**Design Pattern – AddsOn**

Short description with sample code & eBook - Python Programming with Design Patterns:

- <https://www.oreilly.com/library/view/python-programming-with/9780137579921/>

- <https://github.com/Vondark/books/blob/master/docs/src/Python/Mastering-Python-Design-Patterns.pdf>

Here is the schematic representation of the **Design Patterns** categorized into **Creational, Structural, and Behavioral** patterns. Each category connects to the respective design patterns, visually illustrating their relationships. (See below figure)

A diagram of a design pattern

Description automatically generated

**1. Singleton Pattern**

Ensures that a class has only one instance and provides a global point of access.

**class Singleton:**

\_instance = None

def \_\_new\_\_(cls):

if cls.\_instance is None:

cls.\_instance = super(Singleton, cls).\_\_new\_\_(cls)

return cls.\_instance

# Usage

singleton1 = Singleton()

singleton2 = Singleton()

print(singleton1 is singleton2) # Output: True

**2. Factory Pattern**

Provides an interface for creating objects but allows subclasses to alter the type of objects that will be created.

class Animal:

def speak(self):

pass

class Dog(Animal):

def speak(self):

return "Woof!"

class Cat(Animal):

def speak(self):

return "Meow!"

class AnimalFactory:

@staticmethod

def get\_animal(animal\_type):

if animal\_type == "dog":

return Dog()

elif animal\_type == "cat":

return Cat()

else:

return None

# Usage

animal = AnimalFactory.get\_animal("dog")

print(animal.speak()) # Output: Woof!

**3. Observer Pattern**

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.

class Observer:

def update(self, message):

pass

class ConcreteObserver(Observer):

def \_\_init\_\_(self, name):

self.name = name

def update(self, message):

print(f"{self.name} received: {message}")

class Subject:

def \_\_init\_\_(self):

self.\_observers = []

def add\_observer(self, observer):

self.\_observers.append(observer)

def notify\_observers(self, message):

for observer in self.\_observers:

observer.update(message)

# Usage

subject = Subject()

observer1 = ConcreteObserver("Observer 1")

observer2 = ConcreteObserver("Observer 2")

subject.add\_observer(observer1)

subject.add\_observer(observer2)

subject.notify\_observers("Event occurred!")

# Output:

# Observer 1 received: Event occurred!

# Observer 2 received: Event occurred!

**4. Decorator Pattern**

Allows behavior to be added to an individual object dynamically.

def decorator(func):

def wrapper():

print("Before function execution")

func()

print("After function execution")

return wrapper

@decorator

def say\_hello():

print("Hello!")

# Usage

say\_hello()

# Output:

# Before function execution

# Hello!

# After function execution

**5. Strategy Pattern**

Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

class Strategy:

def execute(self):

pass

class ConcreteStrategyA(Strategy):

def execute(self):

return "Strategy A executed"

class ConcreteStrategyB(Strategy):

def execute(self):

return "Strategy B executed"

class Context:

def \_\_init\_\_(self, strategy):

self.strategy = strategy

def execute\_strategy(self):

return self.strategy.execute()

# Usage

context = Context(ConcreteStrategyA())

print(context.execute\_strategy()) # Output: Strategy A executed

context.strategy = ConcreteStrategyB()

print(context.execute\_strategy()) # Output: Strategy B executed