1 Design Patterns
Factory Method oraz Abstract Factory

1. Czym są wzorce projektowe?

Wzorce projektowe (ang. design patterns) to najprościej - ustalone, uniwersalne i sprawdzone w praktyce rozwiązania często pojawiających się problemów projektowych. Przyznaj, ile razy zdarzyło Ci się stworzyć ciężko rozszerzalny i po prostu źle przemyślany projekt aplikacji. Powodem takiego wypadku przy pracy jest zazwyczaj niewzięcie wszystkich czynników pod uwagę czy przekombinowane i nieoptymalne rozwiązania.

I to właśnie w momencie, gdy poszukujemy jak najbardziej efektywnego sposobu na rozwiązanie problemu, na horyzoncie pojawiają się wzorce projektowe. Są to niejako strategie, którymi powinien kierować się programista, aby mieć pewność, że dobrze projektuje swój program.
Zapytasz teraz pewnie, czy to znaczy, że jeżeli wcześniej nie używałem żadnych wzorców w swoich programach, to były one z góry źle projektowane? Z góry? Na pewno nie!

Nie jest powiedziane, że wzorce to jest niejako punkt konieczny do umieszczenia w swoich programach. Mają one służyć jedynie jako pomoc, drogowskaz, więc nie powinny być nieodłącznym elementem programu.

Założę się również, że w wielu przypadkach nieświadomie reimplementowałeś istniejące już wzorce! Bądź świadomy tego, że istnieje naprawdę wiele wszelakich wzorców projektowych i żaden programista nie zna ich wszystkich. A to wcale nie dlatego dlatego, że opierają się one na jakiejś górnolotnej, ciężkiej do opanowania

filozofii. Są one po prostu na tyle proste, że programiści korzystają z nich nieświadomie.

Jednak są też pewne wzorce projektowe, na które już ciężko wpaść mimowolnie i to właśnie to szkolenie rozpocznie serię ich omawiania. Nawet jeśli nigdy nie natkniemy się na problemy, które te wzorce opisują, to ich znajomość nadal się przydaje, bo uczą one jak poradzić sobie z wieloma ciężkimi do rozgryzienia sytuacjami w kodzie.

Na przykładzie - w mojej codziennej pracy jako programista, często pojawiają się sformułowania typu: “O, tutaj warto użyć Fabrykę lub wzorzec Fasady”. I wtedy każdy sprawny programista z zespołu, będzie wiedział, jak rozwiązać dany problem. Nie trzeba wtedy tłumaczyć całej koncepcji, tylko bezpośrednio usprawniamy pracę.

Zbierając więc wszystkie zalety, dzięki wzorcom otrzymujemy:

- Pełną zgodność z zasadami SOLID
- Łatwość komunikacji między programistami
- Możliwość łatwego utrzymania i rozwijania kodu

Już wyobrażam sobie entuzjazm na Twojej twarzy i gotowość do implementowania wzorców projektowych. Mega mnie to cieszy, ale muszę Cię tylko troszeczkę ostudzić. Bądź świadomy tego, iż nie warto na siłę umieszczać w swoim programie pozornie pasujących wzorców projektowych.

Zanim zdecydujesz się na zastosowanie któregoś, musisz zrobić solidny rachunek sumienia i dogłębnie zastanowić się, czy wzorzec nie skomplikuje logiki programu.

Łatwo bowiem postrzegać wzorce projektowe jako, tzw. golden hammery - czyli rozwiązania na wszystkie bóle i gorączki tego świata. Mając przecież takie młotki, wszystko, niekoniecznie słusznie, może wydawać się gwoździem.

Na dowód tego, co mówię - zobacz, jak można łatwo utrudnić napisanie prostego programu Hello World, na siłę implementując różne wzorce :)

code4craft/hello-design-pattern: Hello world using all 23 kinds of GoF design patterns.

