Technologia Programowania 2017/2018
Wykład 5
Mockito, wzorce GoF

Jakub Lemiesz

Lista zadań na laboratorium

Technologia Programowania 2017/2018 – Lista 3 (lab)

Termin: ~~do 17 listopada~~ do 24 listopada
(z powodu godzin rektorskich 31 października straciliśmy wykład)

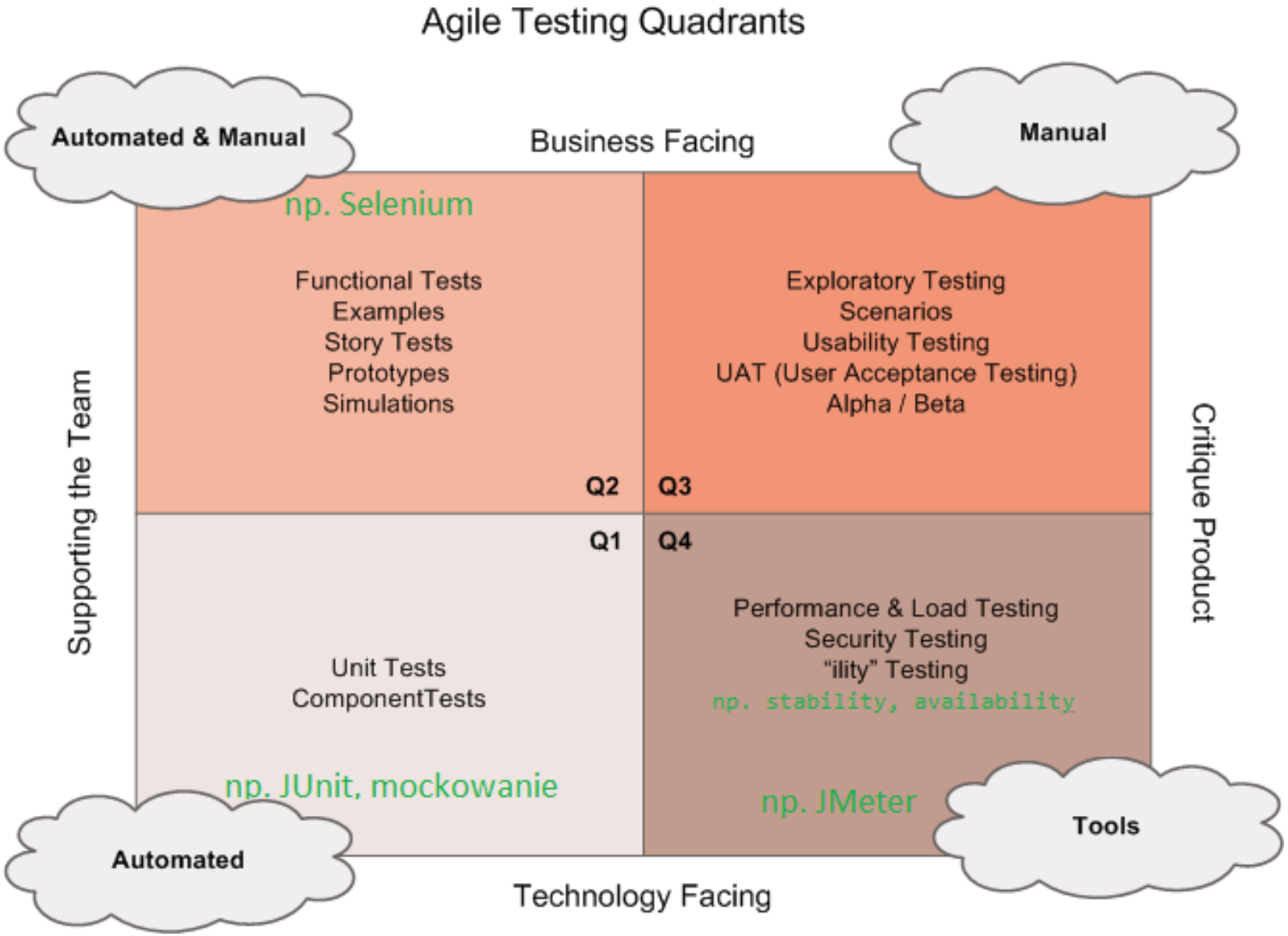
Zadanie 9 — Wykonaj zadania dotyczące wzorca Facade. Zadanie wymaga podstawowej znajomości Mockito, o którym będzie mowa na wykładzie. **(12 p.)**

Zadanie 10 — Wykonaj zadania dotyczące wzorca Flyweight. Najnowszą wersję JMeter możesz pobrać tutaj. Narzędzie ma stosunkowy łatwy w obsłudze interfejs graficzny – zapoznaj się z jego podstawowymi możliwościami. Ustawione w testach 10000 wątków to może być dla niektórych maszyn spore obciążenie, proponuję rozpocząć testowanie z mniejszą liczbą. **(14 p.)**

Testowanie

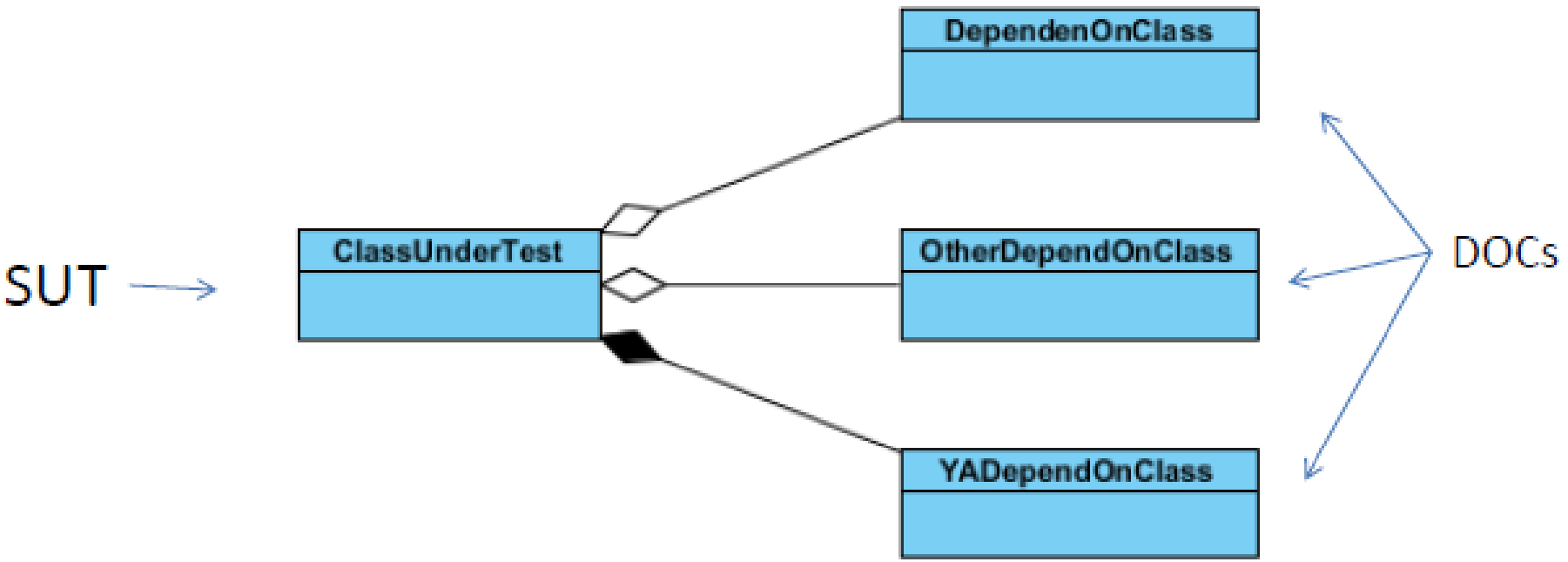
- Tradycyjnie: testy jednostkowe, integracyjne, systemowe, testy akceptacyjne (alfa i beta)
- Metodyki agile: testowanie po każdej iteracji, ciągła integracja, ciągły kontakt z klientem
- Agile Testing Quadrnats – klasyfikacja testów w metodykach agile (*nic nie mówi o kolejności*)

Agile Testing Quadrants - źródło, dyskusja



Testowanie jednostkowe, testowanie komponentów

- Cel: szybko i automatycznie testować małe elementy w izolacji (*TDD, refaktoryzacja*)
- System Under Test (SUT) może zależeć od wielu Depended On Components (DOCs)



Dlaczego zależność od DOCs jest problem?

1. DOCs nie są gotowe
2. Do testów DOCs o szczególnych właściwościach (*np. generator l. losowych zawsze zwracający* π)
3. Odwołanie do DOC może mieć skutki uboczne (*np. połączenie z prawdziwą bazą danych*)
4. Stworzenie i użycie prawdziwych DOCs może być czasochłonne a testy mają być szybkie (*testy integracyjne to późniejszy etap*)

Dummy Objects, Fakes

- Zamiast DOC można użyć tzw. dublera, który ma taki sam interfejs jak DOC, jest „lżejszy” i może wykonywać pewne dodatkowe zadania (*Dummy Objects, Fakes, Stubs, Spies, Mocks*)
- Dummy Object – dla testu nie ma znaczenia jaki jest DOC
- Fake – gdy DOC niedostępny albo nie nadaje się do testów tworzymy „lekką” klasę zastępczą
- Żeby stworzyć Dummy Object lub Fake nie potrzebujemy dodatkowych narzędzi

Dlaczego Mockito?

