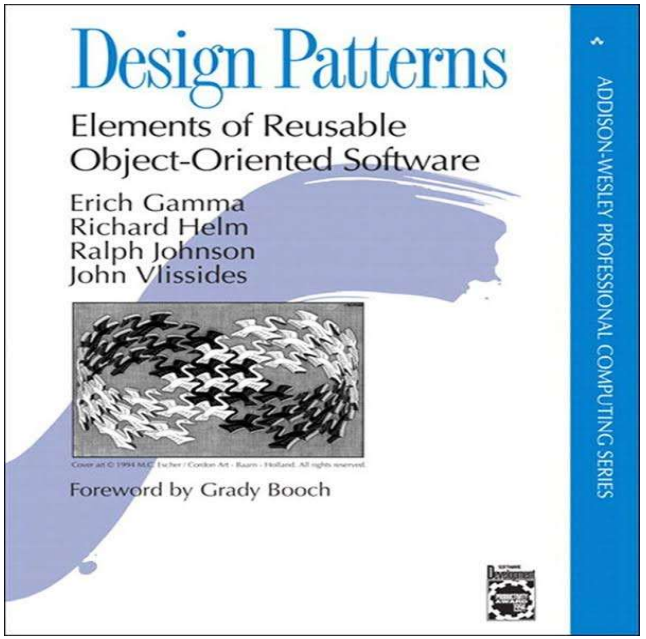
Bjarne Poulsen

SOFTWARE DESIGN & IMPLEMENTATION IN PYTHON

DESIGN PATTERNS



What is a design pattern?

A standard solution to a common programming problem

- A design or implementation structure that achieves a particular purpose.
- A high-level programming idiom.
- A technique for making code more flexible
- reduce coupling among program components
- Shorthand for describing program design
- A description of connections among program components (static structure)
- the shape of a heap snapshot or object model (dynamic structure)

SOLID

SOLID Principles is a coding standard that all developers should have a clear concept for developing software in a proper way to avoid a bad design. It was promoted by Robert C Martin and is used across the object-oriented design spectrum. When applied properly it makes your code more extendable, logical and easier to read.

SOLID is an acronym for 5 important design principles when doing OOP (Object Oriented Programming). ... The actual SOLID acronym was, however, identified later by Michael Feathers. The intention of these principles is to make software designs more understandable, easier to maintain and easier to extend.

https://www.slideshare.net/DrTrucho/python-solid

Single Responsibility Principle :

- A class should have one, and only one, reason to change.

Open-closed Principle :

- Entities should be open for extension, but closed for modification.

Liskov Substitution Principle :

- Objects in a program should be replaceable with instances of their base types without altering the correctness of that program. This means that clients are completely isolated and unaware of changes in the class hierarchy.

Interface Segregation Principle

- Many client-specific interfaces are better than one general-purpose interface.

Dependency Inversion Principle :

- High-level modules should not depend on low-level modules. Both should depend on abstractions.

Single Responsibility Principle in Python.

```
# Given a class which has two responsibilities
class Rectangle:
def __init__(self, width=0, height=0):
        self.width = width
        self.height = height

def draw(self):
# Do some drawing

def area(self):
return self.width * self.height
```

```
# We can split it into two...
class GeometricRectangle:
def __init__(self, width=0, height=0):
        self.width = width
        self.height = height

def area(self):
        return self.width * self.height

class DrawRectangle:
        def draw(self):
        # Do some drawing
```

Why Refactoring?

https://hackernoon.com/why-refactoring-how-to-restructure-python-package-51b89aa91987

Open-closed Principle

Let’s imagine you have a store, and you give a discount of 20% to your favorite customers using this class: When you decide to offer double the 20% discount to VIP customers. You may modify the class like this:

```
class Discount:
    def __init__(self, customer, price):
        self.customer = customer
        self.price = price

    def give_discount(self):
            if self.customer == 'fav':
                return self.price * 0.2
            if self.customer == 'vip':
                return self.price * 0.4
```

```
class Discount:
    def __init__(self, customer, price):
        self.customer = customer
        self.price = price

    def get_discount(self):
            return self.price * 0.2

class VIPDiscount(Discount):
    def get_discount(self):
        return super().get_discount() * 2
```

If you decide 80% discount to super VIP customers, it should be like this.

```
class SuperVIPDiscount(VIPDiscount):
    def get_discount(self):
        return super().get_discount() * 2
```

The Beginning of Patterns

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1994

This book defined patterns in three categories:

Creational patterns deal with the process of object creation

Structural patterns, deal primarily with the static composition and

structure of classes and objects

Behavioral patterns, which deal primarily with dynamic interaction among classes and objects

Creational Patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

– Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

– Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Why Study Patterns ?