2. Klasyfikacja wzorców

W zależności od tego, jaką kategorię problemu rozwiązuje określony design pattern, możemy przypisać go do jednej z 3 grup:

- **Wzorce kreacyjne** - opisują, w jaki sposób tworzone są obiekty
- **Wzorce behawioralne** - opisują, w jaki sposób zachowują się obiekty
- **Wzorce strukturalne** - opisują sposób, w jaki obiekty są zbudowane

Lista implementacji zaliczanych do wyżej wymienionych kategorii jest naprawdę długa i obszerna, dlatego potraktuj poniższą tabelę, jako prosty cheatsheet, aniżeli coś, co trzeba mieć wzorowo opanowane.

Kreacyjne	**Behawioralne**	**Strukturalne**
Factory Method Abstract Factory Prototype Singleton Builder Lazy initialization	Chain of Responsibility Command Interpreter Observer Strategy Visitor	Adapter Bridge Decorator Proxy Facade Composite

Tak jak więc widzisz - potencjalnego materiału do nauki jest sporo, jednak głupotą byłoby opanowywanie wszystkiego powyżej. Tym bardziej, że z ideą, choćby wzorca Composite, na pewno miałeś nieświadomie już do czynienia (zagadnienie kompozycji).

Kolejne wzorce projektowe będziemy omawiali wraz z następnymi szkoleniami, a w tym dokumencie skupimy się na wzorcu Factory Method oraz Abstract Factory z grupy kreacyjnych patternów.

1. Factory Method

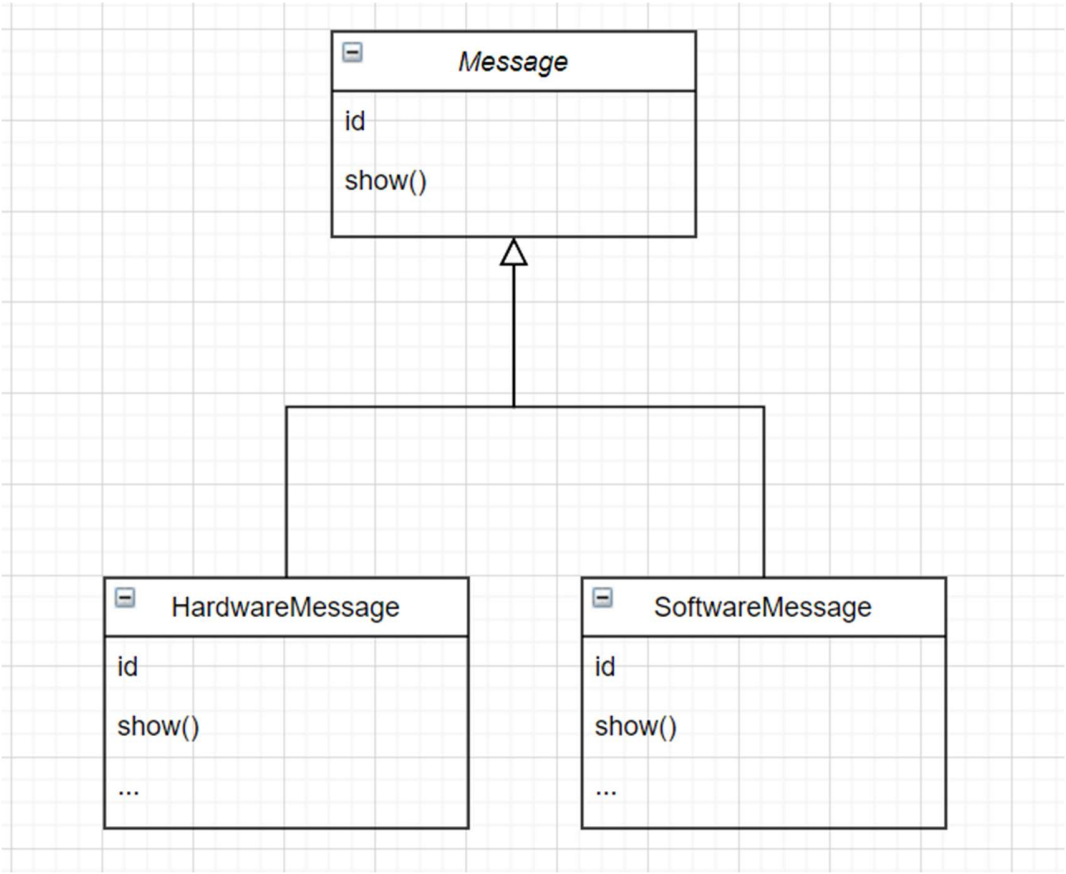
Wzorzec ten wykorzystywany jest w momencie, gdy chcemy stworzyć metodę, która dostarczać nam będzie obiekty konkretnych klas.

