Wzorce projektowe na przykładzie C++

Praktyczne Aspekty Rozwoju Oprogramowania

- Piotr Kowalczyk, Grzegorz Olender
- 20-04-2016

NOKIA

Kwestie organizacyjne

- Kurs Praktyczne Aspekty Rozwoju Oprogramowania został przygotowany przez pracowników firmy Nokia Networks.
- Uczestnicy kursu otrzymają dawkę wiedzy z zakresu rozwoju oprogramowania wraz z przykładami zastosowań praktycznych.
- Harmonogram oraz materiały dodatkowe są dostępne w serwisie Moodle:
  - PWr: http://pst.pwr.wroc.pl/moodle/
  - UWr: http://kno.ii.uni.wroc.pl/ii/
- Prosimy o wyciszenie telefonów oraz wpisanie się na listę obecności.

Agenda

- Wprowadzenie (15 min.)
  - Antywzorce, czyli jak nie programować?
  - Co to są wzorce i po co je stosujemy?
- Wybrane wzorce strukturalne (40 min.)
  - Adapter (Adapter)
  - Proxy (Pośrednik)
- Wybrane wzorce kreacyjne (40 min.)
  - Abstract Factory (Fabryka Abstrakcyjna)
- Wybrane wzorce czynnościowe (40 min.)
  - Observer (Obserwator)

Antywzorce, czyli jak nie programować?

- Dobre praktyki odnośnie wzorców projektowych
  - Rozwiązuj problemy najprostszymi możliwymi sposobami
  - Używaj wzorców tylko tam gdzie jest to naprawdę konieczne i unikaj antywzorców
- Przykładowe antywzorce w programowaniu
  - The Blob
  - Golden Hammer
  - Spaghetti Code
  - Cut-and-Paste Programming
- Więcej informacji dla zainteresowanych tematem:
  - http://sourcemaking.com/antipatterns

Co to są wzorce projektowe?

- Christopher Alexander („A Pattern Language”):
  - „Wzorzec opisuje problem występujący wielokrotnie w danym środowisku, pokazując podstawowe rozwiązanie tego problemu w taki sposób, aby można było wielokrotnie użyć tego rozwiązania do wszystkich wystąpień danego problemu, bez konieczności ponownego wykonywania tych samych czynności projektowych.”
- „Gang of Four” („Design Patterns”):
  - „Opis komunikujących się obiektów i klas, które przerabia się w celu rozwiązania ogólnego danego problemu przy dokładnie określonym kontekście.”
- Mark Johnson:
  - „Wzorce projektowe pomagają uczyć się na przykładzie sukcesów innych jak unikać własnych porażek.”

Po co stosujemy wzorce projektowe?

- Poprawa komunikacji
  - dzięki wspólnej (i spójnej) terminologii
- Poprawa jakości
  - większa odporność na błędy dzięki sprawdzonym rozwiązaniom
- Poprawa produktywności
  - unikanie wymyślania koła na nowo

Jak klasyfikujemy wzorce projektowe?

- Podstawowy podział wzorców projektowych (GoF, „Design Patterns”)
  - Wzorce kreacyjne (creational)
  - Wzorce strukturalne (structural)
  - Wzorce czynnościowe (behavioral)
- Wzorce dzielimy również pod względem ich wewnętrznej struktury:
  - Wzorce oparte na obiektach (większość wzorców)
  - Wzorce oparte na klasach (mniejszość wzorców)
- Ciekawostka
  - Istnieje wzorzec który ma zarówno wersję obiektową jak i klasową, jest nim Adapter (szczegóły w dalszej części prezentacji)

Jak klasyfikujemy wzorce projektowe?

		Klasyfikacja według zastosowania		
		Kreacyjne	**Strukturalne**	**Czynnościowe**
Klasyfikacje według struktury wewnętrznej	**Klasa**	Factory Method	Adapter (class)	Interpreter Template Method
	Obiekt	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Jak klasyfikujemy wzorce projektowe?