- Stubs, Spies, Mocks zazwyczaj tworzone są w oparciu o specjalne biblioteki
- Biblioteki do mokowania są bardzo powszechnie używane: np. JMockit, EasyMock, Mockito i dziesiątki innych dla Java i *C#*
- Mockito jest jednym z najpopularniejszych systemów (*autor biblioteki: Szczepan Faber*)

Spies – czyli śledzenie 'prawdziwych' obiektów

Chcemy mieć pewność, że prawdziwy DOC zachowuje się w odpowiedni sposób:

```
List list = new LinkedList();
List spy = spy(list);
//wywołanie 'prawdziwych' metod
spy.add("one");
verify(spy).add("one");
//możemy zasłonić metody
when(spy.size()).thenReturn(100);
System.out.println(spy.size());
```

Stubs – czyli kikuty

Tworzymy dublera, który ma zamiast metod 'kikuty'.
W testach (*np. JUnit*) piszemy tak:

```
//Dlaczego static?
import static org.mockito.Mockito.*;
//Można mokować klasy albo interfejsy
LinkedList mockList = mock(List.class);
//stubbing - 'kikutowanie?'
when(mockList.get(0)).thenReturn("first");
when(mockList.get(1)).thenThrow(new Exception
when(mockList.get(anyInt())).thenReturn("X");
```

Mocks – czyli szpiegowanie kikutów

Chcemy mieć pewność, że SUT zachowuje się poprawnie i komunikacja między SUT a DOC przebiega we właściwy sposób:

```
mockList.add("A");
//weryfikacja wywoływania metod
verify(mockList).add("A")
verify(mockList).add("B")
//weryfikacja kolejność wywoływania
InOrder inOrder = inOrder(mockList);
inOrder.verify(mockList).add("A");
inOrder.verify(mockList).add("B");
```

Mocks w Mockito – sporo możliwości

```
//exact number of invocations verification
verify(mockedList, never()).add("A");
verify(mockedList, atLeastOnce()).add("A");
verify(mockedList, times(2)).add("A");
verify(mockedList, atLeast(2)).add("A");
verify(mockedList, atMost(5)).add("A");

//firstMock wywołany przed secondMock
InOrder inOrder = inOrder(firstMock, secondMock);
inOrder.verify(firstMock).add("was called first");
inOrder.verify(secondMock).add("was called second");

//mockTwo i mockThree się nie komunikowały
verifyZeroInteractions(mockTwo,mockThree);
```

Mocks w Mockito – matechrs

```
//przykładowe matchery typów
verify(mock).someMethod(anyInt(), anyString());
verify(mock).someMethod(eq("third argument"));
verify(mock).someMethod(argThat(isValid()));
```

```
//własne matchery
class IsListOfTwoElements extends ArgumentMatcher<List>
   public boolean matches(Object list) {
       return ((List) list).size() == 2;
   }
}

verify(mock).addAll(listOfTwoElements());
```

Pobieranie przez Maven'a:

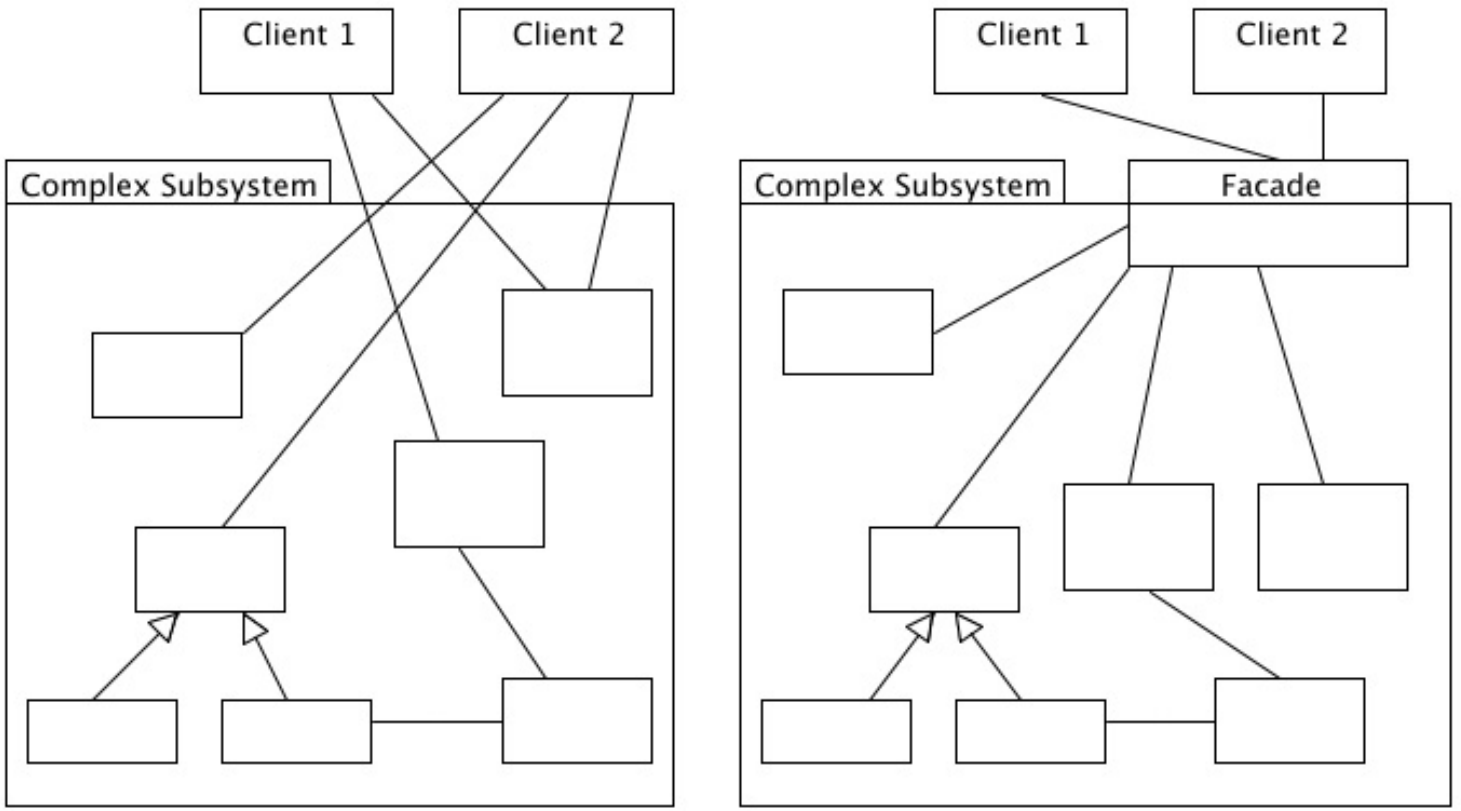
```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.10.19</version>
    <scope>test</scope>
</dependency>
```

Więcej o Mockito na:
mockito.github.io/mockito/docs/current/org/mockito/Mockito.html

GoF: Facade

Problem: sprzężenie między komponentami systemu

Rozwiązanie: klasa `Facade` dostarczająca dla całego podsystemu wysokopoziomowy interfejs