Jaka to będzie klasa, zależeć będzie od, np. wartości argumentu, jaki zostanie

przesłany do metody fabrykującej. Takim dość życiowym przykładem, w którym mógłby sprawdzić się ten wzorzec, jest sytuacja, w której system musi zadecydować, jakiego typu komunikat wyświetlić użytkownikowi.

Załóżmy, że do wyboru ma dwa rodzaje wiadomości: HardwareMessage oraz SoftwareMessage. Omawiany wzorzec mógłby się tutaj sprawdzić w taki sposób, że po tym, jak metoda produkująca (fabryka) otrzymałaby argument w postaci kodu błędu, np. 0xffff (błąd związany z dyskiem twardym) lub 0x0000 (błąd związany z zagłodzeniem procesu), to decydowałaby, jaką wiadomość (w postaci obiektu) ma w danym momencie utworzyć.

Diagram UML powyższych klas mógłby wyglądać coś w podobie do:



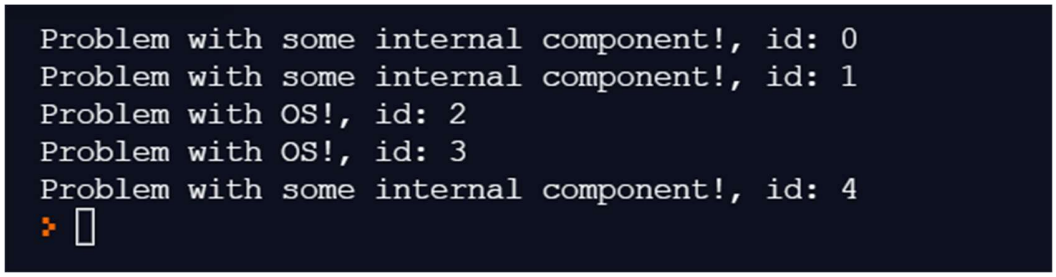
Implementacja:

```
import random
from abc import ABC

class Message(ABC):
    counter = 0
```

```
    def __init__(self):
      self.id = Message.counter
      Message.counter += 1

    def show(self):
      raise NotImplementedError

class HardwareMessage(Message):
 def __init__(self):
    super().__init__()
    self.show()

 def show(self):
    print(f"Problem with some internal component!, id: {self.id}")


class SoftwareMessage(Message):
 def __init__(self):
    super().__init__()
    self.show()

 def show(self):
    print(f"Problem with OS!, id: {self.id}")


def error_factory(is_error:bool = True):
 error_types = {b"0xf" : HardwareMessage, b"0x0" : SoftwareMessage}

 if is_error:
    rand_error = random.choice(list(error_types.keys()))
    error_types.get(rand_error)()


def main():
 for i in range(5):
    error_factory()

if __name__ == "__main__":
```

```
main()
```

Przykładowy wynik:



Całym clue powyższego kodu jest na pewno funkcja error_factory, której w zasadzie nazwa wzorca dotyczy (Factory **Method**). Zauważ, jak racjonalnie wykorzystaliśmy omawiany wzorzec i dzięki temu sprawnie rozwiązaliśmy problem implementacyjny.

