :name, :title
  attr_reader :salary

  def initialize(name, title, salary)
    @name = name
    @title = title
    @salary = salary
    @observers = []
  end

  def salary=(new_salary)
    @salary = new_salary
    notify_observers
  end

  def add_observer(observer)
    @observers << observer
  end

  def delete_observer(observer)
    @observers.delete(observer)
  end

  def notify_observers
    @observers.each do |observer|
      observer.update(self)
    end
  end
end
```

Observer - Ruby

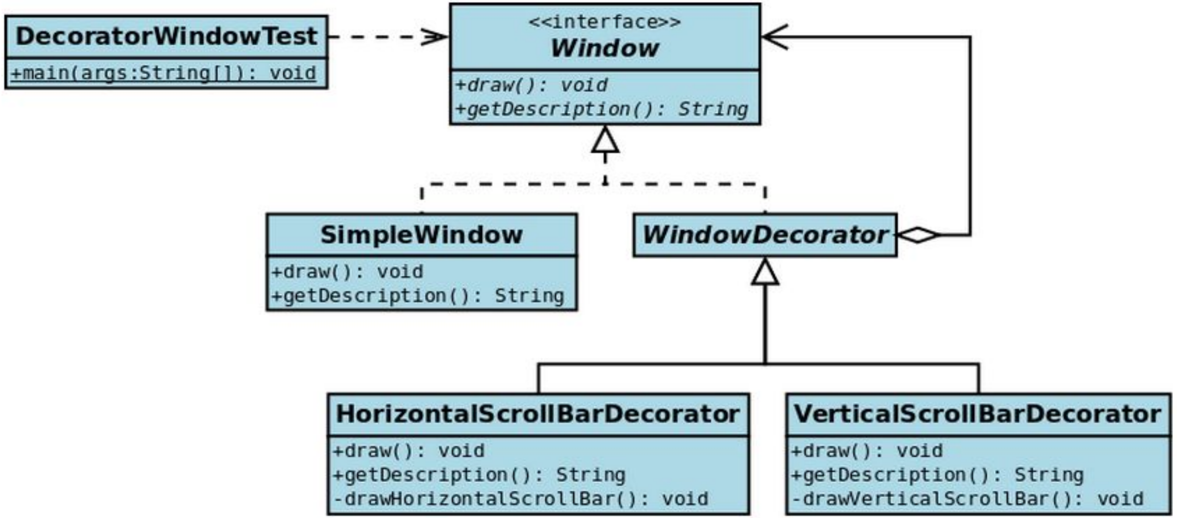
```ruby
require 'observer'
class Employee
  include Observable
  attr_reader :name,  :title
  attr_reader :salary
  def initialize(name, title, salary)
    @name = name
    @title = title
    @salary = salary
  end
  def salary=(new_salary)
    @salary = new_salary
    changed
    notify_observers(self)
  end
end
```

`changed` will set the object's state change to `true`.

`notify_observers` will notify all registered observers when the object's state change is set to `true`.

Decorator

- Add functionality to an object without changing it. Provides flexible alternative to subclassing.



Decorator - example

Here is an implementation of an object that simply writes a text line to a file.

At some point, we might need to print the line number before each one, or a timestamp or a checksum. We could achieve this by adding new methods to the class that performs exactly what we want, or by creating a new subclasses for each use case. However, none of these solutions is optimal.

In the case of the former, the client should know what kind of line is printing all the time. In the case of the latter, we could end up having a huge amount of subclasses, especially if we want to combine the new features.

```
class SimpleWriter
  def initialize(path)
    @file = File.open(path, 'w')
  end
  def write_line(line)
    @file.print(line)
    @file.print("\n")
  end
  def close
    @file.close
  end
end
```

Decorator - refactored

```
class WriterDecorator
  def initialize(real_writer)
    @real_writer = real_writer
  end