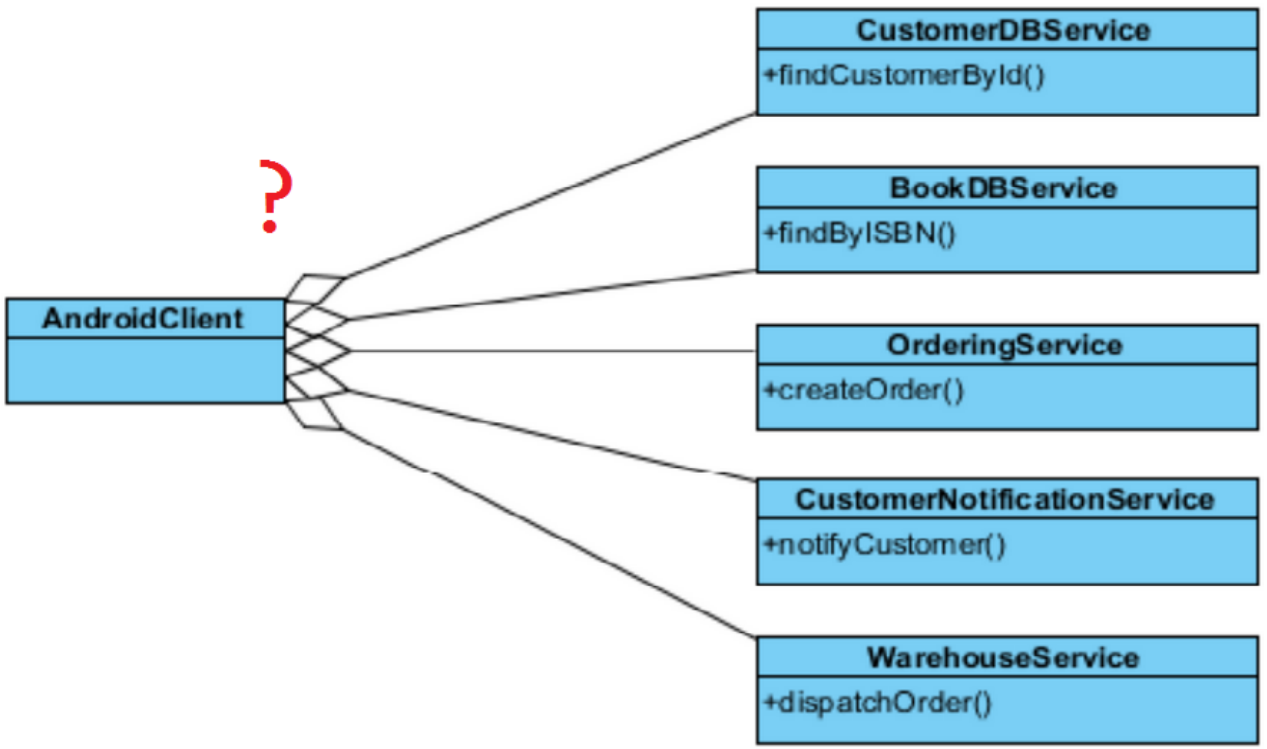
źródło

GoF: Facade – typowe zastosowania

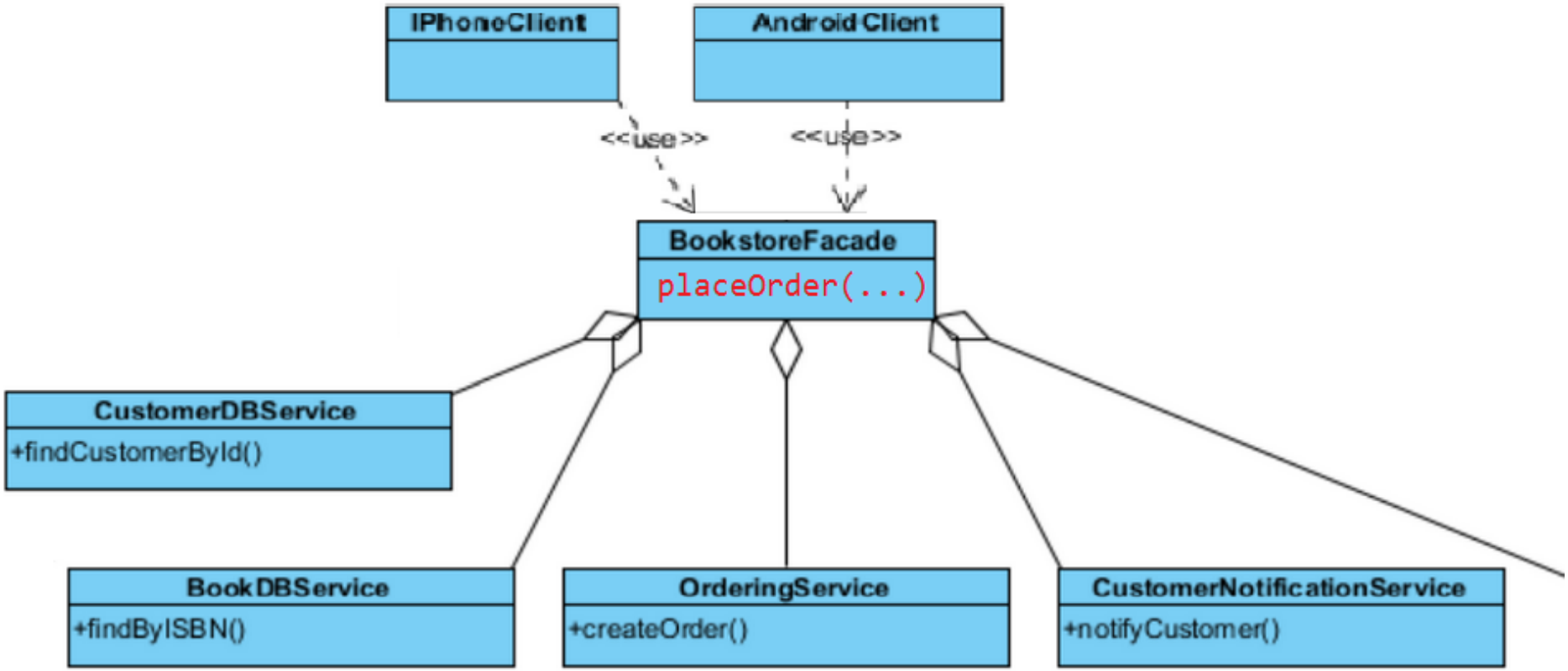
1. Prosty interfejs dla złożonego podsystemu
   (*łatwiej zrozumieć i wykorzystać funkcjonalność*)
2. Zebranie kilku interfejsów w jeden sensowny
   (*np. interfejsy mogą być źle zaprojektowane*)
3. Punkt wejściowy do podsystemu/warstwy
   - niezależny rozwój podsystemów
   - blokada bezpośrednegoi dostępu do metod
     (*np. względy bezpieczeństwa*)

GoF: Facade – zadanie na lab

1. Mamy interfejsy dla różnych klas usługowych (*na razie jedynie mocki*)
2. Chcemy napisac aplikacje, która odwołując się do tych klas umożliwi kupienie książki



GoF: Facade – zadanie na lab



GoF: State

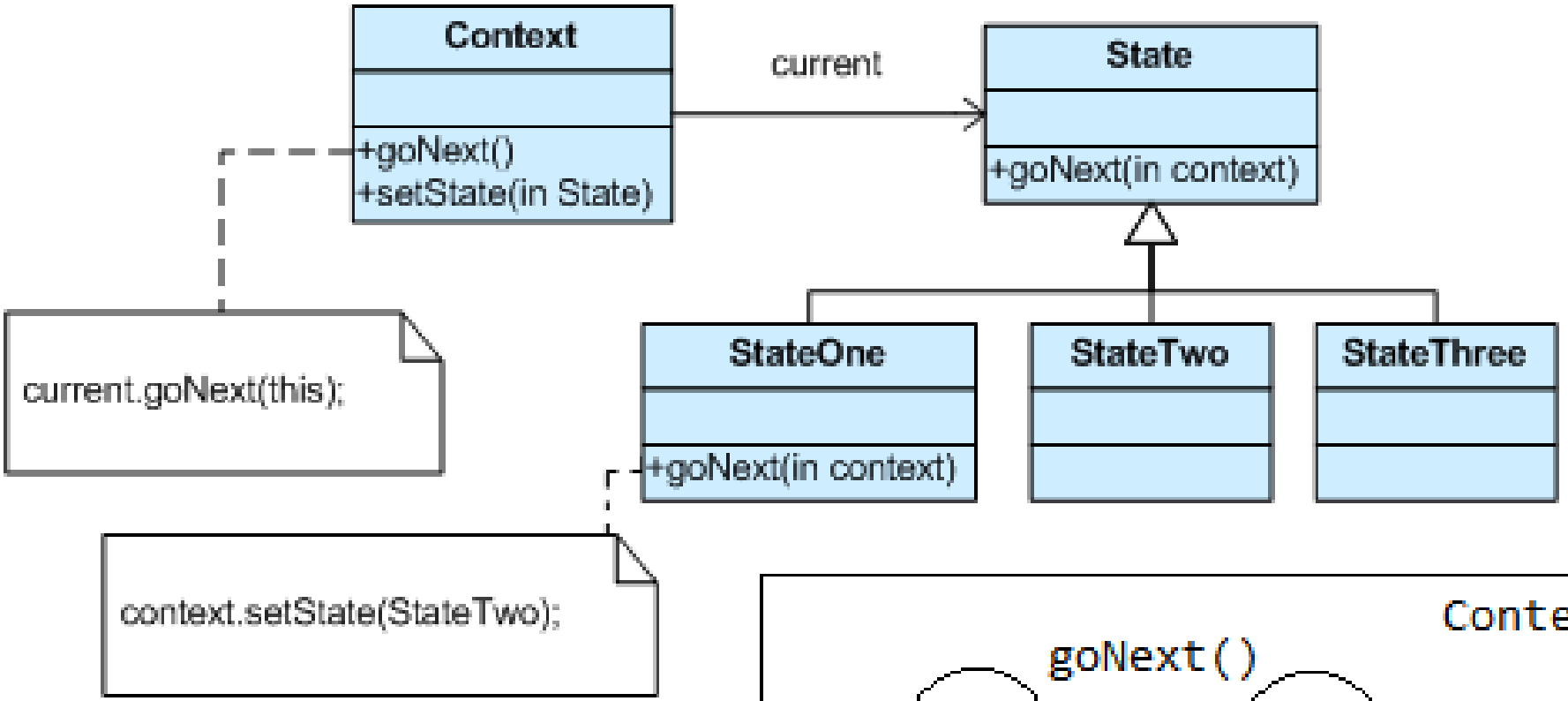
Problem:

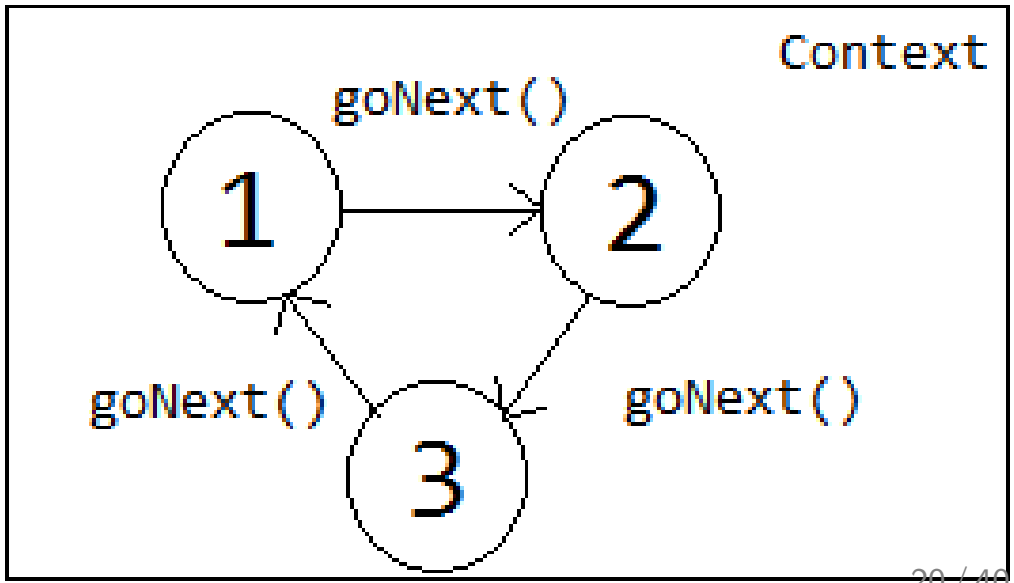
- Obiekt może znajdować się w różnych stanach
- Zachowanie obiektu zależy od jego stanu (*np. metody zawiera logikę warunkową* `case`)

Rozwiązanie:

1. Utwórz klasy reprezentujące poszczególne stany (*implementujące wspólny interfejs*)
2. Deleguj operacje z obiektu kontekstowego do obiektu określającego aktualny stan
3. Określ przejścia między stanami

GoF: State

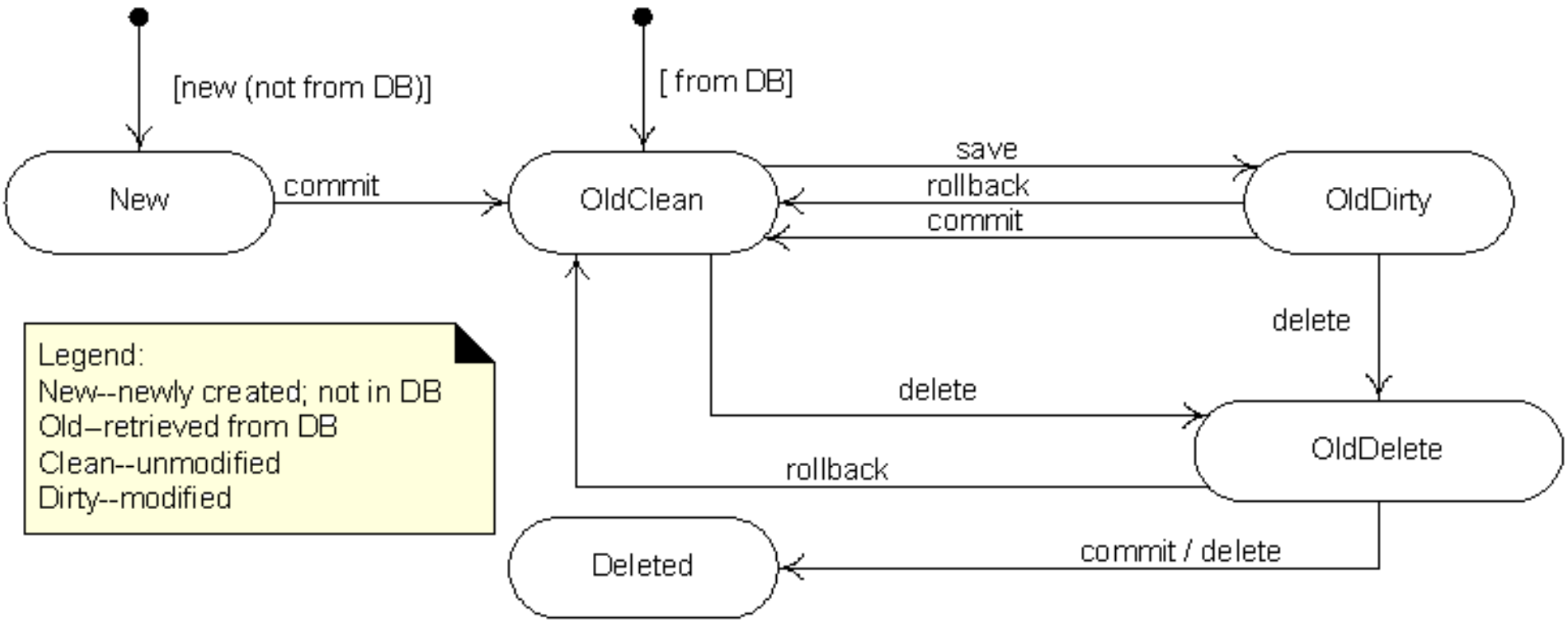

Context
current
State
+goNext()
+setState(in State)
+goNext(in context)
current.goNext(this);
StateOne
StateTwo
StateThree
+goNext(in context)
context.setState(StateTwo);


Context
goNext()
1
2
goNext()
goNext()
3

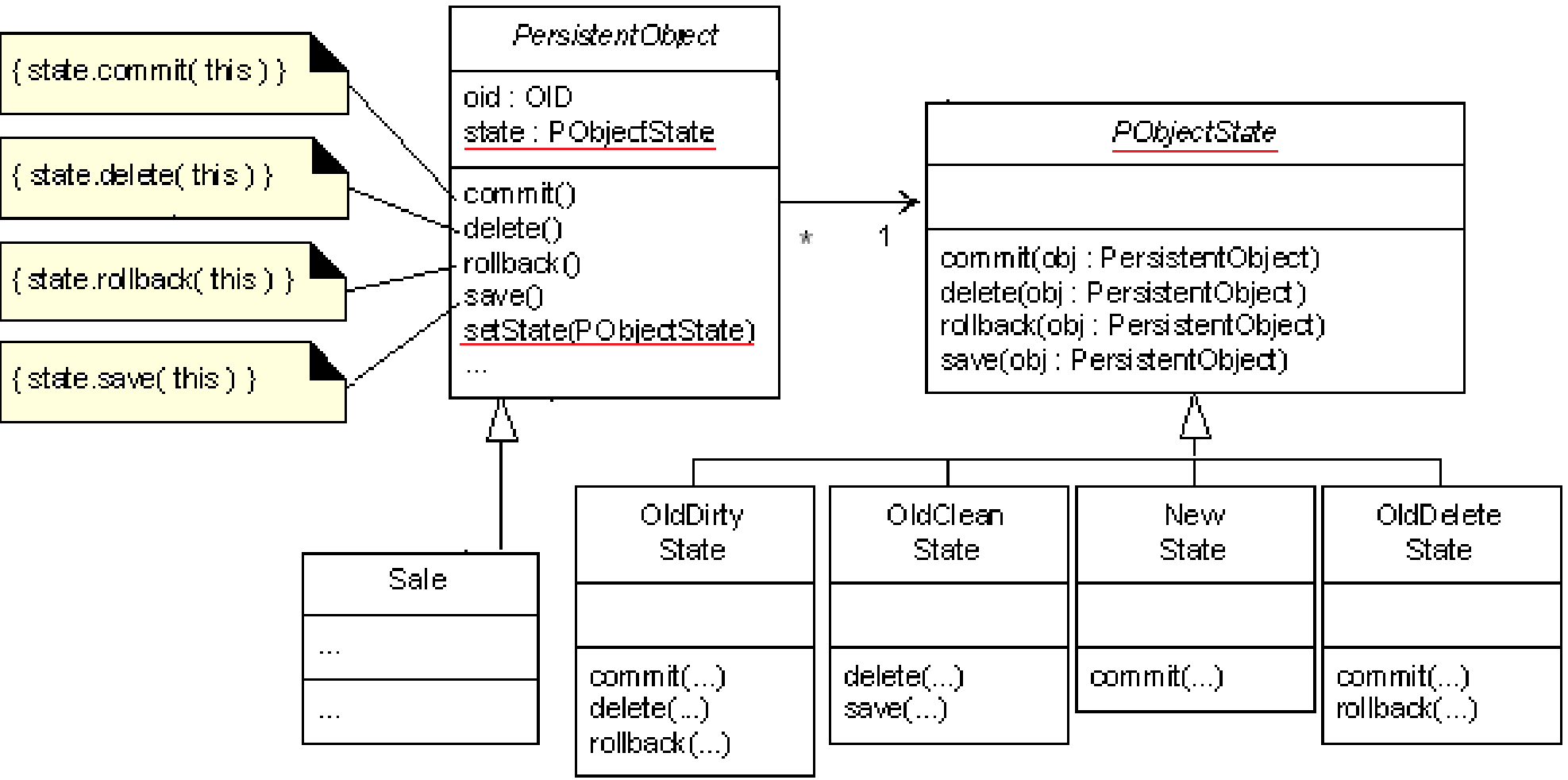
Przykład: stany transakcyjne

1. Stan transakcyjny określa zgodność obiektu z tym co jest w bazie danych: New, OldClean, OldDirty,...
2. Operacje commit, rollback, save i delete zmieniają stan transakcyjny, rezultat zależy od aktualnego stanu
3. Można by to spróbować zapisać w kodzie instrukcjami warunkowymi, ale nie byłoby to ładne rozwiązanie...