W powyższym przykładzie losujemy akurat rodzaj błędu, jaki ma nastąpić (co jest dość niepraktyczne), ale myślę, że efektywnie nakierunkowuje Cię na całą ideę rozwiązania.

Jeżeli przykład z błędami jest dla Ciebie wciąż dość abstrakcyjny i niepraktyczny, to rozważ logikę biznesową bankowego serwisu internetowego.

Możemy wyróżnić w nim, np. 3 rodzaje kont bankowych: konto osobiste, konto firmowe oraz konto oszczędnościowe. Czyż niepraktyczne okazałoby się wykorzystanie Factory Method i stworzenie logiki odpowiadającej za budowanie obiektu określonego rodzaju konta w zależności od wyborów klienta przy rejestrowaniu konta?

Zadanie: Spróbuj odwzorować w kodzie powyższą implementację kont bankowych. Wykorzystaj klasę abstrakcyjną, metodę fabrykującą oraz utwórz symulacje trzech metod obiektów: validateUserIdentity(), calculateInterestRate(), registerAccount(). Symulacja ma polegać jedynie na wyświetlaniu określonych komunikatów.

Zalety Factory Method:

- **Brak ścisłego powiązania między produktem (obiektem klasy tworzonej) a kreatorem (czymś, co tworzy ten obiekt)**
- **Realizowanie drugiej zasady SOLID - Open/Closed Principle.**
  Wracając do ww. przykładu, dodawanie nowych rodzajów błędów sprowadza się jedynie na umieszczeniu nowej pary klucz : wartość w słowniku error_types. Brak żadnej ifologii i bezpośredniego sprawdzania parametrów.
- **Single Responsibility Principle.**
  Możemy jasno wydzielać funkcjonalności programu pomiędzy poszczególnymi częściami kodu. Nie tworzymy żadnej uber-struktury, która łączy wszystkie funkcjonalności programu w jedną całość.

Wady Factory Method:

- Kod może się komplikować w momencie, gdy chcesz utworzyć większą ilość klas i okiełznać ich produkcję. Dlatego też method factory jest często rozwijana do bardziej skomplikowanych wzorców, jak np. Abstract Factory, Builder lub Prototype. Wzorzec projektowy Abstract Factory omówimy w dalszej części tego szkolenia.

