Adapter Pattern

CS 342 Fall 16

Classwork

Velocity

- What did you return from your interface?
    - Miles, Kilometers, hands?
- What names did you use for stellar objects?
    - Catalogue names, Flamsteed Numbers, Greek Lettering System?

Encrypt a File

- The Encrypter class uses a key, and encrypts a file byte by byte by reading a byte from one file and writing to another file
  - key is a string

```
class Encrypter
    def initialize(key)
        @key = key
    end
    def encrypt(reader, writer)
        key_index = 0
        while not reader.eof?
            clear_char = reader.getc
            encrypted_char = clear_char ^ @key[key_index]
            writer.putc(encrypted_char)
            key_index = (key_index + 1) % @key.size
        end
    end
end
```

Using the Encrypter

- Use the previous class with the following:

  -
    ```
    reader = File.open('message.txt')
    writer = File.open('message.encrypted','w')
    encrypter = Encrypter.new('my secret key')
    encrypter.encrypt(reader, writer)
    ```
- What if we now need to encrypt a string?
  - What if the input is not what we expected?
  - Solutions?
    - write a whole new encryption class that works with strings?

Build an Adapter class

```
class StringIOAdapter
    def initialize(string)
        @string = string
        @position = 0
    end
    def getc
        if @position >= @string.length
            raise EOFError
        end
        ch = @string[@position]
        @position += 1
        return ch
    end
    def eof?
        return @position >= @string.length
    end
end
```

- The StringIOAdapter allows you to treats strings as files.
- Driver code:

```
encrypter = Encrypter.new('XYZZY')
reader= StringIOAdapter.new('We attack at
dawn')
writer=File.open('out.txt', 'w')
encrypter.encrypt(reader, writer)
```

Defining Interfaces

- Software is often designed with a (somewhat) arbitrary interface.
- Adapter classes bridge the gap between an expected interface, and the actual interface
  - Adapter classes should be invisible to the client classes after initialization. It should look exactly like the interface they were expecting.

Dynamic Interfaces

- What if we don’t know our interface at compile time?
  - For example, we know an object that contains a string and an integer is being sent over the network, but we don’t know its interface (how to get the data)
    - Do we access the string with getString() or just string()?
- How can we use the object, without knowing its methods?
  - What if we could redefine the interface for an object when we receive it?
    - “WHAT IS THIS MADNESS” you say? Let see...

Near Misses

- Our spaceprobe interfaces were (probably) close, with just minor differences
  - but enough of a difference to crash the mars lander
- The adapter class is a viable solution, but in Ruby we can do better
  - Classes are never final in ruby, which is awesome and terrifying
  - All classes (even library classes) are open for modification during runtime

Class Runtime Modification

```
require_relative ‘Velocity’

class Velocity
	def slower() #overwrites the existing slow method
		@distance = gps.totalDistance(object)*2.54 - 1
	end
	def faster(distance) #overwrites the existing faster method
		@distance = gps.totalDistance(object)*2.54 + 1
	end
	def convert(distance) #adds a new method
		@distance = gps.totalDistance(object)*2.54
	end
end
```

- Any class can be modified at any time
  - even built in classes: Fixnum, Proc, etc.
    - don’t ever do this!

Object Runtime Modification

```
vel = Velocity.new

class << vel
	def slow()
		distance = gps.totalDistance(object)*2.54 -1
	end
	def faster(distance)
		distance = gps.totalDistance(object)*2.54 + 1
	end
end
```

- You can also modify single object behaviour
  - less scary and permanent
- Additional syntax

  -
    ```
    def vel.slow()
        #...
    end
    ```

Adaptor Class or Modify Original

Modify

- Pros
  - Simpler code
  - Runtime flexibility
- Cons
  - Violates encapsulation principles
  - changes the interface for other dependant classes

Adapter Class

- Pros
  - Maintains encapsulation
  - Does not risk side effects due to implementation ignorance
- Cons
  - Requires yet another class
  - Increases code complexity

Problems with the Adaptor

- You develop an adaptor to make a string appear as if it were a file by shadowing getc and eof? methods.
- What if the client then tries to use the basename() method?
  - You have not implemented it, so the call will break the software
- **Pretending be to something you are not is great, unless you get caught**
  - **Life Lesson here**

Adapter is not a subclass

- The Adapter class illustrates the highly recommended techniques of composition and delegation
  - We did not make StringIOAdapter a subclass of String...Why?
- The GoF is against it: “experience shows that unnecessary use of inheritance will corrupt your design”
  - Also, multiple inheritance may be required if more than one adaptee is present

Imposter Patterns

Imposter Patterns are a selection of patterns that wrap the interface of another class. They can be used to hide, modify, or simplify interfaces.

Adapter Pattern is the first of these. Next…