

Singleton

```
def Singleton(cls):
    _in stances = {}
    def getins tan ce():
        if cls not in _insta -
nces:
            _in sta nce s[cls] =
cls()
        return _insta nce s[cls]
    return getins tance
@Singleton
class Counte r(o bject):
    def __init__(self):
        if not hasatt r(self
,'val'): self.val = 0
    def get(self): return
self.val
    def incr(s elf): self.val +=
1
```

Factory

```
@Singleton
class Player Fac tor y(o bject):
    def new(self, name, type):
        if type == 'peasant':
            return Peasan -
t(name)
        elif type == 'warrior':
            return Warrior (name)
        return None
```

Proxy

```
class CounterProxy(object):
    def __init__( self,
type):
        sel f.type =
type
    def incr(s elf):
        if self.type ==
'W':
            Cou -
nte r().incr()
        def get(self):
            Cou nte -
r().get()
```

Producer

Producer (cont)

```
> def run(self):
    while not self.terminate:
        sleep(0.1 * randint(0, 10)) # sleep
randomly
        self.notfull.acquire()
        while len(self.queue) >= self.maxsize: #
full
            print "full queue, waiting"
            self.notfull.wait()
        if self.terminate:
            break
        self.notfull.release()
        self.counter += 1
        self.queue.append(self.counter)
        self.notempty.acquire()
        self.notempty.notify() # notify consumer,
a new item
        self.notempty.release()
    def quit(self):
        self.terminate = True
        self.notfull.acquire()
        self.notfull.notify()
        self.notfull.release()
```

Observer

Observer (cont)

```
> class Person(Observer):
    def update(self, obj):
        print "heyo", obj.state()
a = Person()
b = Clock()
b.register(a)
b.tick()
```

facade

```
class AuthFacade(object):
    def __init__(self
,method):
        if method ==
'passwd':
            #
            elif method
=='oauth':
                #
                elif method
=='otp':
                    #
                    else:
                        -
throw ...
        def auth(s elf ,id ent -
ity ,data):
            #check authen -
tic ation based on setting
```

Consumer

```

class Producer(Thread):
    def __init__(self, queue,
cvs, maxsize=5):
        Thread.__init__(self)
        self.queue = queue
        self.maxsize = maxsize
        self.notfull = cvs[0]
        self.notempty = cvs[1]
        self.terminate =
False
        self.counter = 0

```

```

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 Consumer(Thread):
    def __init__(self, queue,
cvs):
        Thread.__init__(self)
        self.queue = queue
        self.notfull = cvs[0]
        self.notempty = cvs[1]
        self.terminate =
False
    def run(self):
        while not self.terminate:
            sleep(0.1 *
randint(0, 10)) # sleep randomly
            self.notify()
            while len(self.queue) == 0: # empty
                print "empty
queue, waiting"
                self.notify()
                if self.terminate:
                    break
            self.notify()
            item = self.queue[0]
            del self.queue[0]
            print "consumed ",
item

```



By **sercand**
cheatography.com/sercand/

Published 2nd December, 2016.
Last updated 2nd December, 2016.
Page 1 of 3.

Sponsored by **CrosswordCheats.com**
Learn to solve cryptic crosswords!
<http://crosswordcheats.com>

Consumer (cont)

```
> self.notfull.acquire()
    self.notfull.notify() # notify consumer, a
new item
    self.notfull.release()
def quit(self):
    self.terminate = True
    self.notempty.acquire()
    self.notempty.notify()
    self.notempty.release()
```

patterns

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.

Facade: Define a class implementing all details of the library/system and providing a simple uniform interface. Access the library through this interface.

You need to access a hard to duplicate, limited, probably old class definition. You don't 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.

Proxy: Write a class interface implementing functionalities missing in the original interface.

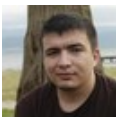
patterns (cont)

Objects depending on each others states need to be informed when the other encountered a change. Event handling systems.

Observer: Maintain a registry of observing objects in Subject object. When an event occurs, notify the observers.

Create objects from a set of classes. Implementation depends on class definitions. Introduction of new classes and other changes need recompilation. Class constructors exposed.

Factory: Use interface functions/objects to encapsulate class names and constructors. Use a interface functions, methods to provide you instances. Rest is handled by polymorphism.



By **sercand**
cheatography.com/sercand/

Published 2nd December, 2016.
Last updated 2nd December, 2016.
Page 2 of 3.

Sponsored by **CrosswordCheats.com**
Learn to solve cryptic crosswords!
<http://crosswordcheats.com>