Návrhové vzory jsou obecná řešení často vyskytujících se problémů v architektuře softwaru. Nejedná se o knihovnu nebo přesný kus kódu, ale popsání postupu / šablona, jak problém řešit. Řeší různé problémy, jako je například omezení výkonu počítače.

## Creational Design Patterns

Snahou těchto návrhových vzorů je popsat postup výběru třídy nového objektu a zajištění správného počtu těchto objektů. Každý z nich nabízí možnosti zpřehlednění nebo zefektivnění kódu. Mezi tyto vzory patří např:

* Singleton – zajistí, že třída má pouze jednu instanci a poskytuje globální přístup k ní
* Object Pool
* Factory Metoda – vytváří objekty pomocí metody továrny, která deleguje vytváření na sub-třídy

### Object Pool

Využívá se v situaci, když je náročnější (vyžaduje více výkonu) vytvoření instance nějaké třídy. Object pool je jednoduše kontejner který obsahuje nějaký počet objektů dané třídy. Takže ve chvíli, kdy je objekt vypůjčen z pool stane se nedostupný až do té chvíle kdy je navrácen.

Na podobném principu funguje multiprocessing.Pool nebo multiprocessing.pool.ThreadPool v pythonu.

1. Výhody:
   1. Minimalizuje náklady na vytváření a ničení objektů, což zvyšuje výkon a snižuje paměťové nároky.
   2. Zlepšuje efektivitu systému tím, že poskytuje opakované využití objektů.
   3. Umožňuje omezit počet objektů vytvářených a používaných v systému, což pomáhá zabránit zahlcení paměti a zvyšuje stabilitu systému.
   4. Může pomoci zabránit problémům s konkurencí a zlepšit synchronizaci vláken v systému.
2. Nevýhody:
   1. Může být obtížné určit správnou velikost poolu objektů, což může vést k nadměrné spotřebě paměti nebo k nedostatku objektů.
   2. Může být složité synchronizovat vlákna v případě, že více vláken současně přistupuje k poolu objektů.
   3. Může vést k vytváření objektů, které nejsou využity, což může vést k zbytečné spotřebě paměti a snížení výkonu systému.

### Singleton

Návrhový vzor, který nám umožňuje vytvořit pouze jedinou instanci dané třídy.

Používáme ho např. v manažeru připojení k databázi nebo v místech, kde chceme globální správu pouze jednou entitou. Návrhový vzor singleton je doporučeno spíše nepoužívat.

1. Výhody:
   1. Zajišťuje, že třída má pouze jednu instanci v systému, což snižuje paměťové nároky a zvyšuje výkon.
   2. Poskytuje globální přístup k instanci třídy, což usnadňuje používání objektu v celém systému.
   3. Je vytvořen až ve chvíli kdy je poprvé potřeba
2. Nevýhody:
   1. Může být obtížné testovat třídu, protože singleton třídy jsou globálně přístupné.
   2. Singleton třídy mohou vést k vytváření tvrdých závislostí v kódu, a tedy zhoršit modularitu.
   3. Není jednoduché používat singlton ve více vláknovém prostředí, protože musíme se starat aby jednotlivá vláka nevytvořili singlton vícekrát.

# classic implementation of Singleton Design pattern

**class** Singleton:

    \_\_shared\_instance **=** 'GeeksforGeeks'

    @staticmethod

**def** getInstance():

        """Static Access Method"""

**if** Singleton.\_\_shared\_instance **==** 'GeeksforGeeks':

            Singleton()

**return** Singleton.\_\_shared\_instance

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

        """virtual private constructor"""

**if** Singleton.\_\_shared\_instance !**=** 'GeeksforGeeks':

**raise** Exception("This class is a singleton class !")

**else**:

            Singleton.\_\_shared\_instance **=** self

# main method

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # create object of Singleton Class

    obj **=** Singleton()

**print**(obj)

    # pick the instance of the class

    obj **=** Singleton.getInstance()

**print**(obj)

### Factory Method

Factory používáme, když máme několik tříd, implementujících stejný interface nebo, ze kterých můžeme udělat instance a chceme nechat rozhodnout program, která z těchto tříd bude nejlepší. Factory metodu také používáme, když chceme vytvářet instance z těchto tříd na základě určité business logiky nebo jiných podmínek.

1. Výhody
   1. Poskytuje rozhraní pro vytváření objektů, což umožňuje oddělení vytváření objektů od jejich použití.
   2. Umožňuje snadnou výměnu konkrétní implementace objektu bez změny kódu v klientovi.
   3. Zlepšuje modularitu kódu a udržitelnost.
2. Nevýhody:
   1. Vyžaduje vytvoření sub-tříd pro každou konkrétní implementaci objektu, což může vést k velkému počtu tříd v systému.
   2. Může být komplikované zjistit, kterou sub-třídu použít pro vytvoření správného objektu, pokud jich existuje více.

