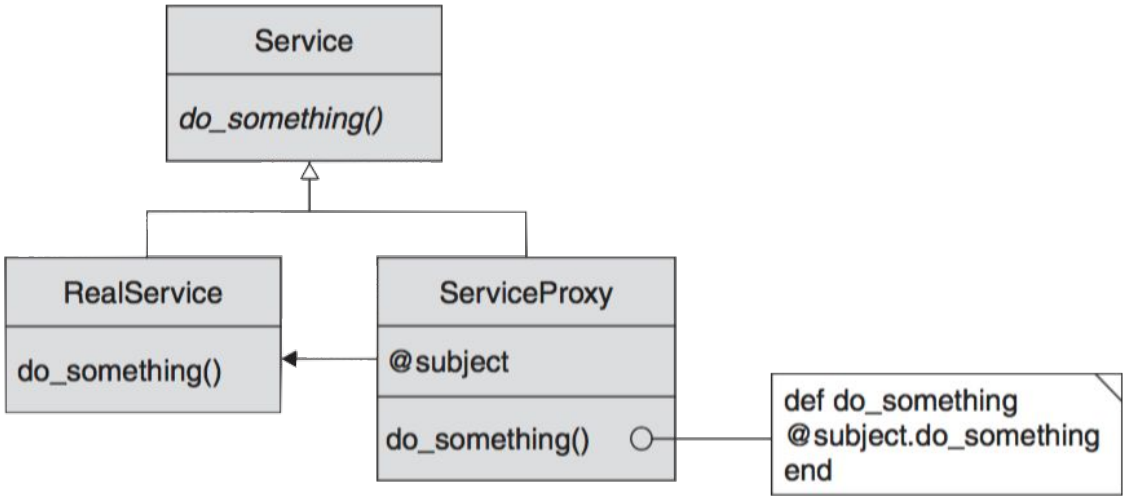
Proxy Pattern

CS342 Fall 15

Design Problem

A Bank Account class authorizes a client and returns an object that will provide them with their account information, balance, last transaction, etc. The object also has to be remote so that it is never stored on the client's computer and must only be created at runtime when the client requests the object.

We need a middleman



Design Discussion

Develop a class that sits between the client and the BankAccount object, and provides transparent access to the following methods:

- balance()
- withdraw()
- deposit()

Banking Class

What's the point?

We have now separated the bank account implementation from the access to the details

```
class BankAccount
        attr_reader :balance
        def initialize(starting_balance=0,
                                account_info)
                @balance = starting_balance
                @info = account_info
        end
        def deposit(amount)
                @balance += amount
        end
        def withdraw(amount)
                @balance -= amount
        end
end
```

```
class BankAccountProxy
        def initialize(real_object)
                @object = real_object
        end
        def balance()
                @object.balance
        end
        def deposit(amount)
                @object.deposit(amount)
        end
        def withdraw(amount)
                @object.deposit(withdraw)
        end
end
```

Proxy Pattern

- Provides a surrogate or placeholder for another object to control access to it.
- Creates a chokepoint for the information

Protection

- We can now add a protection layer to our bank account proxy
- The BankAccount object never gets called unless the security is passed first

Protection Proxy

```
class BankAccountProtectionProxy
    def initialize(real_object, owner)
        @object = real_object
        @owner = owner
    end
    def balance()
        check_access
        @object.balance
    end
    def deposit(amount)
        check_access
        @object.deposit(amount)
    end
    def withdraw(amount)
        check_access
        @object.deposit(withdraw)
    end
```

```
def check_access
    if Etc.getlogin != @owner_name
        raise "Illegal access: #{Etc.getlogin} cannot access account."
    end
end
```

Why Proxy?

- We could have included the checking code in the BankAccount object itself.
  - What advantage does using a proxy for protection provide?
    - Separation of concerns: The proxy worries about who is or is not allowed to do what. The only thing that the real bank account object need be concerned with is the bank account.
  - What if we want to change security protocols?
    - Easily swap security protocol during runtime, since security is handled by proxy
  - What security advantage does proxy provide?
    - All access has to be explicitly coded, so less chance of information leak

Remote Proxy

- What if your BankAccount object lived on a server
  - Client would have to write code to connect to the server, send data, receive data, interpret data, then repeat
- Instead, package up the details of connect, write, read, organize into a proxy class.
  - This is how most API (Application Programming Interface) interfaces work using RPC (Remote Procedure Call)

API access

- TMDB.com has an API
  - info about Critic Ratings, actors, year, etc. for movies
- Without a proxy
  - contact TMDB.com with an api key
  - returns an authentication token
  - make a call to an API url with token
  - receive info requested
  - Parse the return JSON file resource
  - Organize data into an object
- With a proxy
  - create aTMDB object
  - call getInfo().