Reuse tried, proven solutions:

- Provides a head start
- Avoids gotchas later (unanticipated things)
- No need to reinvent the wheel

Establish common terminology:

- Design patterns provide a common point of reference
- Easier to say, “We could use Strategy here.”

Provide a higher level prospective:

– Frees us from dealing with the details too early

Most design patterns make software more modifiable, less brittle:

– we are using time tested solutions

Using design patterns makes software systems easier to change:

—more maintainable

Helps increase the understanding of basic objectoriented design principles:

– encapsulation, inheritance, interfaces, polymorphism and SOLID.

Style for Describing Patterns

The structure that will be used is::

- Pattern name
- Recurring problem: what problem the pattern addresses
- Solution: the general approach of the pattern
- UML for the pattern

  Participants: a description as a class diagram
- Use Example(s): examples of this pattern, in Python

Pattern name: Singleton Design Pattern

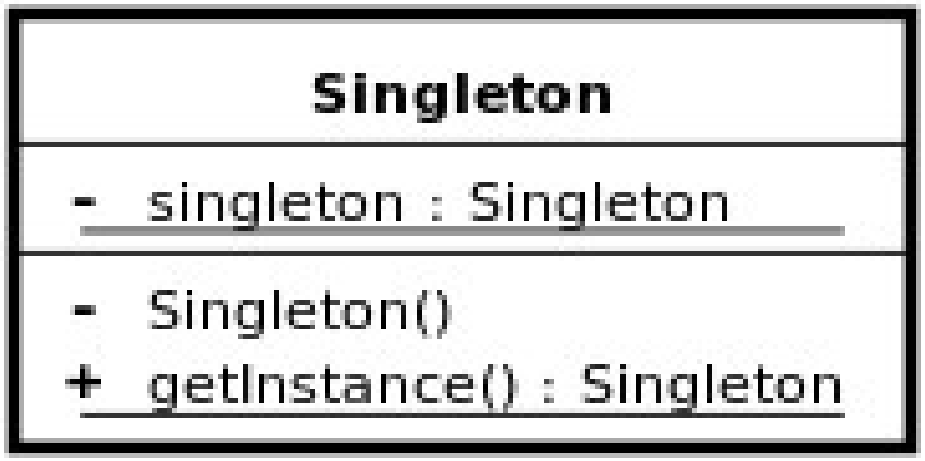
Recurring problem.

Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.

Definition:

The singleton pattern is a design pattern that restricts the instantiation of a class to one object.

Let’s see various design options for implementing such a class.


Singleton
- singleton : Singleton
- Singleton()
+ getInstance() : Singleton

```
class Singleton(type):
    """ Define an Instance operation that lets clients access its unique instance."""

    def __init__(cls, name, bases, attrs):
        super().__init__(name, bases, attrs)
        print("singleton")
        cls._instance = None

    def __call__(cls):
        if cls._instance is None:
            cls._instance = super().__call__()
        print("__call__")
        return cls._instance
```

```
def main():
    m1 = MyClass()
    print("m1")
    m2 = MyClass()
    print("m2")
    print(m1 is m2)
    MyClass()

if __name__ == "__main__":
     main()
```

```
Myclass
singleton
__call__
m1
__call__
m2
True
__call__
```

Make an example of using singleton design pattern 20 min.
A group should present their solution to the class?

https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Singleton.html

Dependency Injection.

What is dependency injection?

Instead of a class creating its own dependencies, its dependencies are inserted into it.

"Software is always under constant pressure to change, so it's very important to keep your code flexible. Inversion of control helps you break your application into pieces of manageable size, and then glue them back together in a much more flexible way. Then, the next time your client asks you to make a 'little' change to the product, you can laugh triumphantly instead of quivering in pain.“

-Nate Kohari(Creator of Ninject)

we need to touch on two SOLID principles.