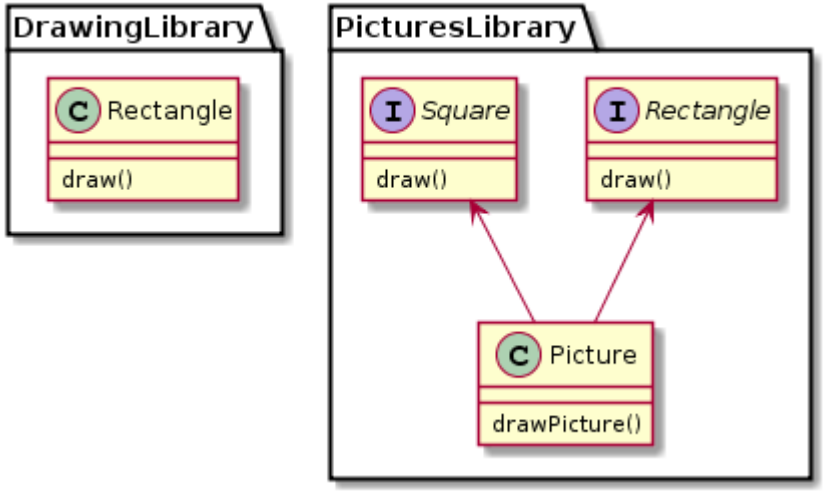
		Klasyfikacja według zastosowania		
		Kreacyjne	**Strukturalne**	**Czynnościowe**
Klasyfikacje według struktury wewnętrznej	**Klasa**	Factory Method	Adapter (class)	Interpreter Template Method
	Obiekt	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Adapter (Adapter)

Adapter
Problem

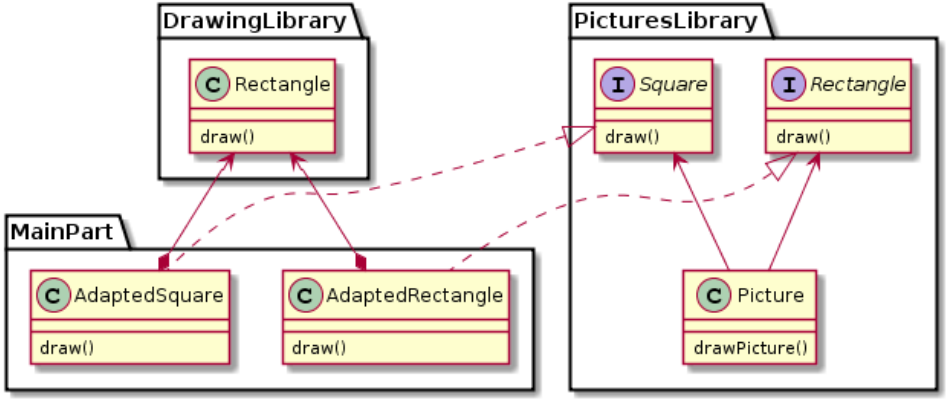
• Chcemy wykorzystać klasę `DrawingLibrary::Rectangle` w module `PicturesLibrary`

• Moduł `PicturesLibrary` potrafi jednak używać tylko interfejsów `PicturesLibrary ::Rectangle` oraz `PicturesLibrary ::Square` które są niekompatybilne z `DrawingLibrary::Rectangle`



Adapter
Rozwiązanie

• Tworzymy odpowiednie adaptery dla klasy `DrawingLibrary::Rectangle`, które zachowują się tak jak obiekty typu `PicturesLibrary::Rectangle` oraz `PicturesLibrary::Square`



