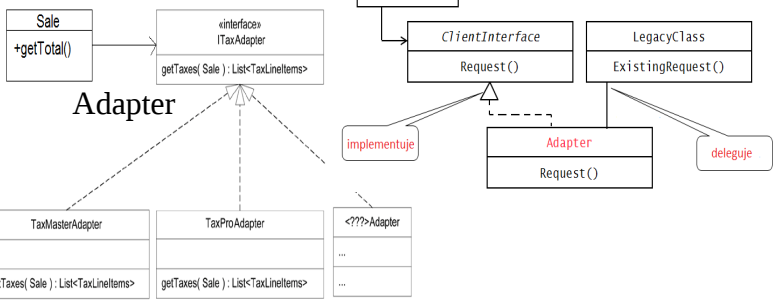
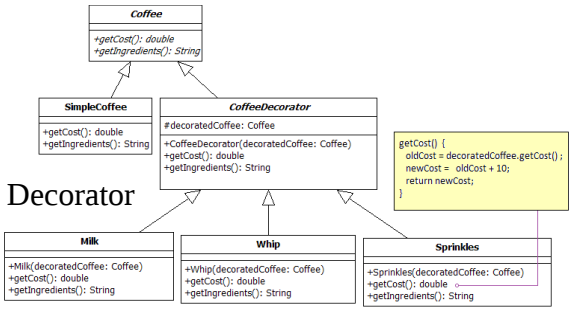
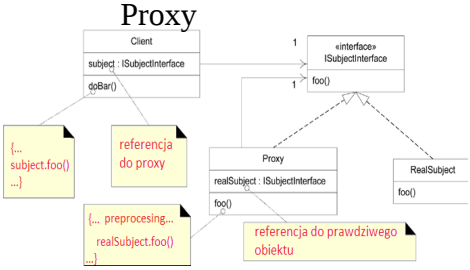