Dependency Inversion Principle

Interface Segregation Principle

Manual Injection.

```
class A(object):
    def __init__(self):
        print("A")

class A1(A):
    def show(self):
        print("A1")


class A2(A):
    def show(self):
        print("A2")



class A3(A):
    def show(self):
        print("A3")
```

```
"""Dependency injection example, A1, A2 and A3 """

class B(object):
    """Example A1, A2 and A3."""
    def __init__(self, A):
        """Initializer."""
        self._A = A  # Type A is injected
```

```
"""Dependency injection example, Base & A."""
if __name__ == '__main__':
    A1_A = B(A1())
    A2_A = B(A2())
    A3_A = B(A3())
    A1_A._A.show()
```

```
A
A
A
A1
```

Inversion of Control (IoC) Container.

https://en.wikipedia.org/wiki/Inversion_of_control

https://pypi.org/project/dependency-injector/

Many Python libraries have the facilitate this:

- python-dependency-injector.
- Serum.
- Injector. https://pypi.org/project/Inject/2.0.0/

import dependency_injector.containers as con

import dependency_injector.providers as prov

```
class Aer(con.DeclarativeContainer):
    """IoC container of A providers."""

    A1O = prov.Factory(A1)

    A2O = prov.Factory(A2)

    A3O = prov.Factory(A3)
```

```
class Ber(con.DeclarativeContainer):
    """IoC container of B providers."""

    A1OiB = prov.Factory(B, Aer.A1O)

    A20iB = prov.Factory(B, Aer.A2O)

    A3OiB = prov.Factory(B, Aer.A3O)
```

```
if __name__ == '__main__':
    O1 = Ber.A1OiB()
    O2 = Ber.A20iB()
    O3 = Ber.A3OiB()
    print(type(O1))
    O1.show()
    print(type(O1._A))
    O1._A.show()
    print(type(Ber.A1OiB))
    print(type(Ber))
```

```
A
A
A
<class '__main__.B'>
B
<class '__main__.A1'>
A1
<class 'dependency_injector.providers.Factory'>
<class
'dependency_injector.containers.DeclarativeContainerMetaClass'>
```

Make an example of using dependency injection 30 min.
A group should present their solution to the class?

https://preslav.me/2018/12/20/dependency-injection-in-python/

The Observer Design Pattern

Define a one-to-many dependency between objects so that when an object changes it is state, all its dependents are notified and updated automatically.

Goal:

Communication without Coupling

Advantage:

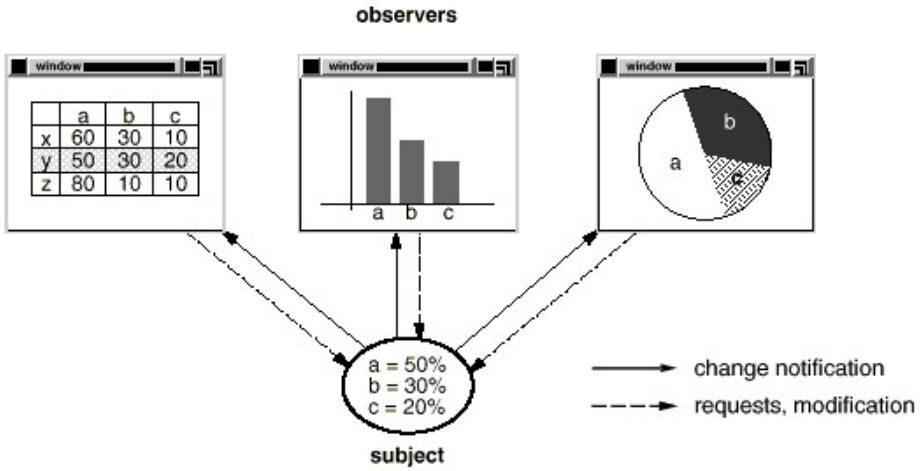
Makes each object easier to implement and maintain, more reusable, enabling flexible combinations.

Disadvantage:

Behavior is distributed across multiple objects; any change in the state of one object often affects many others.

Recurring problem:

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing



Solution:

The object that has some interesting state is often called subject, but since it's also going to notify other objects about the changes to its state, we'll call it publisher. All other objects that want to track changes to the publisher's state are called subscribers.

Task

Decouple a data model (subject) from “parties” interested in changes of its internal state

Requirements

- subject should not know about its observers
- identity and number of observers is not predetermined
- novel receivers classes may be added to the system in the future
- polling is inappropriate (too inefficient)

https://www.protechtraining.com/blog/post/tutorial-the-observer-pattern-in-python-879

Source

Make an example of using Observer patttern 20 min.

Assignment.

A design pattern must be selected for use in your program.

1. There must be a justification for choosing the design pattern.
2. There must be a class and sequence diagram describing the selected design pattern.