Web API

- How web requests work
  - A URL is just an address of a file on a server
  - GET resource request returns status code and the resource (file) if present
  - Every page you visit is downloaded to your machine and is then interpreted by your browser
- You can also make requests for non-HTML pages
  - images, mp3, video
- The standard data format is becoming JSON
  - also XML or plaintext

API standards

- An API is a URL scheme that allows RESTful resource requests
  - You can make a request to a public server, and get your data returned as a JSON document
  - a RESTful server means the same url always returns the same resource
- Most scripting languages have built-in JSON parser, which turns a JSON file into an object (or array) in your code
  - You then use this data in your program

JSON Interlude

- Javascript Object Notation
  - JSON is just a text file that follows certain conventions
- JSON's structure allows us to store data in files for later
  - An example of JSON formatted text
    - {"name": "Steven", "age": 39, "isEmployed": true}
      - #what type of collection type does this look like?
- Working with JSON files in ruby is simple
  - Import the JSON module
    - The JSON module parses objects, and automatically converts them to JSON strings

Writing JSON

- You can take any JSON string and convert it to a Hash
  - jsonData = '{"name": "Steven", "age": 39}'
    jsonToRuby = JSON.parse(jsonData)
- You can take any object, and convert it to a JSON string
  - myhash = {:name=>‘Steven’, :age=>39, :isEmployed=>True}
    hashToJson = myhash.to_json
- Any object can be converted to a json string:
  - 1.to_json

Using JSON

- We can convert our objects to strings, and write them out to a file
  - This allows us to send state across a network
- Then we read in our JSON file as a string, and convert to a hash
  - Reading state (or data) sent across a network
- JSON is much simpler and ‘lighter’ than other data exchange formats (XML), making it ideal for API data requests

Deferred Creation

```
class BankAccountProxy
        def initialize(account_num)
                @account=account_num
         end
        def balance()
                subject
                @object.balance
        end
        def deposit(amount)
                subject
                @object.deposit(amount)
        end
        def withdraw(amount)
                subject
                @object.deposit(withdraw)
        end
        def subject
           @subject || (@subject = BankAccount.new(@account))
        end
```

- What if our bank account object was expensive to create.
  - Wait until absolutely necessary

Virtual Proxy

- Defers creation of the object until needed
  - Once the object is created, proxy delegates requests to the real object
- Con
  - Proxy is now responsible for creating the object, taking control from the client

Proxy Drudgery

- Proxy classes require a rewrite of all proxied methods
  - Classes that have >100 methods are not uncommon, and proxying those classes would be a lot of extra code waiting to introduce bugs
- Ruby to the Rescue
  - Message passing is a fundamental concept of Object oriented programming
    - The method account.deposit(50) means you are sending the deposit method to an account object
      - In a statically typed language this is synonymous with calling a method ‘on’ an object

method_missing

- When you invoke account.deposit(50), Ruby will do exactly what you expect:
  - look for the deposit method first in BankAccount's class,
  - then in its superclass, and so on
    - If Ruby finds the method, we get the behavior
- What if it doesn't find the behavior (method)?
  - Ruby calls a default method called 'method_missing()'
    - It then searches the class structure for method_missing, which it will find in the BasicObject class
    - method_missing raises the NoMethodError exception.

Redefine Missing Methods

- By overriding method_missing(), you can build a class that can catch any arbitrary method (or message)
  - method_missing has the following parameters
    - symbol => the name of the message (or method) called
    - args => the parameters of the message
- Using the symbol and parameters, you can send a message to any object
  - Because ruby uses the message model, all methods are really a sent message with the superclass method send()

send()

- send() invokes the method identified by symbol, passing it any arguments specified
  - for example, myObj.foo(2) == myObj.send(:foo, 2)
- The arguments to send are the same as the arguments to method_missing when no method is found
  - How can we use this for easier proxies?

Lazy Proxies

- Just don’t implement the methods you want to proxy
  - class AccountProxy
    def initialize(real_account)
      @subject = real_account
    end
    def method_missing(name, *args)
      check_access
      @subject.send(name, *args)
    end
  end

Reuseable Proxies

- No matter how long your subject class is, you never need to extend your proxy beyond the method_missing
  - All methods of the class will be caught
  - You can override specific methods that need special attention
- You can reuse the class for anything that requires the same proxy procedures
  - Regardless of what its behaviours are, if it needs an access_check before calling its method, just reuse the proxy

Downside of method_missing?

- Performance
  - The message search must travel up the inheritance chain before calling method_missing instead of a direct call
- Obfuscated code
  - It may not be obvious what is happening in your code if you can call a method that is not in the class

Adapter vs. Proxy

- Adapter
  - Changes the interface of the class or object to the expected interface
- Proxy
  - Maintains the objects same interface, but controls access to it in some way
- Both transparently stand in between the Subject and the Client

Classwork:

University Personnel

```
class PersonnelAdaptor
        def intialize(faculty)
                @emp = faculty
        end
        def method_missing(symbol, args)
                if(symbol == :makeReport)
                        @emp.send('make'+@emp.class+'Report')
                else
                        super.method_missing
                end
        end
end
faculty_a = PersonnellAdapter(faculty)
faculty_a.makeReport()
-OR-
class Faculty
        def makeReport()
                        makeJuniorFacultyReport()
        end
end
faculty.makeReport()
-OR-
class <<< facultyObject
        def makeReport()
                        makeJuniorFacultyReport()
        end
end
faculty.makeReport()
```

Classwork:

Placeholder Image

```
class ProxyImage
        def initialize(subject, x, y)
                @image = subject
                @placeholder = ImageFile(...)
        end

        def showImage()
                @placeHolder.show(x, y)
                image = subject.showImage()
                @placeHolder.remove()
                image.show(x, y)
        end
end
```