  def write_line(line)
    @real_writer.write_line(line)
  end

  def close
    @real_writer.close
  end
end
```

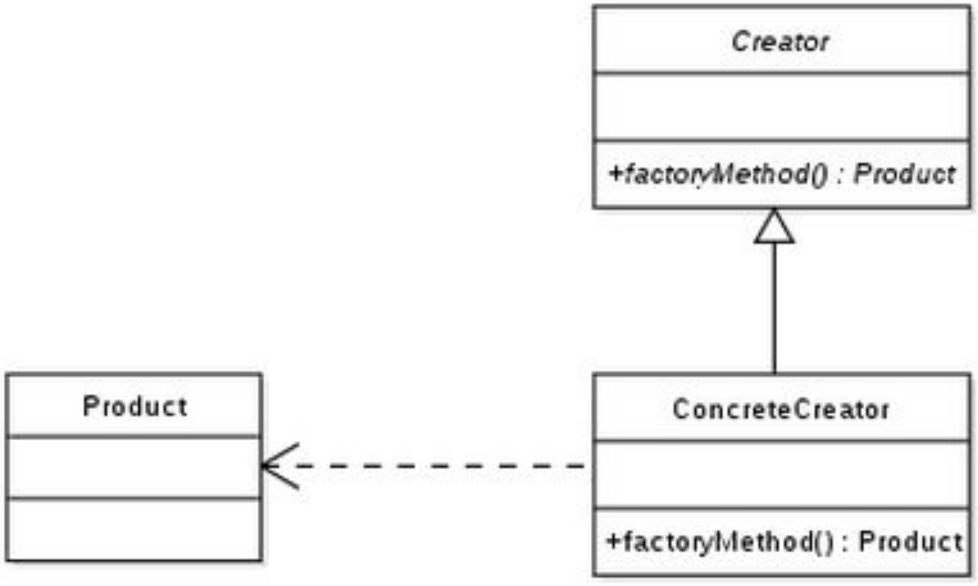
```
class NumberingWriter < WriterDecorator
  def initialize(real_writer)
    super(real_writer)
    @line_number = 1
  end

  def write_line(line)
    @real_writer.write_line("#{@line_number}: #{line}")
    @line_number += 1
  end
end

writer = NumberingWriter.new(SimpleWriter.new('final.txt'))
writer.write_line('Hello out there')
```

You can also chain the decorators.

```
writer = CheckSummingWriter.new(TimeStampingWriter.new(
NumberingWriter.new(SimpleWriter.new('final.txt'))))
writer.write_line('Hello out there')
```

Factories

- Abstract creation of family of objects.



Factories - example

Imagine that you are asked to build a simulation of life in a pond that has plenty of ducks.

But how would we model our `Pond` if we wanted to have frogs instead of ducks? In the implementation above, we are specifying in the `Pond`'s initializer that it should be filled up with ducks.

```
class Pond
  def initialize(number_ducks)
    @ducks = number_ducks.times.inject([]) do
|ducks, i|
      ducks << Duck.new("Duck#{i}")
      ducks
    end
  end

  def simulate_one_day
    @ducks.each {|duck| duck.speak}
    @ducks.each {|duck| duck.eat}
    @ducks.each {|duck| duck.sleep}
  end
end

pond = Pond.new(3)
pond.simulate_one_day
```

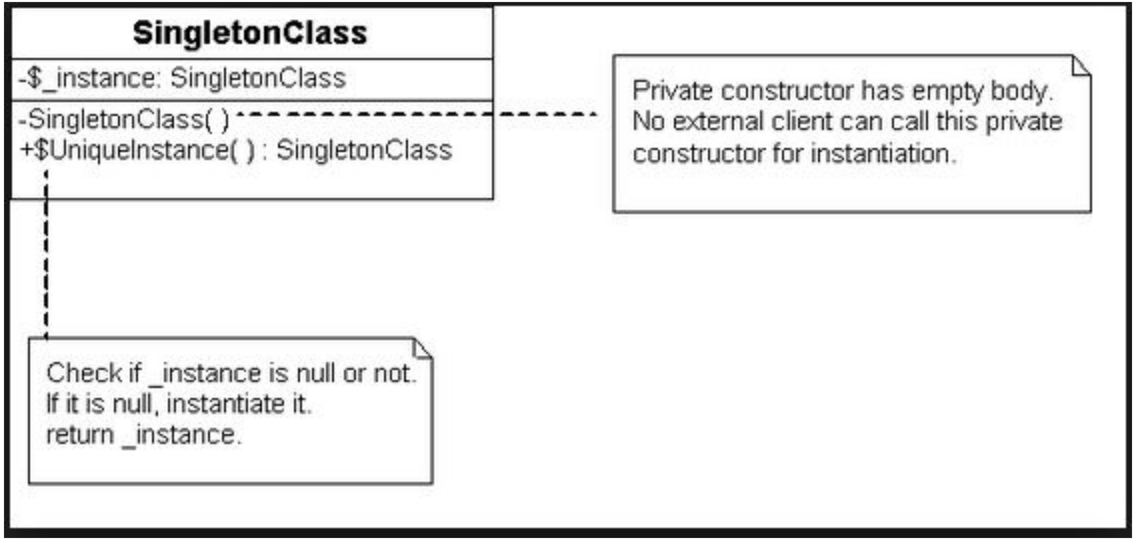
Factories - refactored

```ruby
class Pond
  def initialize(number_animals)
    @animals = number_animals.times.inject([]) do |animals, i|
      animals << new_animal("Animal#{i}")
      animals
    end
  end

  def simulate_one_day
    @animals.each {|animal| animal.speak}
    @animals.each {|animal| animal.eat}
    @animals.each {|animal| animal.sleep}
  end
end
```