# Python Code for factory method

# it comes under the creational

# Design Pattern

**class** FrenchLocalizer:

    """ it simply returns the french version """

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

        self.translations **=** {"car": "voiture", "bike": "bicyclette",

                             "cycle":"cyclette"}

**def** localize(self, msg):

        """change the message using translations"""

**return** self.translations.get(msg, msg)

**class** SpanishLocalizer:

    """it simply returns the spanish version"""

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

        self.translations **=** {"car": "coche", "bike": "bicicleta",

                             "cycle":"ciclo"}

**def** localize(self, msg):

        """change the message using translations"""

**return** self.translations.get(msg, msg)

**class** EnglishLocalizer:

    """Simply return the same message"""

**def** localize(self, msg):

**return** msg

**def** Factory(language **=**"English"):

    """Factory Method"""

    localizers **=** {

        "French": FrenchLocalizer,

        "English": EnglishLocalizer,

        "Spanish": SpanishLocalizer,

    }

**return** localizers[language]()

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    f **=** Factory("French")

    e **=** Factory("English")

    s **=** Factory("Spanish")

    message **=** ["car", "bike", "cycle"]

**for** msg **in** message:

        print(f.localize(msg))

**print**(e.localize(msg))

        print(s.localize(msg))

## Structural design pattern

Structural Patterns představují skupinu návrhových vzorů zaměřujících se na možnosti uspořádání jednotlivých tříd nebo komponent v systému. Snahou je zpřehlednit systém a využít možností strukturalizace kódu.

* Fasáda –Ulehčuje komunikaci suživatelem.
* Proxy – prostředním v komunikaci

### Fasáda

Fasáda poskytuje jednotné rozhraní pro celou skupinu tříd, což umožňuje klientům jednodušší a srozumitelnější přístup k systému. Fasáda také skrývá komplexitu systému a izoluje klienta od jeho interní implementace.

1. Výhody:
   1. Poskytuje jednoduché a srozumitelné rozhraní pro komunikaci s komplexním systémem.
   2. Skrývá komplexitu systému a izoluje klienta od jeho interní implementace.
   3. Umožňuje snadné řízení přístupu k systému a umožňuje správu systému na jednom místě.
2. Nevýhody:
   1. Fasáda může být omezením pro pokročilejší uživatele, kteří potřebují více přístupu ke komponentám systému.
   2. Přidání dalších funkcí nebo komponent do systému může být složité a náročné na úpravu Fasády.

"""Facade pattern with an example of WashingMachine"""

**class** Washing:

    '''Subsystem # 1'''

**def** wash(self):

        print("Washing...")

**class** Rinsing:

    '''Subsystem # 2'''

**def** rinse(self):

**print**("Rinsing...")

**class** Spinning:

    '''Subsystem # 3'''

**def** spin(self):

        print("Spinning...")

**class** WashingMachine:

    '''Facade'''

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

        self.washing **=** Washing()

        self.rinsing **=** Rinsing()

        self.spinning **=** Spinning()

**def** startWashing(self):

        self.washing.wash()

        self.rinsing.rinse()

        self.spinning.spin()

""" main method """

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    washingMachine **=** WashingMachine()

    washingMachine.startWashing()

### Proxy

je strukturální vzor, který umožňuje vytvořit prostředníka mezi klientem a skutečným objektem. Jeho asi nejčastější použití je zapouzdření instance jiného objektu nebo přidání pomocné funkčnosti. Proxy nám tedy umožňuje řídit přístup ať už k celému či částečnému rozhraní objektu přes nějaký jiný zastupující objekt (proxy – zástupce, reprezentant).

* Remote Proxy – Proxy může být použit pro přístup k vzdálenému objektu, který je uložen na jiném serveru nebo počítači.
* Virtual Proxy – Proxy může být použit pro zpřístupnění velkého objektu, který zabírá mnoho paměti, ale části objektu se používají jen zřídka. Proxy objekt pak inicializuje jen ty části objektu, které jsou potřeba.
* Protection Proxy – Proxy může být použit pro kontrolu přístupu k objektu, když je potřeba ověřit, zda má klient oprávnění k provedení určité akce.
* Reverse Proxy – load balancer Reverse Proxy funguje na principu množiny serverů, jenž všechny jsou co se týče obsahu identické. Zátěž je rozdělena na základě zatížení jednotlivých serverů, přičemž může být orientována dle výkonu serveru, či jeho již stávajícího vytížení.

1. Výhody:
   1. Oddělení klienta od skutečného objektu umožňuje snadné dodání nové funkcionality bez nutnosti měnit kód klienta nebo skutečného objektu.
   2. Proxy může být použit pro optimalizaci výkonu, když není nutné inicializovat skutečný objekt, dokud není skutečně potřeba.
   3. Proxy umožňuje snadné řízení přístupu k objektu, když je potřeba zabezpečit určitou funkci nebo data.
2. Nevýhody:
   1. Přidání dalšího prostředníka mezi klienta a skutečný objekt může snížit výkon a zpomalit systém.
   2. Komunikace přes proxy může způsobit komplikace v případě, že objekt vyžaduje okamžitou odpověď, protože proxy zpomaluje přístup k objektu.