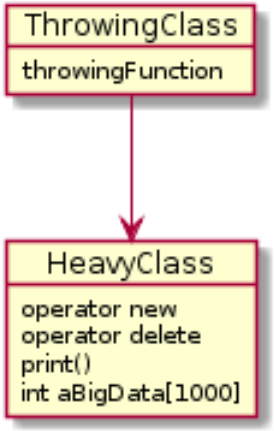
Adapter
Podsumowanie

- Podstawowe cechy:
  - realizuje zasadę odwrócenia zależności (Dependency Inversion Principle)
  - umożliwia dopasowanie interfejsu jednego obiektu (lub grupy obiektów) tak aby mógł być wykorzystany w innym (niekompatybilnym) otoczeniu
  - nie tworzy bezpośrednich zależności pomiędzy modułami które łączy
  - nie wymaga modyfikowania istniejącego już kodu
- Możliwe wersje implementacji:
  - obiektowa (wykorzystująca agregację) lub klasowa (wykorzystująca dziedziczenie)
- Przykładowe zastosowania:
  - adapter przekształcający interfejs
  - adapter zawężający interfejs
  - adapter agregujący interfejsy

Proxy (Pośrednik)

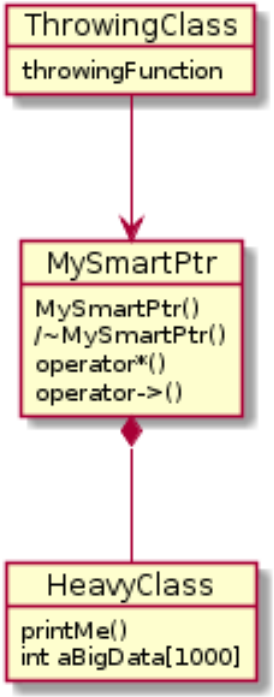
Proxy
Problem

- Klient (`ThrowingClass`) musi zarządzać pamięcią utworzonego dynamicznie obiektu typu `HeavyClass`
- W przypadku nieukończenia wykonywania procedury (np. rzucenie wyjątku) nastąpi wyciek pamięci



Proxy
Rozwiązanie

- Tworzymy pośrednika (`MySmartPtr`) który zarządza pamięcią obiektu typu `HeavyClass`
- Pośrednik udostępnia ten sam interfejs co `HeavyClass` (poprzez operatory *, ->) dzięki czemu klient (`ThrowingClass`) używa pośrednika tak samo jak instancji klasy `HeavyClass`

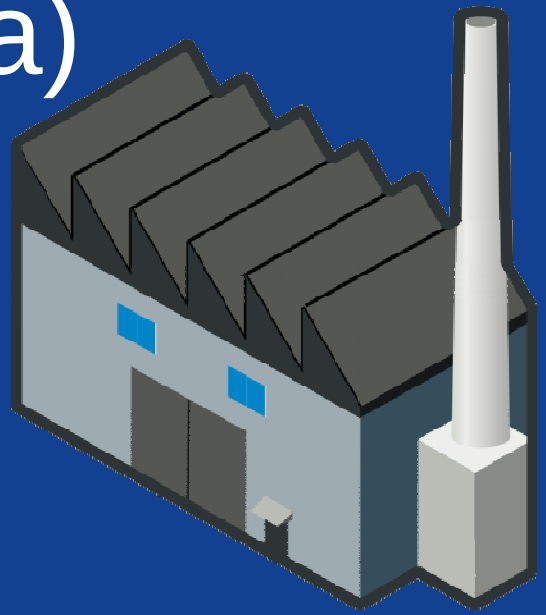


Proxy
Posumowanie

- Podstawowe cechy:
  - udostępnia ten sam interfejs co rzeczywista klasa jednak zmienia jej zachowanie
  - zawiera wskaźnik/referencję do rzeczywistego obiektu
  - kontroluje dostęp do rzeczywstego obiektu
  - zarządza czasem życia rzeczywistego obiektu
- Przykładowe zastosowania:
  - zarządzanie pamięcią (smart pointer)
  - tworzenie obiektów na żądanie (virtual proxy)
  - reprezentacja zdalnych obiektów (remote proxy)
  - kontrola dostępu do obiektu (protection proxy)