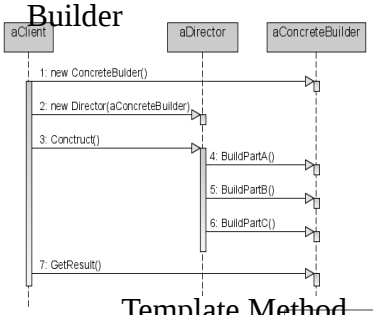
Adapter



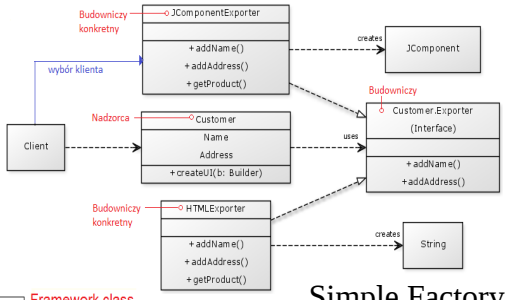
Decorator

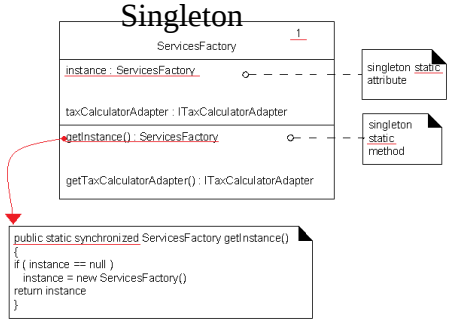


Proxy

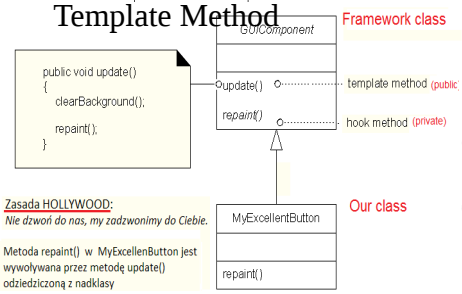


Builder

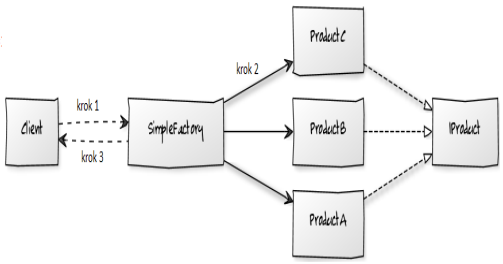




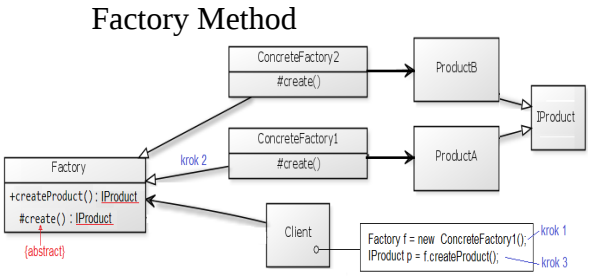
Singleton



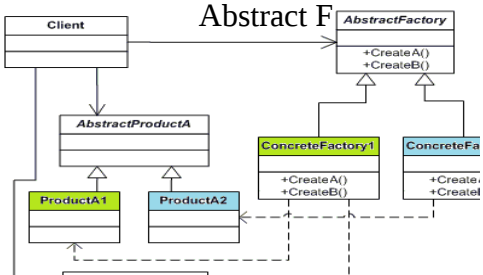
Template Method



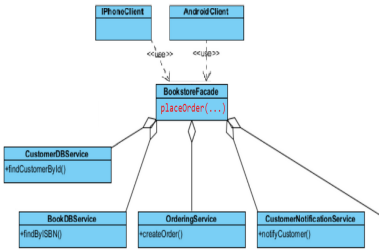
Simple Factory



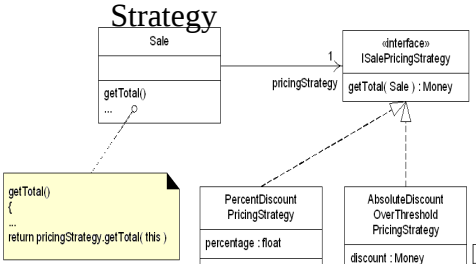
Factory Method



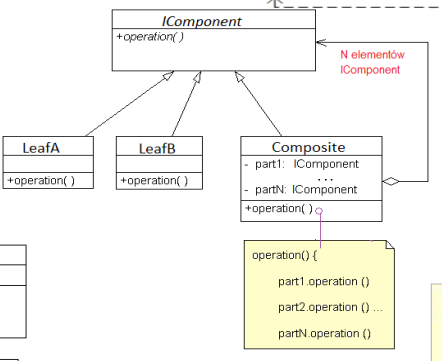
Abstract F



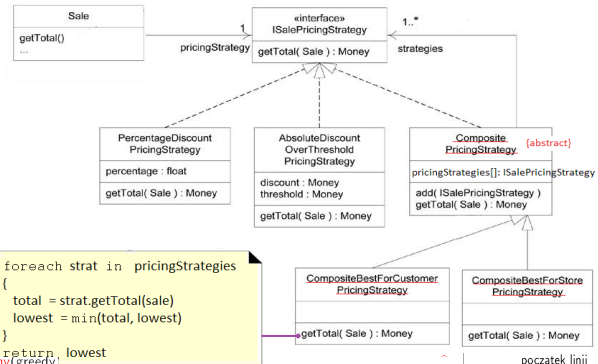
Facade

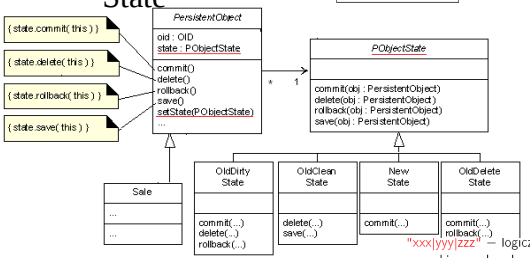


Strategy



Composite





State

```
foreach strat in pricingStrategies
{
  total = strat.getTotal(sale)
  lowest = min(total, lowest)
}
return lowest
```


- Literały: ciągi znaków, np. "abc"
- \ ^ $. | ? * + () [] {} mają specjalne znaczenie, by użyć ich jako literałów potrzebny znak wyjścia \

```
Wzorzec: "(x"
Tekst: "(x"
Exception: unclosed group near index 2
```

```
Wzorzec: "(x)"
Tekst: "(x)"
matches(): cały tekst nie pasuje do wzorca.
find(): dopasowano podłańcuch "x"
```

