Sofware Development with Scripting Languages: Design Patterns

Onur Tolga Şehitoğlu

Computer Engineering,METU

2 April 2012

Motivation

- Object Oriented design $\leftarrow$ code reusability.
- Solving similar problems in similar ways.
- Can we re-use design as well?
- Design patterns are not algorithms, data structures, libraries.
- Reusable solution to a commonly occuring problem.
- Origins: architecture. Applied to software development later.

Classification of Design Patterns based

- Creational: creation of new objects
- Structural: relationships between entities
- Behavioural: communication patterns between objects
- Domain specific: (user interface, visualization, security, server, concurrency, ...)

Singleton Pattern

- Problem: You need only one instance of an object in the application
- For example database, file store, game board, interface object, etc.
- Naive solution: Use global a variable, just don't create the second object. You are the programmer aren't you?
- Problem with the naive solution: Global namespace gets crowded variable name vs class name confusion, you need to make sure second object is not created, always need to create the object even not used.
- Singleton: A special class implementation that returns always the same object when created.

```
class Singleton(object):
    def __new__(cls, *a, **k):
        if not hasattr(cls, '_inst'):
            # super calls __new__ of the 'cls'
            cls._inst = super(Singleton, cls).__new__(cls, *a, **k
        return cls._inst
class Counter(Singleton):
    def __init__(self):
        if not hasattr(self,'val'):
                self.val = 0
    def get(self):
        return self.val
    def incr(self):
        self.val += 1

a = Counter()
b = Counter()
Counter().incr()
Counter().get()
```

Subclassing other superclasses are problems. A python decorator version:

```
def singleton(cls):
   _instances =    # keep classname vs. instance
   def getinstance():
      '''if cls is not in _inst.. create, store, return the instan
      if cls not in _instances:
         _instances[cls] = cls()
      return _instances[cls]
   return getinstance

@singleton
class Counter(object):
    def __init__(self):
        if not hasattr(self,'val'): self.val = 0
    def get(self):
        return self.val
    def incr(self):
        self.val += 1

a = Counter()
Counter().incr()
Counter().get()
```

- Python decorators are evaluated during function/class definition and maps function/class definition into a new one
- Decorators are functions returning a callable that replaces function/class constructor

```
def dec():
    def f():
       .....
    return f

# ---
@dec
class cls(): class defn here
# equivalent to
class cls(): class defn here
cls = dec( cls )
# ---

obj = cls()     # call actually made on f() inside dec()
```

Factory Pattern

- Problem: Create objects from a set of classes. Implementation depends on class definitions. Introduction of new classes and other changes need recompilation. Class constructors exposed.
- Solution: Use interface functions/objects to encapsulate class names and constructors. Use a interface functions, methods to provide you instances. Rest is handled by polymorphism.
- Central lifetime management and recycling of objects are possible.

```python
class Player(object):
    def getName(self):
        return self.name
    def hitpower(self):
        return self.hit

class Peasant(Player):
   def __init__(self, name):
       self.name = name
       self.hit = 1
class Warior(Player):
   def __init__(self,name):
      self.name = name
      self.hit = 10

@singleton
class PlayerFactory(object):
   def new(self,name,type):
       if type == 'peasant':
            return Peasant(name)
       elif type == 'warior':
            return Warior(name)
       else:
            return None

a = PlayerFactory().new('ali','peasant')
b = PlayerFactory().new('veli','warior')
```

Facade Pattern

- Problem: A complex library, tool or system; consisting of many functions and/or classes; probably poorly designed is tried to be accessed. It is hard to read and understand. There are many dependencies distributed in the source, needs many housekeeping tasks and stages to access. Any change in the system require changes in the whole source code.
- Solution: Define a class implementing all details of the library/system and providing a simple uniform interface. Access the library through this interface.

```
'''libraries here'''
class AuthFacade(object):
   def __init__(self,method):
       if method == 'passwd':
            # ... use crypt for linux password authentication
       elif method == 'oauth':
            # ... use oauth methods to check authentication
       elif method == 'otp':
            # ... use one time password routinges for authenticati
       else:
            throw ...
   def auth(self, identity, data):
        ''' check authentication based on setting'''

a=AuthFacade('otp')
result  = a.auth('onur','123456')
```

Proxy Pattern

- Problem: You need to access a hard to duplicate, limited, probably old class definition. You donot have a chance to improve it or change the interface. Or, you want to have restricted access to methods (authorization). Or, you want smarter access like caching.
- Solution: Write a class interface implementing functionalities missing in the original interface.

```
@singleton
class Counter(object):
   def __init__(self):
        self.val = 0
   def incr(self):
        self.val += 1
   def get(self):
        return self.val

class CounterProxy(object):
   def __init__(self, type):
        self.type = type
   def incr(self):
        if type == 'W':
                Counter().incr()
   def get(self):
        return Counter().get()

a = CounterProxy('R')
b = CounterProxy('W')
a.incr()
b.incr()
```

- Proxy, Facade, and Adapter provide similar encapsulation.
- In Proxy, interface of the wrapped class is same with the Proxy. It adds extra functionality (i.e. accounting, caching, remote object, access control)
- In Adapter, interface of the wrapped class is different than Adapter class. Caller interface is mapped to the adapted interface.
- In Facade, there is no single class to be wrapped but a set of classes, a subsystem. Facade simplifies it.