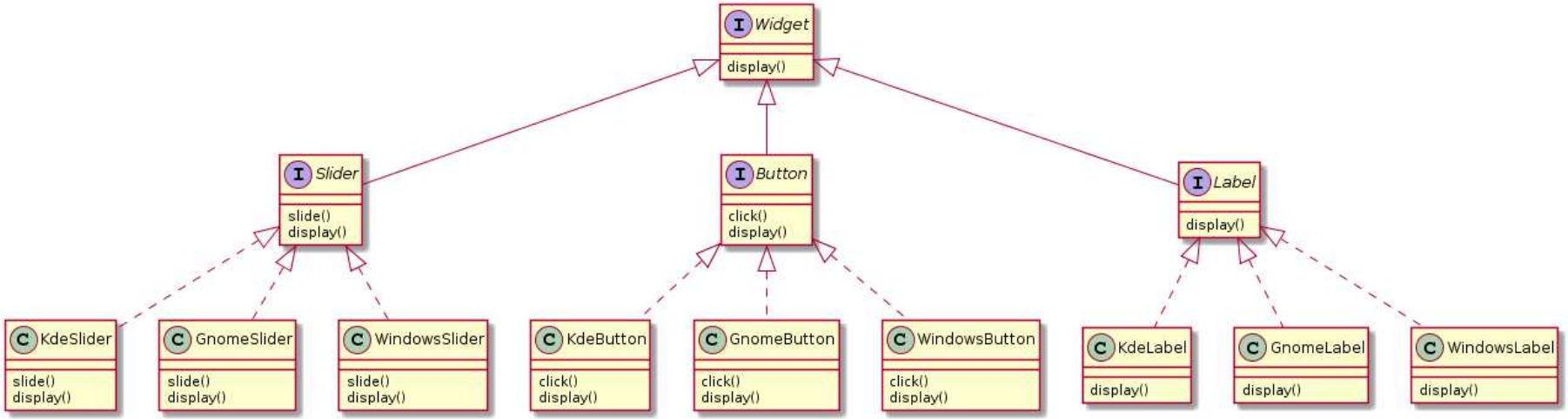
Abstract Factory

(Fabryka Abstrakcyjna)