```ruby
class FrogPond < Pond
  def new_animal(name)
    Frog.new(name)
  end
end

pond = FrogPond.new(3)
pond.simulate_one_day
```

Singleton

- One unique object



Singleton - example

```ruby
class SimpleLogger
  attr_accessor :level

  ERROR = 1
  WARNING = 2
  INFO = 3

  def initialize
    @log = File.open("log.txt", "w")
    @level = WARNING
  end

  def error(msg)
    @log.puts(msg)
    @log.flush
  end

def warning(msg)
    @log.puts(msg) if @level >= WARNING
    @log.flush
  end

  def info(msg)
    @log.puts(msg) if @level >= INFO
    @log.flush
  end
end
```

Singleton - refactored

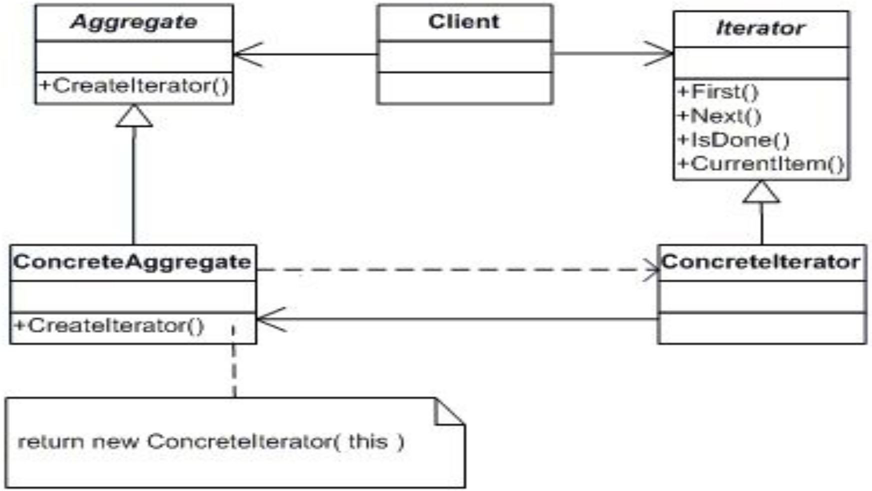
```ruby
class SimpleLogger

  # Lots of code deleted...
  @@instance = SimpleLogger.new

  def self.instance
    return @@instance
  end

  private_class_method :new
end

SimpleLogger.instance.info('Computer wins chess
game.')
```

We can get the same behavior by including the `Singleton` module, so that we can avoid duplicating code if we create several singletons:

```ruby
require 'singleton'

class SimpleLogger
  include Singleton
  # Lots of code deleted...
end
```

Iterator

- Allow client to access each item in a collection without exposing details of the container.



External Iterator

```ruby
class ArrayIterator
  def initialize(array)
    @array = array
    @index = 0
  end

  def has_next?
    @index < @array.length
  end
  def item
    @array[@index]
  end
  def next_item
    value = @array[@index]
    @index += 1
    value
  end
end
```

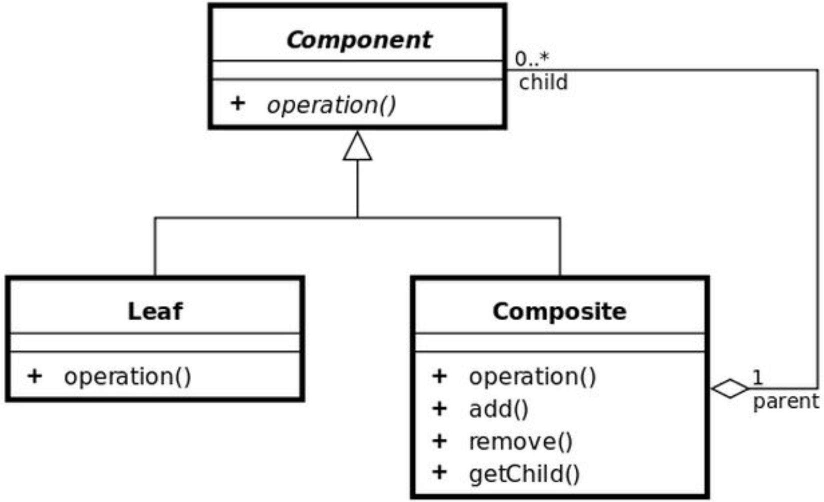
Internal Iterator

```ruby
class Account
  attr_accessor :name, :balance

  def initialize(name, balance)
    @name = name
    @balance = balance
  end

  def <=>(other)
    balance <=> other.balance
  end
end
```

```ruby
class Portfolio
  include Enumerable

  def initialize
    @accounts = []
  end

  def each(&block)
    @accounts.each(&block)
  end

  def add_account(account)
    @accounts << account
  end
end
```

Composite

- Component whose operations make sense on both individuals and aggregates.



Visitor

- Apply type-specific operation to elements in a container without changing the objects' code.