"xxx|yyy|zzz" – logiczna alternatywa, wyszukiwany dowolny z "xxx","yyy","zzz"

Nawiasy kwadratowe [...] definiują klasy znaków

Np. [ab.] jest równoważne wyrażeniu a|b|\(czyli oznacza jedną z wymienionych liter

Negacja klasy – jeśli pierwszym znakiem w klasie jest ^ dopasowanie nastąpi dla każdego znaku poza wymienionymi, [^abc] to każdy poza a, b i c

? – jeden raz lub wcale
* – zero lub więcej razy
+ – raz lub więcej razy
{n} – dokładnie n razy
{n,} – co najmniej n razy
{n, m} – co najmniej n ale nie więcej niż r

Domyślnie kwantyfikator jest zachłanny(greedy) – sprawdza dopasowanie do całego tekstu, jeśli nie ma, usuwa kolejne znaki z końca tekstu

Kwantyfikator można uczynić wstrzemięźliwym (reluctant) lub zaborczym (possesive) poprzez dopisanie po kwantyfikatorze znaku ? lub +

Wstrzemięźliwy – rozpoczyna od początku tekstu i pobiera znak po znaku aż dopasuje

Zaborczy – sprawdza zgodność wyrażenia z całym tekstem (wszystko albo nic)

t: "yyyy XXX yyy XXX"

Wzorzec: .*XXX
matches(): cały tekst pasuje do wzorca
find(): "yyyy XXX yyy XXX" od pozycji 0

Wzorzec: .*?XXX
matches(): cały tekst pasuje do wzorca
find(): dopasowano "yyyy XXX" od pozycji 0
find(): dopasowano " yyy XXX" od pozycji 8

Wzorzec: .*+XXX
matches(): cały tekst NIE pasuje do wzorca
find(): nie znaleziono żadnego wystąpienia wzorca

\d = [0 – 9],	\D = [^\d]
\s = [\t\n\x0B\f\r],	\S = [^\S]
\w = [a – zA – Z0 – 9],	\W = [^\w]
\p{Lower} = [a – z],	\p{Upper} = [A – Z]
\p{Punct} =!" #$%&' ( ) * +, -./ :; <=>?@ [\]^_ {|} ~	
\p{ASCII} - znak ASCII	
\p{InNazwaBlokuUnicode} - np. polskie znaki są w \p{InLatinExtended–A}	

^	początek linii
$	koniec linii
\b	na granicy słowa
\B	nie na granicy słowa
\A	początek wejścia
\G	koniec poprzedniego dopasowania
\Z	koniec wejścia

Expert
Creator
Low Coupling
High Cohesion
Polymorphism
Indirection
Pure Fabrication
Protected Variations
Controller

Model dziedziny – asocjacje

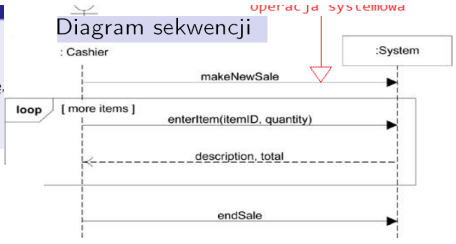
Asocjacja to relacja między instancjami klas opisująca ważne powiązanie

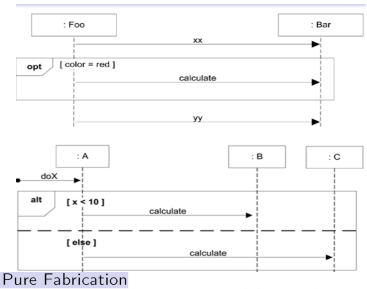
Agile Manifesto (2001, USA)