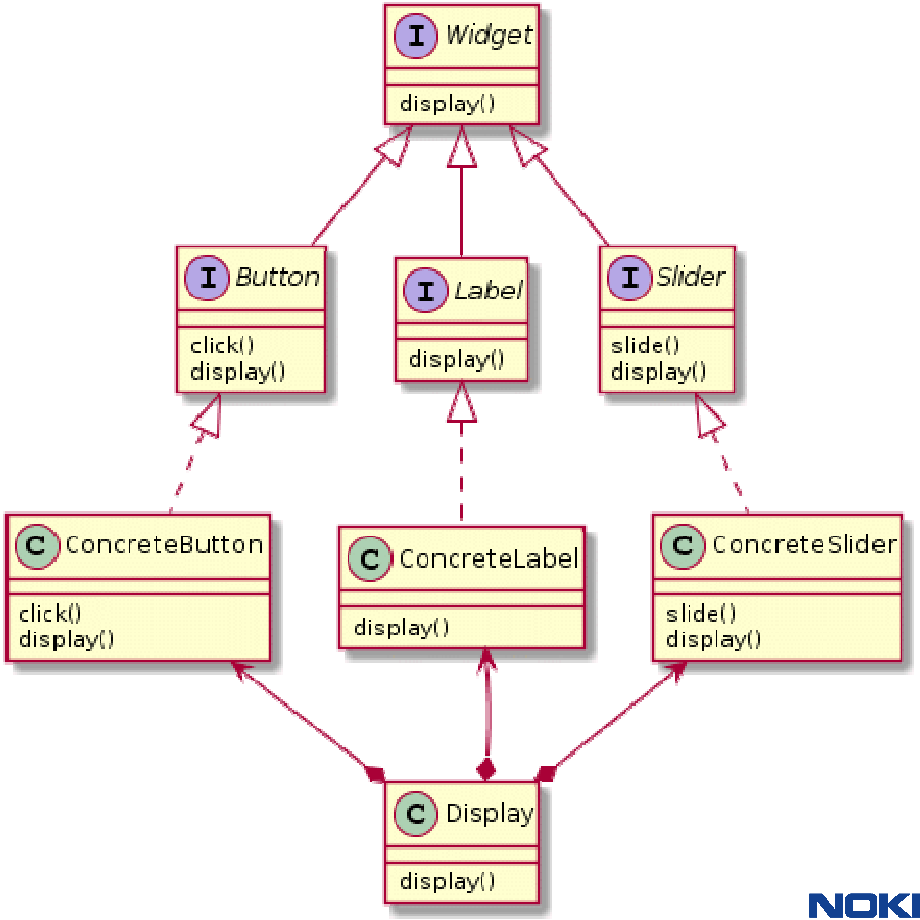
Abstract Factory
Problem

- Istnieje kilka rodzin klas implementujących ten sam interfejs



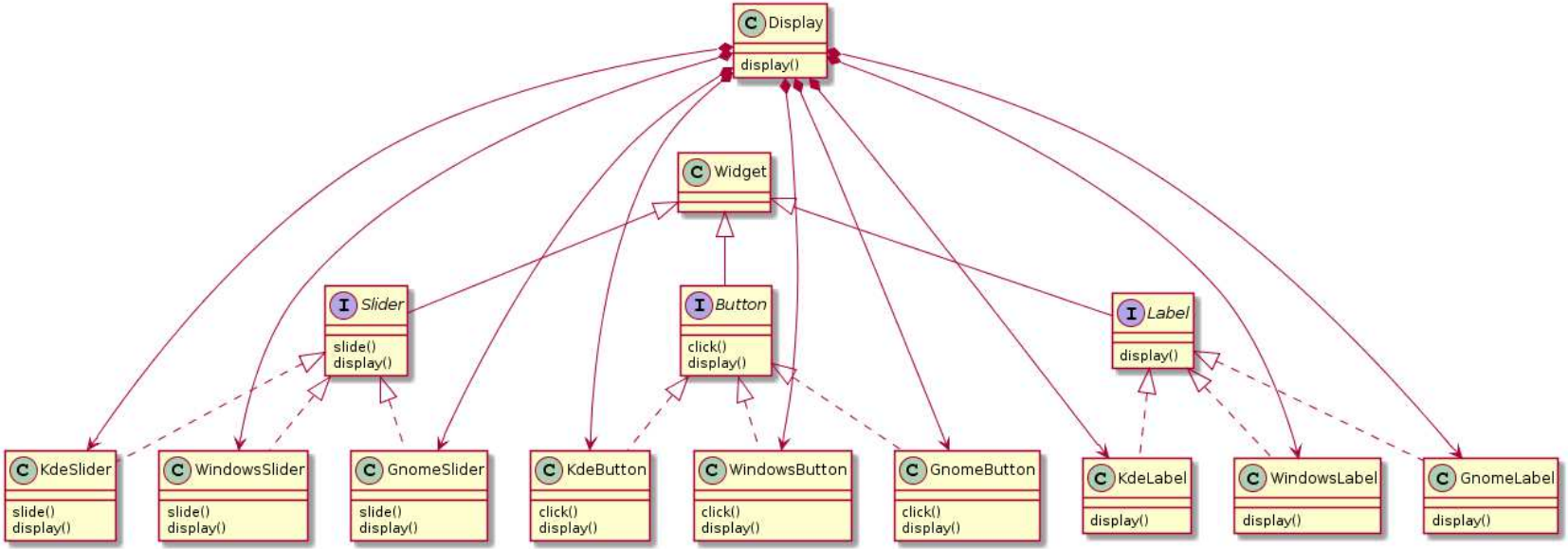
Abstract Factory
Problem

- Klient (`Display`) tworzy i posiada zestaw obiektów należących do danej rodziny