Przykład: stany transakcyjne



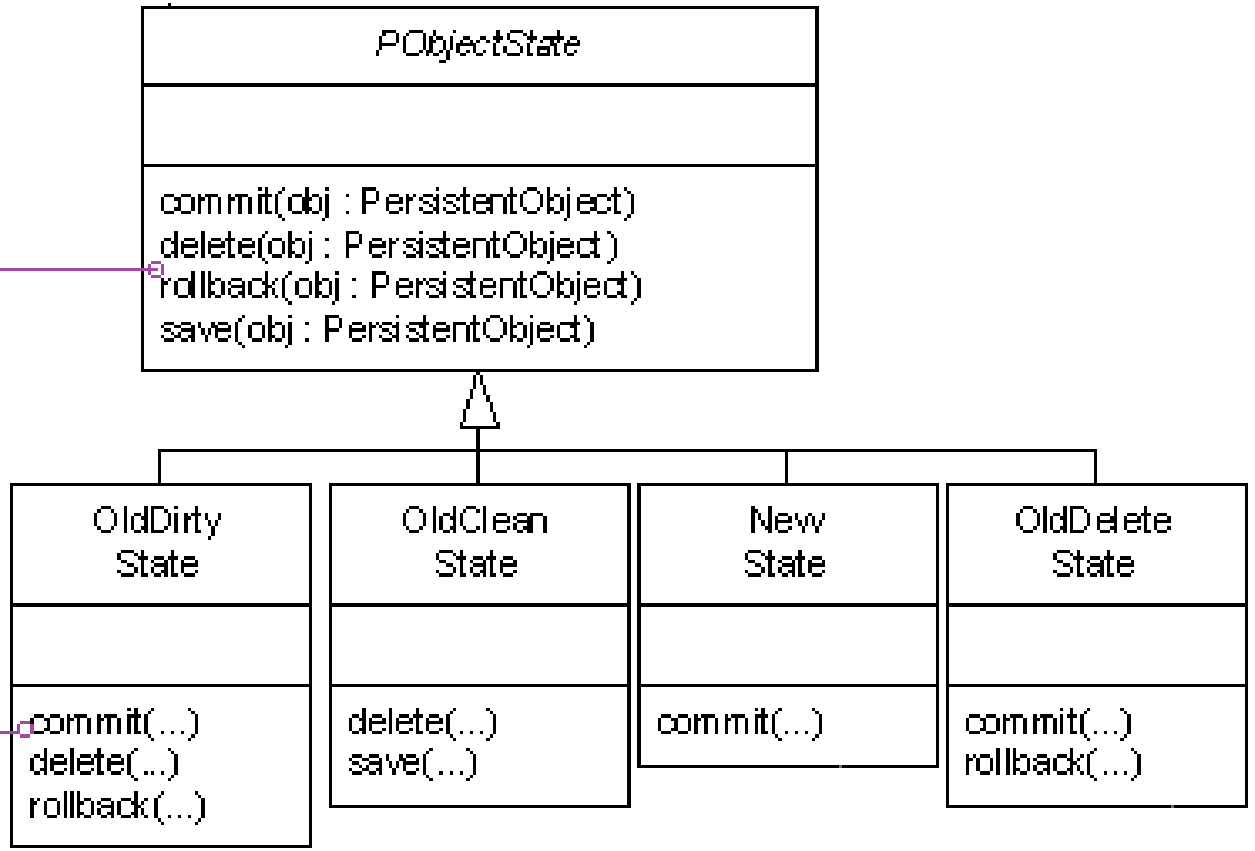
Przykład: stany transakcyjne



Wydajność – stany transakcyjne jako singletony

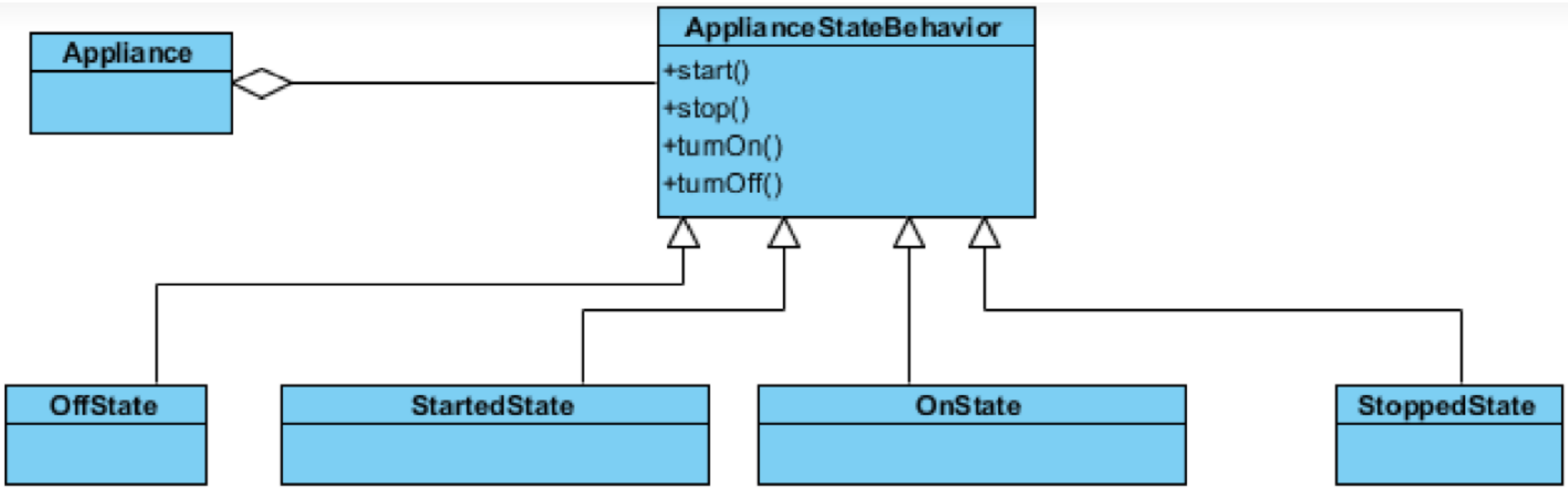
1. Typowo metody klasy stanu przyjmują jako argument referencję do obiektu kontekstowego

2. Zamiast interfejsu można użyć klasy abstrakcyjnej by domyślnie metody stanu miały puste ciała (wygoda)




```
{//commit
DBFacade.getInstance().update(obj);
obj.setState(OldCleanState.getInstance());
```

GoF: State - zadanie na lab



Po co nam fabryki?

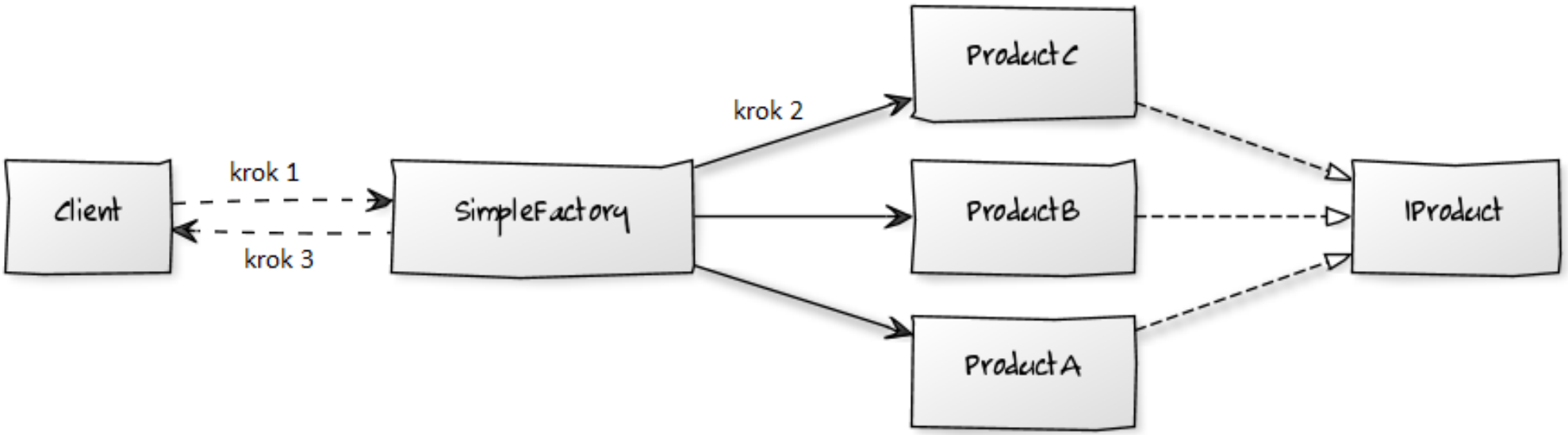
Problem: kto ma odpowiadać za tworzenie obiektów

1. o złożonej logice wytwórczej?
2. odpowiedniego typu?
3. nie związanych z warstwą dziedzinową?