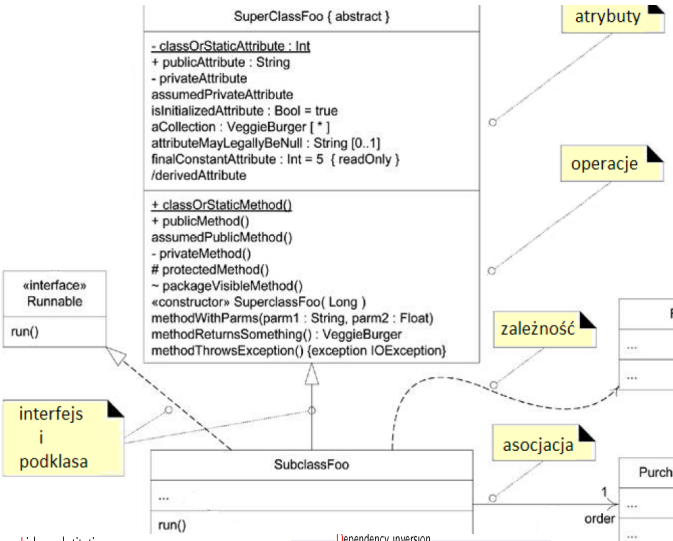
Przedkładamy:

- ludzi i interakcje ponad procesy i narzędzia,
- działające oprogramowanie ponad obszerną dokumentację,
- współpracę z klientem ponad formalne ustalenia,
- reagowanie na zmiany ponad podążanie za planem.

Diagram sekwencji







Przydziel zobowiązanie „ekspertowi" - klasie, która ma informacje konieczne do jego realizacji

metody-polecenia – wykonują działanie, nie zwracają wartości

metody-zapytania – zwracają dane, nie zmieniają stanu obiektu

Nową instancję A powinna tworzyć klasa B, która:

- zawiera, pamięta lub bezpośrednio używa A
- posiada dane inicjalizacyjne dla A

Sprzężenie - miara określająca w jakim stopniu dana klasa jest zależna od innych klas, bibliotek

Problemy z wysokim sprzężeniem klasy A:

- zmiany w innych klasach wymuszają zmiany A
- trudno jest „zrozumieć" klasę A w izolacji
- trudno jest powtórnie użyć klasę A, bo potrzebne są też obiekty z nią powiązane

Spójność - miara określająca w jakim stopniu metody klasy są do siebie podobne i ze sobą powiązane

Niska spójność powoduje, że:
trudno zrozumieć kod i cel istnienia klasy
trudno klasę utrzymywać, rozwijać i ponownie wykorzystać

Pure Fabrication

Problem: ograniczenie się w modelu projektowym do obiektów z modelu dziedziny może prowadzić do niskiej spójnością i wysokiego sprzężenia

Pomysł: wymyśl klasę pomocniczą, spoza modelu dziedziny i przydziel jej odpowiednie zobowiązania

Pośrednictwo

Unikaj bezpośrednich powiązań do elementów potencjalnie niestabilnych. W razie potrzeby wykorzystaj obiekt pośredniczący.

Polimorfizm – w praktyce oznacza nadanie tej samej nazwy metodom różnych klas, gdy metody te są związane ze sobą:

overriding – klasy mają wspólny interfejs lub nadklasę, metody mają taką samą sygnaturę (*głównie o tym myślimy*)

overloading – metody mają tą samą nazwę ale w zależności od sygnatury mogą mieć inny kod

Single responsibility
Open-closed principle
Liskov substitution
Interface segregation
Dependency inversion

Protected Variations – ochrona przed zmiennością

- Ogólna zasada: rozpoznaj miejsca, w których zmiany mogą się pojawić i otocz je stabilnym interfejsem

Open-Close Principle: klasy powinny być otwarte na rozszerzenia i zamknięte na modyfikacje

Law of Demeter (*nie rozmawiaj z obcymi*) – czyli do kogo metoda może wysłać komunikat:

- obiekt `this`, atrybut `this`
- parametr metody
- obiekt utworzony wewnatrz metody

Liskov substitution

Metody, które używają referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych (≈ polymorphism, design by contract)

Interface segregation

Podziel „duże" interfejsy na mniejsze i dokładniejsze, tak by żaden klient nie był zależny od metod których nie używa

Dependency inversion

Uzależniać klasy od abstrakcji, a nie od konkretnych klas.