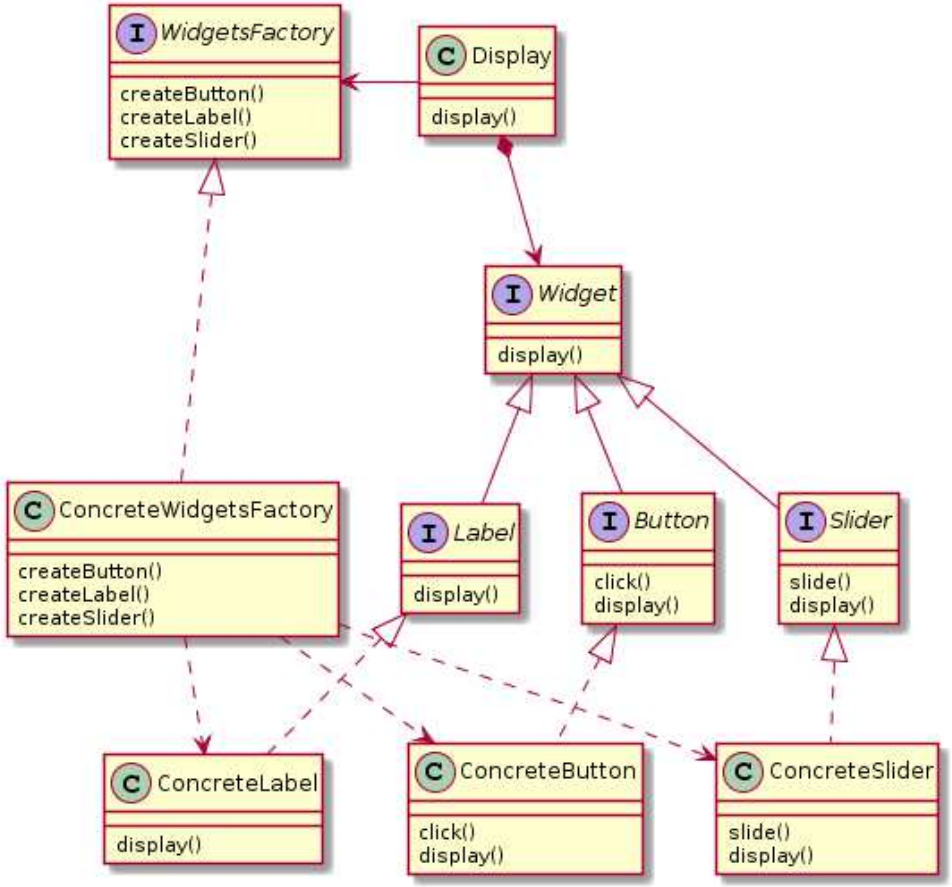
Abstract Factory
Problem

Istnieje ogromna liczba niepotrzebnych zależności



Abstract Factory
Rozwiązanie

- Tworzymy fabrykę abstrakcyjną (`WidgetsFactory`)która jest zaimplementowana przez konkretne fabryki (`ConcreteWidgetsFactory`)
- Proces tworzenia obiektów konkretnej rodziny delegujemy do konkretnej fabryki
- Klasa `Display` zależy jedynie od abstrakcyjnej fabryki i nie zależą od konkretnych implementacji tworzonych widżetów