Pomysł:

Utwórzmy obiekt fabryki (Pure Abstraction)
i przydziel mu te zobowiązania (High Cohesion),
po to by zapewnić Protected Variation.

Simple Factory – http://www.yuml.me/



krok 1 Klient wywołuje metodę tworzącą fabryki, najczęściej fabryka to singleton:

$$IProduct\ p = Factory.getInstance().createProduct();$$

krok 2 Wszystkie produkty implementują ten sam interfejs. Fabryka decyduje jaki produkt tworzy (*np. czyta plik konfiguracyjny*)

krok 3 Fabryka zwraca stworzony obiekt do klienta. Klient nie musi wiedzieć jaki obiekt dostanie, ale zna interfejs.

Simple Factory - pierwsze podejście

W kliencie

```
SimpleFactory.getInstance().createProduct('A');
```

```
public class SimpleFactory {
    ...
    public IProduct createProduct(String type) {
        IProduct prod = null;
        if (type.equals("A"))
            prod = new ProductA();
        else
            prod = new ProductB();

        return prod;
```

Jaka zasada jest złamana?

Simple Factory – refleksja

W kliencie

```
SimpleFactory.getInstance().createProduct();
```

```
public class SimpleFactory {
    ...
    public IProduct createProduct() {

     String className = System.getProperty(..) ;

     IProduct prod =
      (IProduct)Class.forName(className).newInstance();

     return prod;
```

GoF: Factory Method - wykorzystanie polimorfizmu

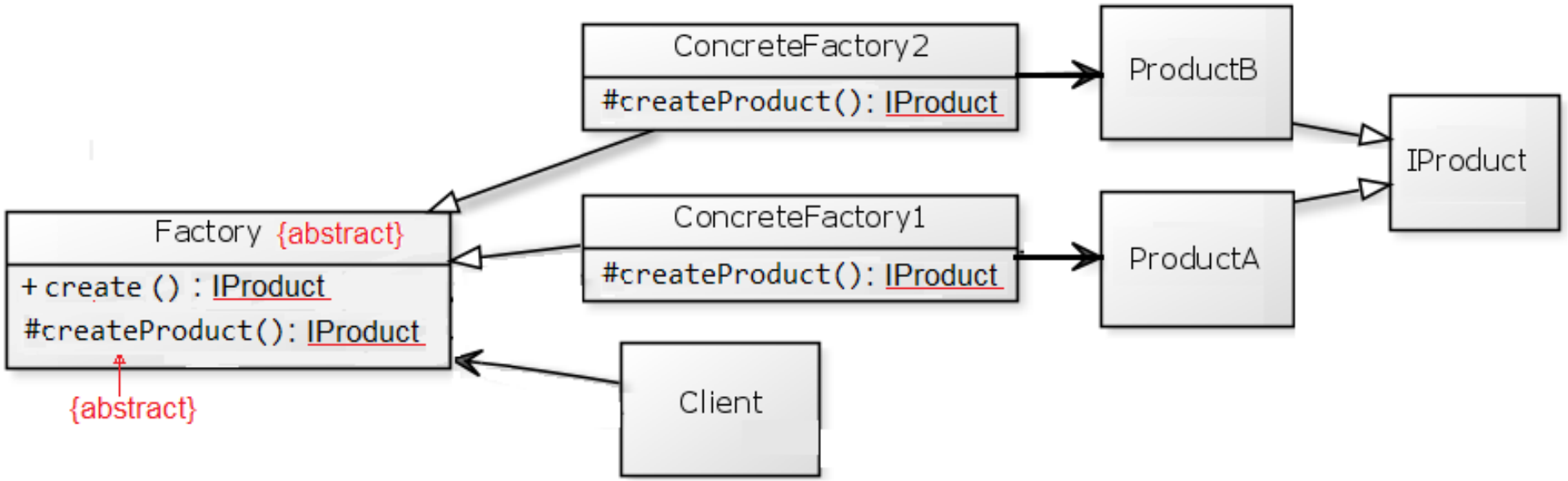
W nadklasie definiujemy ogólną metodę wytwórczą, tzw. metodę szablonową a w podklasach definiujemy konkretną polimorficzną metodę tworzące

```
public abstract class Factory {
    public IProduct create() {
        IProduct prod = createProduct();
        prod.additionalProcessing();
        return prod;
      }
    protected abstract IProduct  createProduct();
```

W kliencie

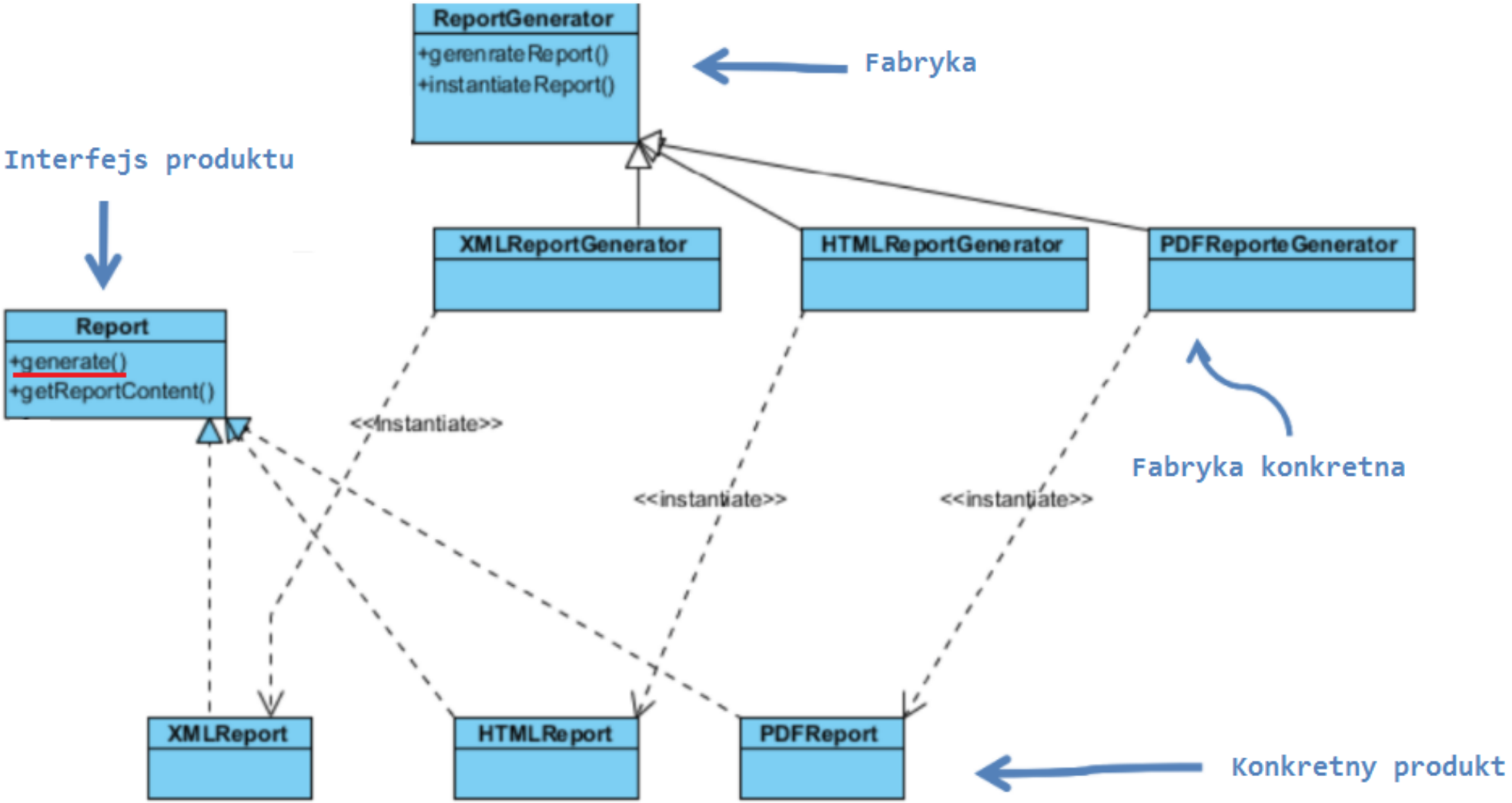
```
Factory f = new ConcreteFactoryA();//runtime decision
IProduct p = f.create()
```

