Final Review

Everything must end….

Decorator

- Decorator contains
  - Component class the defines the interface for the object and decorators
  - Concrete component that actually performs the operation
  - Decorator that adds something additional to the operation, then forwards it on

```
class Component
      def intialize(component)
            @component
      end

      def operation()
            raise ‘abstract’
      end
end
```

```
class ConcreteComponent
      def operation()
            …
      end
end
```

```
class Decorator
      @component

      def operation()
            …
            @component.operation()
      end
end
```

Decorator

- When to use Decorator:
    - add various features in no particular order or number
    - vary the responsibilities of an object during runtime
- Limitations
    - CheckSummingWriter.new(TimeStampingWriter.new(NumberingWriter.new(SimpleWriter.new('final.txt'))))
    - Decorators incur a performance hit as they grow longer and longer
- Ruby Extras
    - alias keyword
        - Client still has to build the object with an ugly line like:
        - allows you to assign a method another name at runtime
            - class MyClass

              alias newMethodName oldMethodName

Singleton

- Singleton contains
  - A single class that manages a single instance and sets the new method to private

```
class Singleton

      private_class_method :new

      def self.instance
            @@instance || @@instance = Singleton.new
            return @@instance
      end

end
```

```
single = Singleton.instance
```

Singleton

- When to use Singleton:
    - You want only one instance and that provides global access to that one instance.
    - The client should not manage the creation and access to its sole instance.
- Limitations
    - Leads to tightly coupled classes that are dependent on one another’s state
    - Makes unit testing difficult
- Ruby Extras
    - Class variables
        - use that @@ to define a class variable, ex. @@class_var
    - Class methods
        - use self to define class methods, ex. def self.foo()

Factory

- Factory contains
  - A Base Factory and Sub Factories that produce sub-products

```
class BaseFactory
	def createProduct ()
		product = newProduct()
		...
	end
	def newProduct()
		raise NotImplementedError.new
	end
end
class ConcreteFactoryA < BaseFactory
	def newProduct
		ConcreteProductA.new
	end
end
class ConcreteFactoryB < BaseFactory
	def newProduct
		ConcreteProductB.new
	end
end
```

```
class BaseProduct
	…
end
class ConcreteProductA < BaseProduct
	…
end
class ConcreteProductB < BaseProduct
	...
end
```

Factory

- When to use factory:
  - You need to create and process a family of related objects
  - The Template Method for creating objects
- Limitations
  - Can result in if/else statements (requiring modification) if parameterized in static languages
- Ruby Extras
  - Subclasses are unnecessary
    - You can pass in the appropriate class as a parameter because classes are also objects in ruby

Builder

- Builder contains
    - A Builder, Product, and director to configure the product

```
class Builder
	def initialize
		@product = Product.new
	end

	def addOptionA
		@product.optionA = OptionA.new
		self
	end

	...
end
```

```
class Product
	@optionA
	...
end

...
#director
builder = Builder.new
builder.addOptionA.addOptionB(x).buildProduct
```

Metaprogramming

- When to use factory:
  - Define simple classes, methods, etc, on the fly
  - Use eval to run ruby code during runtime
- Limitations
  - Dangerous security and difficult to debug
- Ruby Extras
  - Built into ruby

Builder

- When to use builder:
  - When you need to configure and create a complex object made up of other complex objects
  - When you need to verify the options of a configured object
  - Essentially decorator applied to creating objects
    - uses methods instead of classes
- Limitations
  - Requires the director to know the options available
- Ruby Extras
  - Magic Methods
    - parse methods calls in method_missing to map onto configuration methods

Interpreter

- Interpreter contains
  - Client, Expression, Terminal, and Non-Terminal classes

```
class Expression
      def initialize()
            raise “abstract”
      end
      def value()
            raise “abstract”
      end
end
```

```
class Terminal < Expression
      def value()
                  ...
      end
end
```

```
class NonTerminal < Expression
    def value()
       self.operation(@child1.value, @child2.value)
    end

   operation()
      #non-terminal operation
   end
end
```

Interpreter

- When to use Interpreter:
  - You need to define a language for the user
- Limitations
  - Limited to business problems, cannot make overly complex
- Ruby Extras
  - You ruby’s flexible syntax to make the language parser-less