Decyzje o wiązaniu obiektów wyprowadzamy „na zewnątrz"

```
Coffee c = new SmallCoffee();
c = new Milk(c) ;
c = new Sprinkle(c);
c = new Sprinkle(c);
c = new Sprinkle(c);
c.getCost();
```

krok 1 Klient ma referencje do abstrakcyjnej fabryki `Factory` i inicjalizuje ją jedną z jej podklas.

krok 2 `Factory` ma chronioną abstrakcyjną metodę `create()` z której korzysta publiczna metoda `createProduct()`. Metoda `create()` jest implmentowana w fabrykach konkretnych.

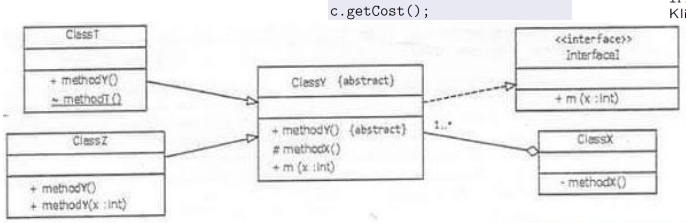
krok 3 Fabryka konkretna tworzy produkt implementujący interfejs `IProduct` i zwraca go do klienta przez metodę `createProduct()` Klient zna jedynie interfejs produktu.

```
public class Test  {

public boolean equals( Test other ) {
  System.out.println( "Moja metoda equals()" );
  return false;  }

public static void main( String [] args ) {
  Object t1 = new Test();
  Object t2 = new Test();
  Test t3 = new Test();
  Object o1 = new Object();

  int count = 1;
  System.out.println( count++ );// prints 1
  t1.equals( t2 ) ;        t2 jest typu Object
  System.out.println( count++ );// prints 2
  t1.equals( t3 );      t1 jest rzutowana na Object
  System.out.println( count++ );// prints 3
  t3.equals( o1 );   wywoływana metoda z nadklasy
  System.out.println( count++ );// prints 4
  t3.equals(t3);                override
  System.out.println( count++ );// prints 5
  t3.equals(t2);     t2 jest typu Object
 }
}
```



W ClassY można odwołać się do metody methodX() obiektu typu ClassX *1, private 2, nie ma instancji*

W ClassX można odwołać się do metody methodX() obiektu typu ClassY *nie bo protected*

W ClassX można odwołać się do metody methodY() obiektu typu ClassY *tak, ma w sobie ta klase + public*

W ClassX można odwołać się do metody m(x:int) obiektu typu ClassY *tak, leci przez interface*

W ClassX można odwołać się do metody methodT() klasy ClassT *nope,*

Metoda methodY(x: int) w klasie ClassZ przesłania (ang. override) metodę nadklasy *overload*

Abstract Factory vs. Factory Method

1. W Factory Method mamy jeden rodzaj produktów, w Abstract Factory rodzinę powiązanych produktów (*wiele interfejsów*)
2. Konkretne fabryki dziedziczące po Abstract Factory mogą wykorzystywać metodę szablonową, ale nie muszą

Factory Method

```
Factory f = new ConcreteFactory();
IProduct p = f.createProduct()
```

Abstract Factory

```
Factory f = new ConcreteFactory();
IProductA pA = f.createProductA();
IProductB pB = f.createB(); //metoda nieszablonowa
```

AF Produkty mają wspólny interfejs.

BU Produkty tworzone przez różnych budowniczych mogą być różne, więc nie ma powodu by ustalać dla nich wspólny interfejs.

AF Akcent jest na tworzenie rodziny powiązanych produktów, każdy produkt w jednym kroku.

BU Akcent jest na tworzenie złożonego produktu krok po kroku. Kroki możemy omijać/modyfikować co daje większa elastyczność.

