|  |  |
| --- | --- |
| STRATEGY PATTERN | * Używanie kompozycji a nie dziedziczenia * Wprowadza się abstrakcyjne "strategie", które potem są implementowane w konkretne wersje algorytmów.   Przykład:  Mamy kaczki z których każda musi kwakać. Zamiast tworzyć metodę quack() w superklasie Duck z której dziedziczą wszystkie inne kaczki, tworzymy np. Interfejs albo Subklasę IQuackStrategy. W interfejsie IQuackStrategy jest abstrakcyjna metoda quack() i potem implementacje np: SimpleQuackStrategy i LoudQuackStrategy z których każda będzie miała własną implementację quack();  <IQuackStrategy>  {quack()}  \*  |  /\  / \  SimpleQuackStrategy LoudQuackStrategy  { @Override quack() } { @Override quack() }   * Główna klasa Duck powinna wtedy posiadać pole IQuackStrategy w które w konstruktorze wstrzykniemy odpowiednią implementację tego interfejsu. * Dzięki temu możemy np. w trakcie trwania programu podmienić implementację danej abstracji w naszym obiekcie. |
| FACTORYPATTERN | * Wzorzec do tworzenia klasa posiadającej metodę do automatycznego tworzenia instancji obiektów. |
| FACTORY METHOD | * Korzysta z interface’u * Umożliwia podklasom zmianę typu obiektu tworzonego   <Logistics>  + createTransport() o- Transport t = createTransport()  + planDelivery()  ^  \_\_\_|\_\_\_  | |  RoadLogistics SeaLogistics  @Override + createTransport() @Override + createTransport()  return new Track() return new Ship()  Deklaracja typu zwracanego za pomocą interface’u, z implementowaną metodą:  <Transport>  + deliver()  ^  \_\_\_|\_\_\_  | |  Track Boat  { @Override + deliver() } { @Override + deliver() }  **Możliwość zastosowania:**  Nie wiemy jakiego typu będzie obiekt z który będzie pracował kod:   * Oddzielamy kod konstrucyjny produktu od kodu, który produkt używa, dzięki czemu możemy go łątwo rozszerzyć.   Jeśli chcesz zaoszczędzić zasoby systemowe wykorzystując wcześniej utworzony obiekt   * Ważne w przypadku dużych projektów * Chroni kod przed duplikatami obiektów * Dobrym miejscem jest konstruktor klasy, który nie powinien zwrócić instancji obiektu już istniejącego   **Zalety i wady:**   * Unikasz ścisłego połączenia między twórcą a konkretnymi produktami. * Zasada pojedynczej odpowiedzialności. Możesz przenieść kod tworzenia produktu w jedno miejsce w programie, co ułatwia obsługę kodu. * Zasada otwarta / zamknięta. Możesz wprowadzać nowe typy produktów do programu bez zrywania istniejącego kodu klienta. * Kod może stać się bardziej skomplikowany, ponieważ musisz wprowadzić wiele nowych podklas, aby zaimplementować wzorzec. Najlepszym scenariuszem jest wprowadzenie wzorca do istniejącej hierarchii klas twórców. |
| ABSTRACT FACTORY | Pozwala tworzyć rodziny obiektów bez określenia konkretnej klasy  <Chair>  + hasLegs()  + sitOn()  ^  \_\_\_|\_\_\_  | |  VictorianChair ModernChair  @Override + hasLegs() @Override + hasLegs()  @Override + sitOn() @Override sitOn()  <FurnitureFactory>  + createChair() : Chair  + createTable() : Table  + createSofa() : Sofe  ^  \_\_\_|\_\_\_  | |  VictorianFurnitureFactory ModernFurnitureFactory  @Override + createChair() : Chair @Override + createChair() : Chair  @Override + createTable() : Table @Override + createTable() : Table  @Override + createSofa() : Sofe @Override+ createSofa() : Sofe  Wzorzec tworzy osobny wariant |