Decorator

- Problem: Objects having different behaviour for different run-time settings.
- Naive solution: Have a subclass for each different combination of attributes. n attributes $\rightarrow 2^n$ subclasses.
- Solution: Create a decorator class wrapping the original behaviour. Cascading behaviours for each new decorator.

```
class Coffee(object):
   def __init__(self,size):
        self.size = size
   def cost(self):
        if self.size == 'S': return 2.0
        elif self.size == 'M': return 3.0
        elif self.size == 'L': return 3.5
        else raise InvalidSize

class CoffeeDecorator(object):
        def __init__(self,w):
                self._w = w  #wrapped object
        def __getattr__(self,name):
                return getattr(self._w, name)
class WithMilk(CoffeeDecorator):
        def cost(self):
                return 1.0 + self._w.cost()
class WithCaramel(CoffeeDecorator):
        def cost(self):
                return 0.5 + self._w.cost()

a = WithCaramel(WithMilk(Coffee('L')))
print a.cost()
```

Observer

- Problem: Objects depending on eachothers states need to be informed when the other encountered a change. Event handling systems.
- Solution: Maintain a registry of observing objects in Subject object. When an event occurs, notify the observers.

```
class Subject(object):
        _observers = []
        def register(self,obs):
                self._observers.append(obs)
        def unregister(self,obs):
                self._observers.remove(obs)
        def notify(self):
                for o in self._observers:
                        o.update(self)
        def state(self): pass

class Observer(object):
        def update(self,subj): pass

class Clock(Subject):
        def __init__(self):
                self.value = 0
        def state(self):
                return self.value
        def tick(self):
                self.value += 1
                self.notify()

class Person(Observer):
        def update(self, obj):
                print "heyo␣",obj.state()

a = Person()
b = Clock()
b.register(a)
b.tick()
```

Strategy

- Problem: Combination of behaviours from multiple class hierarchies.
- Naive Solution: Use multiple inheritence. However for each combination of classes a new derivation required. Adding new behaviours is not easy.
- Solution: Encapsulate behavours using interfaces and keep references in the actual class.

State

- Problem: A class needs to behave differently based on its current state.
- Naive solution: Keep state in member variables and use conditional statements to implement specific behaviours at each method. Each method depends on state and conditions replicated. Changing a state need update of conditionals at each method.
- Solution: Encapsulate state and implement each behaviour on concrete classes of State depending on current state.

```
class Cursor(object):
        def __init__(self):
                self.current = Select()
                mouse = "up"
                start = None
        def mouseDown(self,point):
                self.current.mouseDown(self,point)
        def mouseUp(self):
                self.current.mouseUp(self,point)
        def setTool(self,tool):
                self.current = tool
class State(object):
        def mouseDown(self, obj, point):
                self.obj.start = point
class Select(State):
        def mouseUp(self, cur, point):
                if figure.selected:
                        figure.selected.move(cur.start, point)
                else
                        figure.selectobjects(cur.start, point)
class Line(State):
        def mouseUp(self, cur, point):
                figure.newlineobject(self.cur.start, point)
class Erase(State):
        def mouseUp(self, cur, point):
                figure.deleteobjects(cur.start, point)

a = Cursor()
a.mouseDown((10,0))
a.mouseUp((0,20))
a.setTool = Line()
a.mouseDown((10,10))
a.mouseUp((20,20))
```

Command

- Problem: Client program want to send commands but does not want to know internals of how command is executing. Providing extra functionalities like undo, macro, transaction, composition etc.
- Solution: Encapsulate information for a method call inside of a command object. (object, method and parameters)
- Client, invoker and receiver. Invoker keeps track of method calls and functionalities.
- Muli-level undo (stack of method calls), transatcions, progress bars, wizards, macro records, parallel processing.

Model View Controller

- Architectural Patterns patterns of software architecture
- Model View Controller: Split of roles in an application.
  - Model is the business logic and internal representation.
  - View is the user interface, presentation of information.
  - Controller is the user input.
- Interactions
  - user interaction $\rightarrow$ handler in the controller
  - controller $\rightarrow$ model update, change
  - model change $\rightarrow$ view (observer) or
    controller $\rightarrow$ view inform about the change
  - view $\rightarrow$ model, load the changes and render the new information.