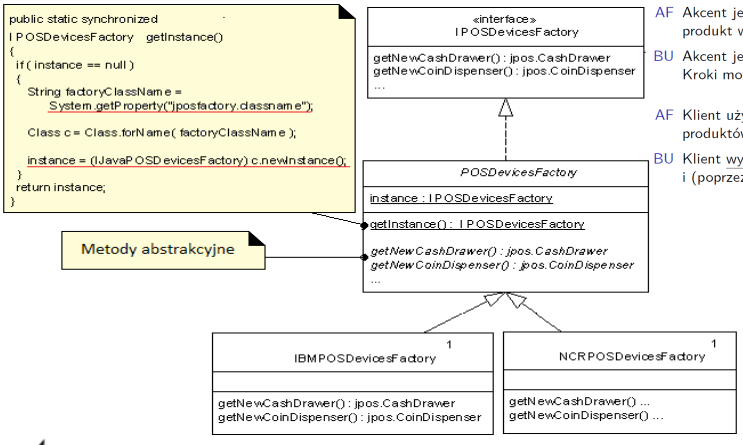
AF Klient używa metod Abstract Factory do stworzenia grupy produktów, ale często nie wie z jakiej konkretnej fabryki korzysta.

BU Klient wybiera budowniczego, od którego zależy postać produktu i (poprzez nadzorcę) poleca ten produkt zbudować.

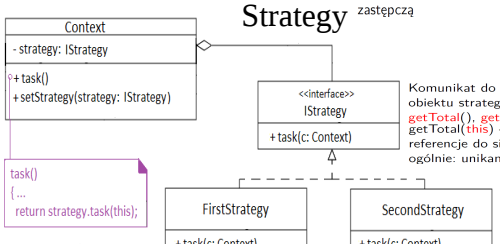
```
public static synchronized
IPOSDevicesFactory  getInstance()
{
 if ( instance == null )
 {
   String factoryClassName =
       System.getProperty("jposfactory.classname");

   Class c = Class.forName( factoryClassName );

   instance = (IJavaPOSDevicesFactory) c.newInstance();
 }
 return instance;
}
```



Strategy



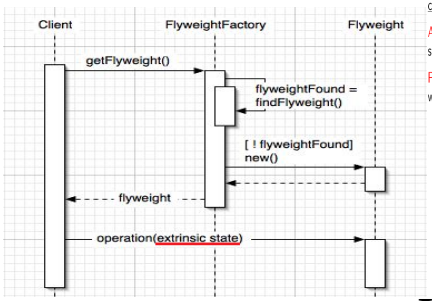
Komunikat do obiektu kontekstowego i do obiektu strategii typowo ma tę samą nazwę: getTotal(), getTotal(Sale) getTotal(this) - o. kontekstowy przekazuje referencje do siebie na potrzeby współpracy, ogólnie: unikamy ekstrakcji

Dummy Object – np. metoda w SUT wymaga parametru, ale dla testu nie ma znaczenia jaki jest ten parametr

Fake – gdy DOC jest niedostępny albo nie nadaje się do testów tworzymy „lekką" klasę zastępczą

Aspect Oriented Programming

Klasy logiki biznesowej zajmują się tylko logiką, ale są pokryte warstwami innych funkcjonalności (aspektami), o których nie wiedzą

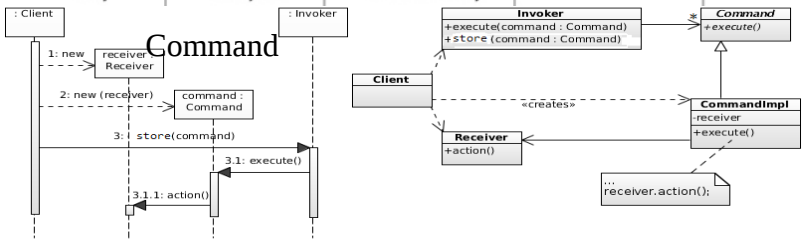


Aspect składa się z advice oraz pointcut, które określają co, kiedy i gdzie trzeba zrobić

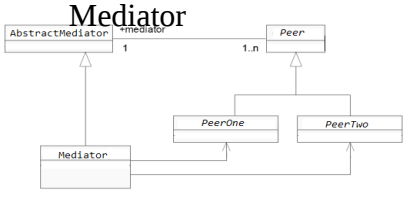
Advice określa co trzeba zrobić (jaki kod wywołać) i kiedy w stosunku do danego pointcut (*before, after, after exception...*)

Pointcut określa gdzie to trzeba zrobić, czyli punkt odniesienia wyznaczony przez joint points (*wywołania metod*)

Command



Mediator



JACEK MUCHA