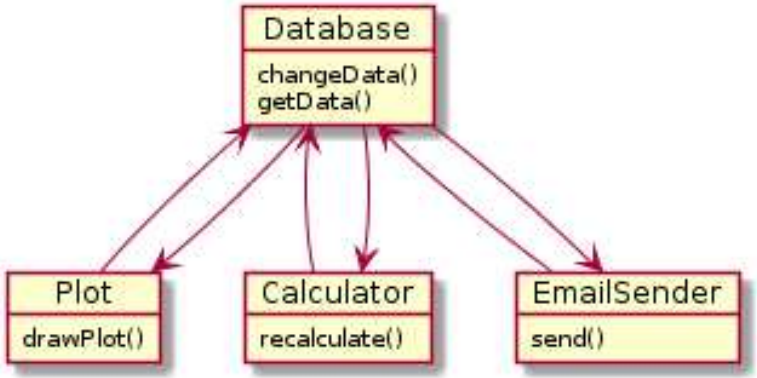
Abstract Factory
Posumowanie

- Zalety:
  - realizuje zasadę otwarte-zamknięte (Open-Closed Principle)
  - uniezależnia klienta od implementacji konkretnych rodzin obiektów
  - zapobiega duplikowaniu kodu oraz problemom związanym z utrzymaniem takiego kodu
  - ułatwia konfigurowanie systemu jedną z wielu dostępnych implementacji
  - uniezależnia klienta fabryki od tego w jaki sposób obiekty są tworzone
- Wady:
  - konkretna fabryka musi czasem poradzić sobie ze zbyt wąskim interfejsem tworzenia elementów
  - utrudnia dodawanie nowego typu obiektu do rodziny obiektów

Observer (Obserwator)

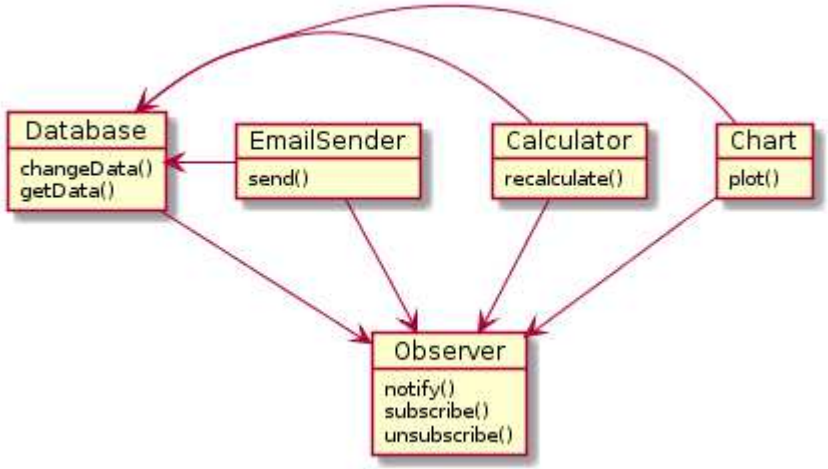
Observer
Problem

• Chcemy zamodelować interakcje pomiędzy obiektami należącymi do różnych warstw abstrakcji



Observer
Rozwiązanie

- Tworzymy Obserwatora (`Observer`) który przerywa cykl zależności
- Wszystkie powiązane ze sobą obiekty zależą od generycznego mechanizmu powiadamiania zaimplementowanego w Obserwatorze
- Baza danych powiadamia Obserwatora o zaistniałym zdarzeniu
- Pozostałe obiekty mogą rejestrować się na wybrane zdarzenie dostarczając Obserwatorowi sposób jego obsłużenia



Observer
Posumowanie

- Podstawowe cechy:
  - realizuje zasadę otwarte-zamknięte (Open-Closed Principle)
  - daje możliwość rejestrowania / wyrejestrowywania się na dane wydarzenie
  - wysyła powiadomienie jeśli obiekt obserwowany wykona określoną akcję
  - definiuje sposób obsługi zdarzenia (np. przy pomocy std::function<void ()>)
  - przerywa cykliczne zależności pomiędzy obiektami
- Przykłady zastosowania:
  - notyfikacja dowolnie dużej liczby obiektów
  - notyfikacja obiektów niepowiązanych ze sobą logicznie (unikanie zbędnych zależności w kodzie)
  - daje możliwość wiązania obiektów (Obserwator - Obserwowany) w trakcie działania programu
- Uwaga: istnieją gotowe implementacje mechanizmu obserwatora (np. boost::signals2)


NOKIA