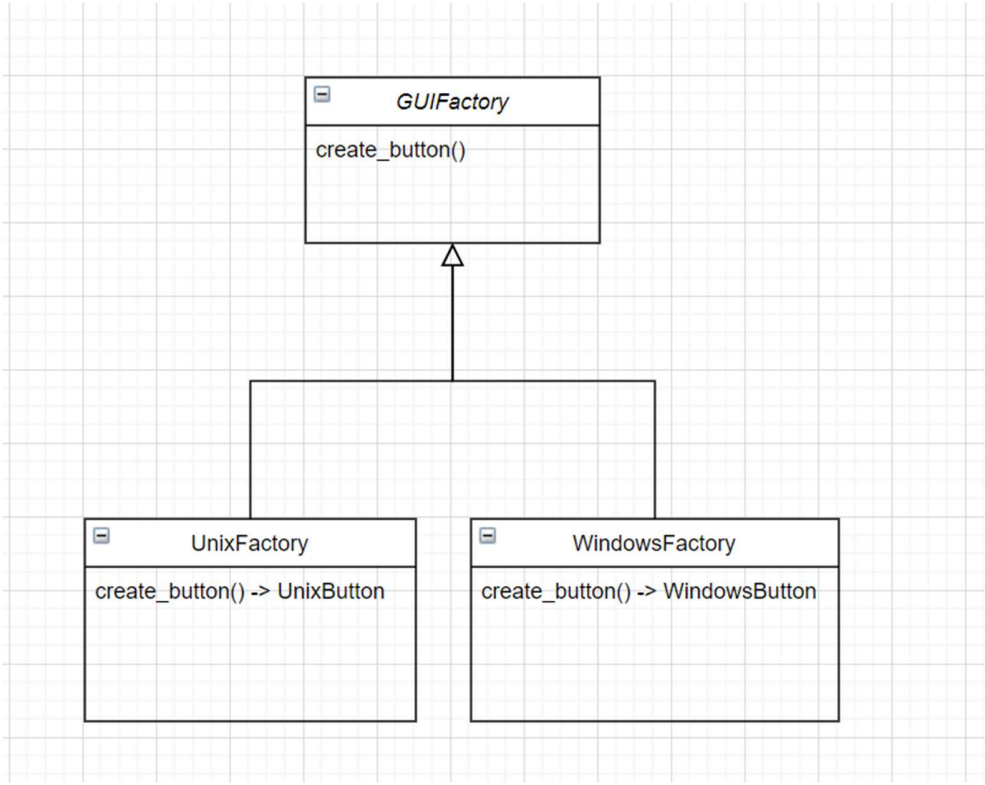
2. Abstract Factory

Od strony ideologicznej i zasady działania, abstract factory jest bardzo zbliżone do factory method, z tą różnicą, że umożliwia łatwiejsze tworzenie większej ilości spokrewnionych ze sobą typów obiektów. A to dlatego, iż ten kreacyjny wzorzec kładzie nacisk na tworzenie **klas fabryk** dla określonych typów produktów.

Tak więc, odnosząc się do przykładu z produkowaniem różnych rodzajów wiadomości, do Naszego programu dodalibyśmy dwie nowe klasy - fabrykę **HardwareMessageFactory** oraz **SystemMessageFactory**, które, jak się zapewne domyślasz, produkowałyby (zwracały) odpowiedniego rodzaju wiadomości.

W następnym przykładzie nie będę jednak reimplementował wcześniejszego zadania, a przedstawię Ci wizję zupełnie nowego problemu, w którym zastosujemy Abstract Factory.

A zadanie jest proste - stworzyć fabryki produkującą odpowiednie komponenty interfejsu użytkownika (np. Button), w zależności od tego, czy jest on operatorem systemu Unix-owego czy też Windows-owego.




```
from abc import ABC

class Button(ABC):
 def render(self):
   raise NotImplementedError


class UnixButton(Button):
 def render(self):
   print("Rendering an Unix button...")


class WindowsButton(Button):
 def render(self):
   print("Rendering a Windows button...")
```

```
class GUIFactory(ABC):
 def create_button(self):
   raise NotImplementedError


class UnixFactory(GUIFactory):
 def create_button(self) -> UnixButton:
   return UnixButton()


class WindowsFactory(GUIFactory):
 def create_button(self) -> WindowsButton:
   return WindowsButton()


def main():
 config = {"Windows" : WindowsFactory, "Unix" : UnixFactory}
 system_name = "Windows" # or = "Unix"
 factory = config.get(system_name)()
 button = factory.create_button()
 button.render()

 system_name = "Unix"
 factory = config.get(system_name)()
 button = factory.create_button()
 button.render()


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

Wynik:

Rendering a Windows button…
Rendering an Unix button...

Po przeanalizowaniu powyższego kodu, chyba nie muszę wymieniać, jakie są zalety korzystania z Abstract Factory. W zasadzie otrzymujemy te wszystkie pozytywne cechy, co w przypadku factory method, ale dodatkowo mamy możliwość implementowania bardziej skomplikowanej i złożonej logiki, nie tracąc równocześnie na jakości kodu.

Jednak miej równocześnie gdzieś z tyłu głowy, aby nie przesadzić z interfejsami (klasami abstrakcyjnymi) dla poszczególnych klas. Łatwo bowiem niepotrzebnie skomplikować swój program o własnie tak zbędne abstrakcje.