GoF: Factory Method



1. Klient otrzymuje referencje do jednej z fabryk konkretnych (*nie wie z góry której, ale zna interfejs fabryki i produktu*)
2. Abstrakcyjna klasa `Factory` ma chronioną abstrakcyjną metodę `createProduct()` z której korzysta publiczna metoda `create()` (`createProduct()` *jest implmentowana w fabrykach konkretnych*)
3. Fabryka konkretna tworzy produkt implementujący interfejs `IProduct` i zwraca go do klienta przez publiczną metodę `create()`

GoF: Factory Method – przykład na lab



GoF: Factory Method – przykład na lab

```
public abstract class ReportGenerator {
  public Report generateReport(ReportData data, String type) {
    Report generatedReport = intantiateReport();
    generatedReport.generateReport(data);
    return generatedReport;
  }
  protected abstract Report intantiateReport();
}
```

```
public class JSONReportGenerator extends ReportGenerator {
  @Override
    protected Report intantiateReport() {
      return new JSONReport();
  }
}
```

GoF: Abstract Factory

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

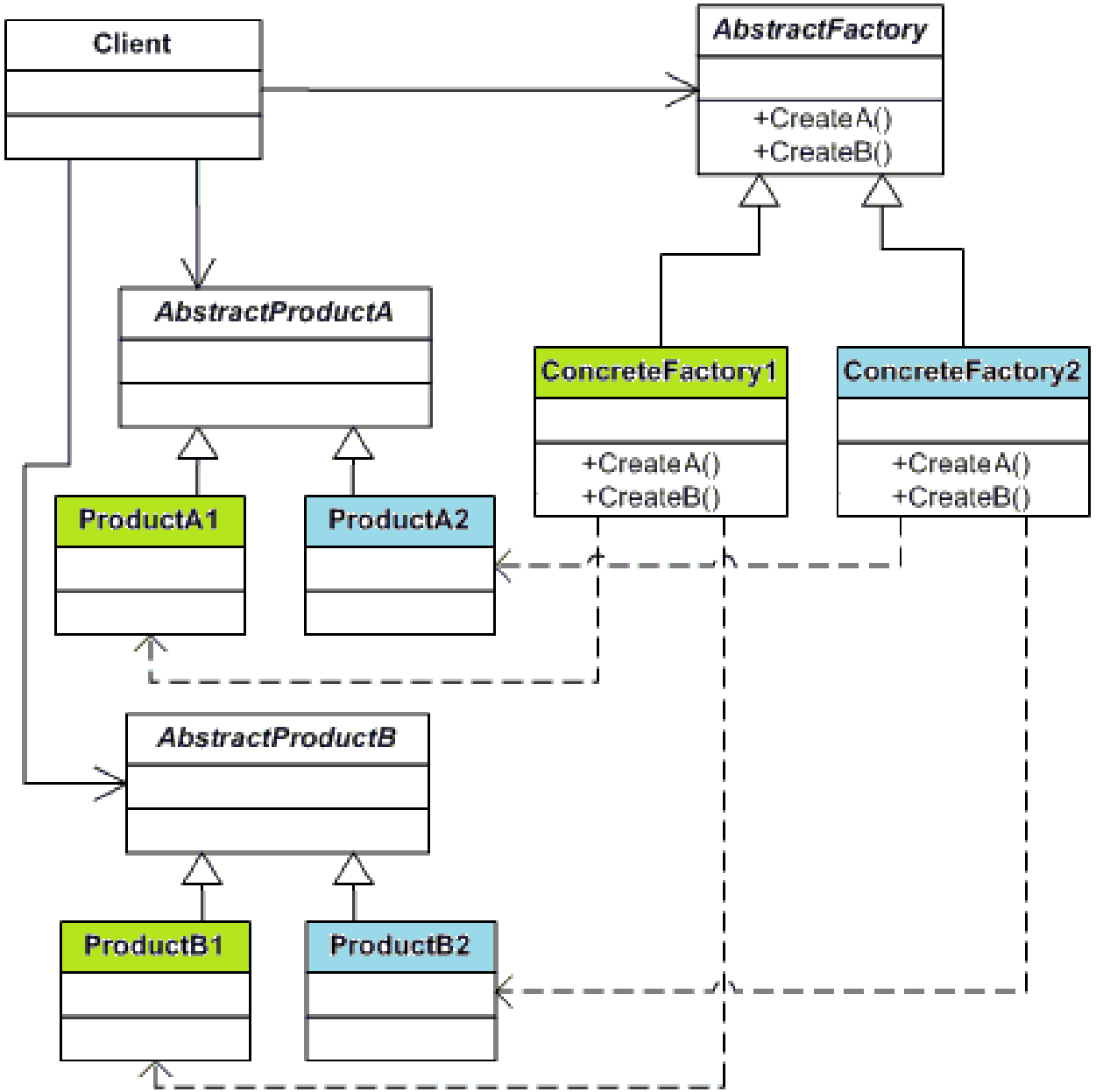
Factory Method

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

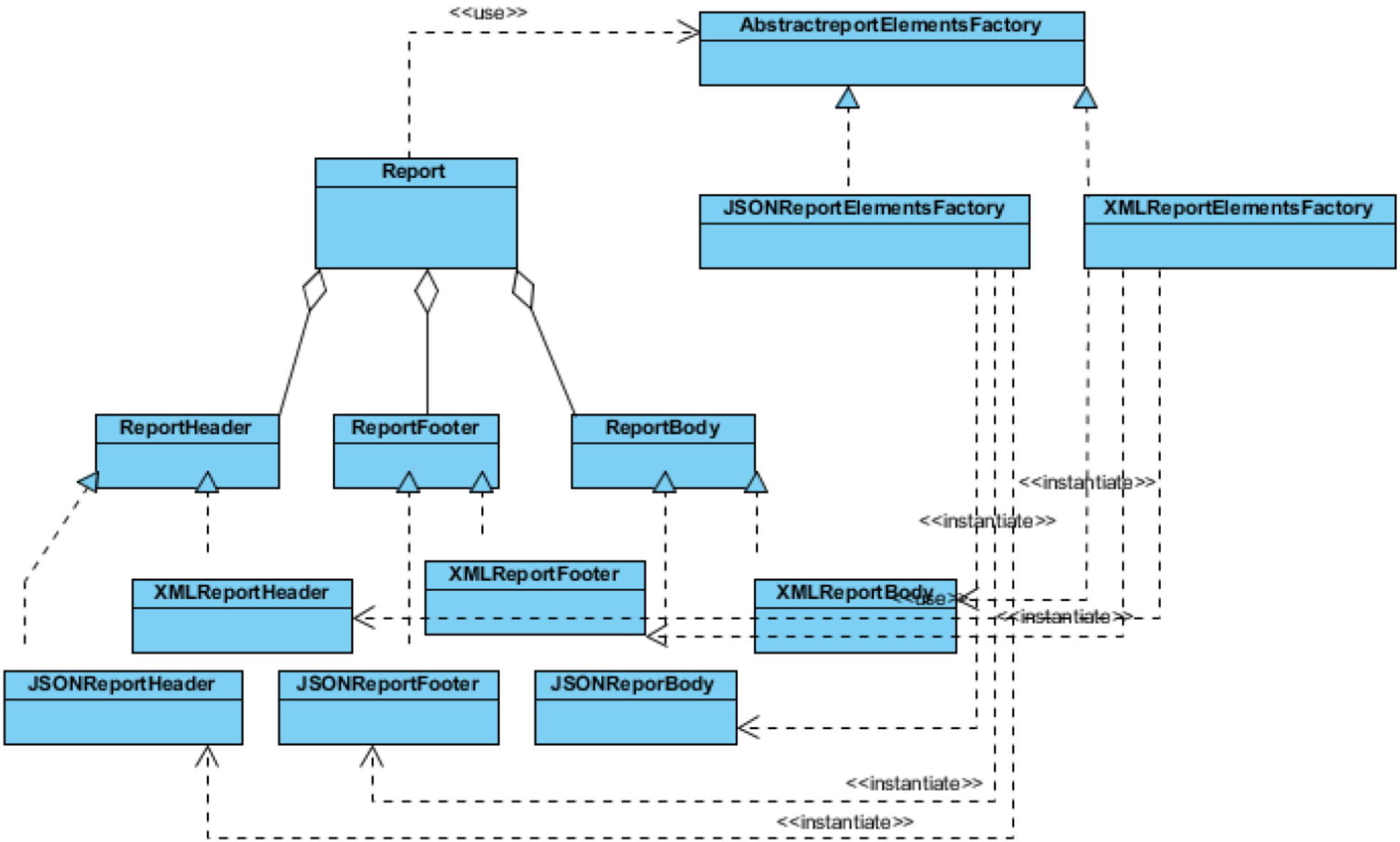
Abstract Factory

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

GoF: Abstract Factory



GoF: Abstract Factory – przykład na lab



GoF: Abstract Factory – przykład na lab

```
public class JSONReportElementsFactory implements AbstractReportElementsFactory {
  @Override
  public ReportBody createReportBody() {
    return new JSONReportBody();
  }
  @Override
  public ReportFooter createReportFooter() {
    return new JSONReportFooter();
  }
  @Override
  public ReportHeader createReportHeader() {
    return new JSONReportHeader();
  }
}
```

```
public class Report {
  public Report(AbstractReportElementsFactory factory) {
    this.setBody(factory.createReportBody());
    this.setFooter(factory.createReportFooter());
    this.setHeader(factory.createReportHeader());
  }
}
```