**class** College:

    '''Resource-intensive object'''

**def** studyingInCollege(self):

**print**("Studying In College....")

**class** CollegeProxy:

    '''Relatively less resource-intensive proxy acting as middleman.

     Instantiates a College object only if there is no fee due.'''

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

        self.feeBalance **=** 1000

        self.college **=** None

**def** studyingInCollege(self):

        print("Proxy in action. Checking to see if the balance of student is clear or not...")

**if** self.feeBalance <**=** 500:

            # If the balance is less than 500, let him study.

            self.college **=** College()

            self.college.studyingInCollege()

**else**:

            # Otherwise, don't instantiate the college object.

**print**("Your fee balance is greater than 500, first pay the fee")

"""main method"""

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # Instantiate the Proxy

    collegeProxy **=** CollegeProxy()

    # Client attempting to study in the college at the default balance of 1000.

    # Logically, since he / she cannot study with such balance,

    # there is no need to make the college object.

    collegeProxy.studyingInCollege()

    # Altering the balance of the student

    collegeProxy.feeBalance **=** 100

    # Client attempting to study in college at the balance of 100. Should succeed.

    collegeProxy.studyingInCollege()

## Behavior patterns

tyto vzory se používají pro definování interakcí mezi objekty a třídami v systému

* Iterátor –Umožňuje nám procházet prvky v různých strukturách.
* Null Object –Nahrazuje null hodnotu.

### Iterátor

je návrhový vzor, který umožňuje procházet prvky kolekce bez ohledu na způsob, jakým jsou uloženy. Iterátor poskytuje jednotné rozhraní pro přístup k prvkům kolekce, což umožňuje snadnou iteraci a práci s daty. Díky tomuto vzoru můžeme jednoduše procházet prvky kolekce bez znalosti vnitřního mechanismu, který je použit pro uložení dat.

1. Výhody:
   1. Umožňuje snadnou iteraci a práci s daty bez nutnosti znalosti vnitřního mechanismu kolekce
   2. Jednoduše se používá v kombinaci s dalšími návrhovými vzory, jako je například Fasáda nebo Strategie
2. Nevýhody:
   1. Může být pomalý, pokud se používá s velkými datovými strukturami
   2. Nelze použít na některé specifické datové struktury, které nejsou iterovatelné, například stromy nebo grafy. V takových případech je třeba použít jiné návrhové vzory, jako je například Prohlížeč (Visitor) nebo Návštěvník (Traversal).

**class MyIterator:**

**def \_\_init\_\_(self, data):**

**self.\_data = data**

**self.\_index = 0**

**def \_\_iter\_\_(self):**

**return self**

**def \_\_next\_\_(self):**

**if self.\_index >= len(self.\_data):**

**raise StopIteration**

**value = self.\_data[self.\_index]**

**self.\_index += 1**

**return value**

**numbers = [1, 2, 3, "Ahoj", 5]**

**my\_iterator = MyIterator(numbers)**

**for num in my\_iterator:**

**print(num)**

### Null Object

umožňuje nahradit null hodnoty objektům, které implementují stejný rozhraní, jako jsou ostatní objekty. Tento vzor poskytuje objekt s výchozími hodnotami, který může být použit místo null hodnoty, a to bez nutnosti vytvářet složitou logiku pro kontrolu null hodnot. Tento vzor pomáhá minimalizovat problémy s chybějícími hodnotami a umožňuje snadnější testování kódu a eliminovat většinu problémů sním spojeným, jako NullPointerException a podobné.

1. Výhody:
   1. Minimalizuje problémy s chybějícími hodnotami a zlepšuje stabilitu aplikace
   2. Umožňuje snadnější testování kódu
   3. Snadno se používá v kombinaci s dalšími návrhovými vzory, jako je například Iterátor
2. Nevýhody:
   1. Může zvýšit nároky na paměť, protože objekty s výchozími hodnotami mohou být vytvářeny často
   2. Může být náročné implementovat u složitějších objektů

**class RealObject():**

**def do\_something(self):**

**print("Doing something!")**

**class NullObject():**

**def do\_something(self):**

**pass**

**def create\_object(flag):**

**if flag:**

**return RealObject()**

**else:**

**return NullObject()**

**my\_object = create\_object(True) # Vytvoří reálný objekt**

**my\_object.do\_something() # Vypíše "Doing something!"**

**my\_object = create\_object(False) # Vytvoří null objekt**

**my\_object.do\_something() # Nic se nestane, protože null objekt nemá žádnou funkcionalitu**