AbstractFactory – przykład (Larman)

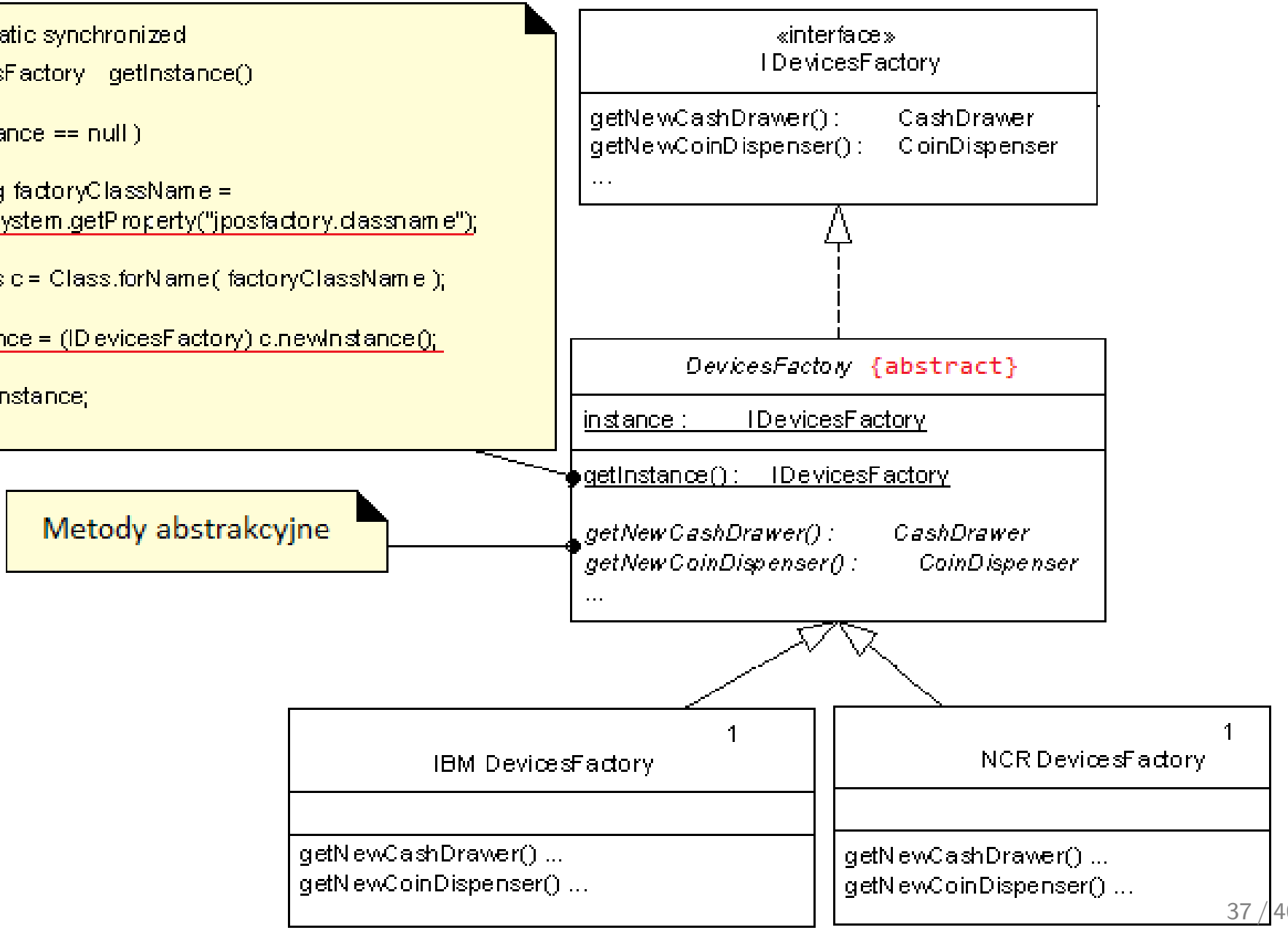
1. Piszemy system do obsługi kas (***P**oint **O**f **S**ale*)
2. Kasy mają różnych producentów, każda kasa wymaga odpowiedniej rodziny sterowników (*np. do szuflady na gotówkę, monet, klawiatury*)
3. Rodzinę sterowników dostarcza producent kasy
4. Jak napisać system możliwie uniwersalnie, by móc łatwo zmieniać kasy bez zmian w systemie?

Abstract Factory – przykład (Larman)

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

   Class c = Class.forName( factoryClassName );

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



AbstractFactory od strony klienta

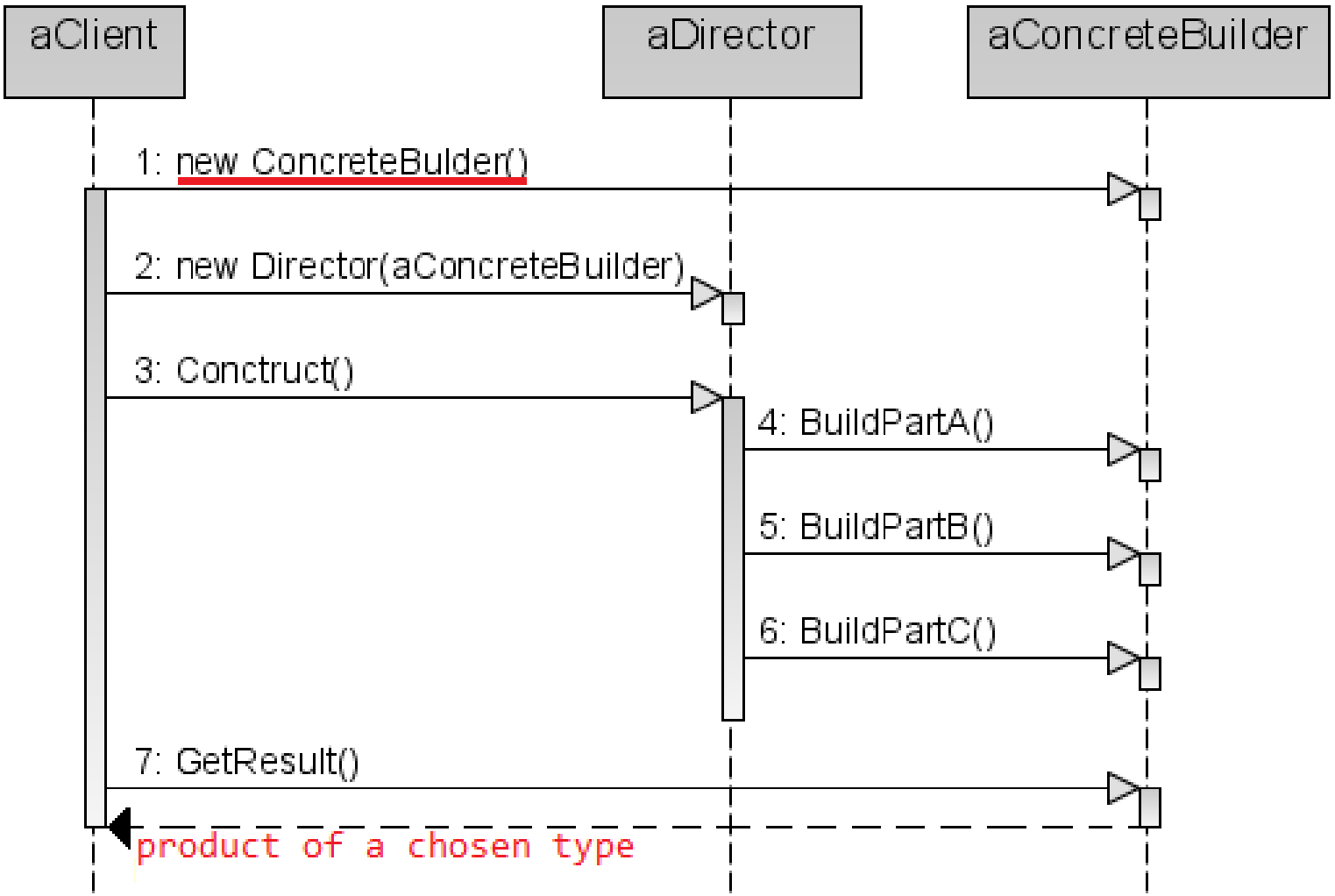
- Zamiast:

```
IDeviceFactory f = new IBMDeviceFactory();
ICashDrawer c = f.getNewCashDrawer();
```

- mamy:

```
IDeviceFactory f = DeviceFactory.getInstance();
ICashDrawer c = f.getNewCashDrawer();
```

GoF: Builder – przypomnienie



Abstract Factory vs. Builder – różnice

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.