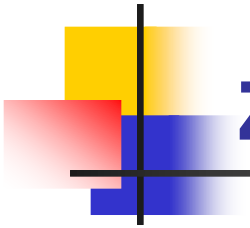
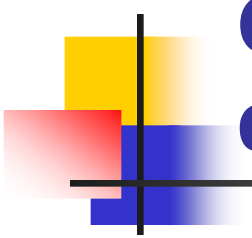


Zagadnienia

- Pojęcie wzorca projektowego
- Sposób opisu wzorców projektowych
- Notacja UML – podstawowe diagramy zapisu wzorców projektowych



Co to jest wzorzec projektowy (ang. design pattern)?

Co to jest wzorzec projektowy (ang. design pattern)?

- Reużywalne, standardowe rozwiązanie zagadnienia/problemu informatycznego, które może pojawić się w procesie wytwarzania/implementacji oprogramowania.
- **Wzorzec projektowy jest:**
  - **Reużywalny** – w takiej samej postaci może zostać wielokrotnie zastosowany do rozwiązywania problemu bez względu na dziedzinę projektu/oprogramowania
  - **Sprawdzony** – przydatność wzorca potwierdzona doświadczeniem nabytym poprzez jego zastosowanie w wielu projektach informatycznych – optymalne rozwiązanie danego zagadnienia
  - **Powszechnie znany** – ułatwia wymianę informacji i zwiększa czytelność kodu

Historia wzorców projektowych

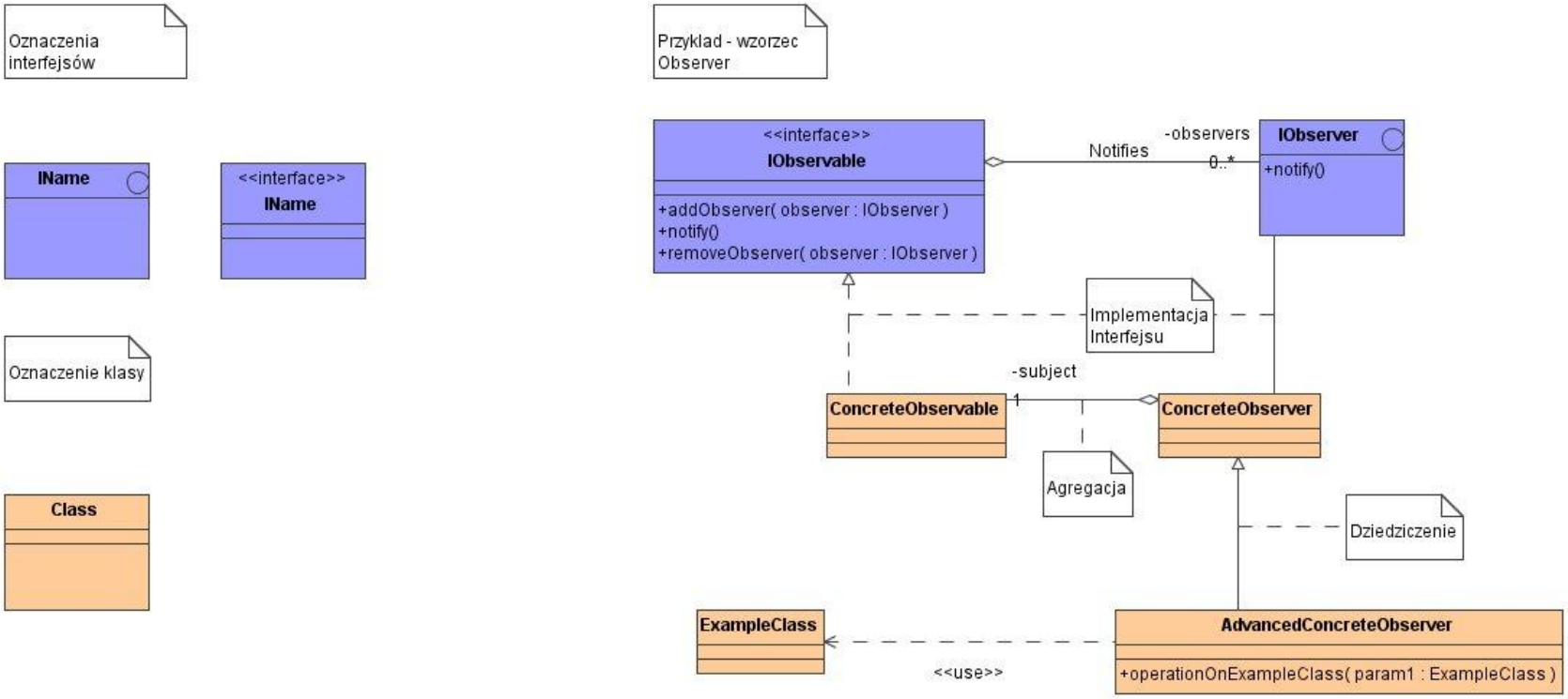
- Christopher Alexander – wzorce w budownictwie opisane w roku 1977 "A Patterns Language: Towns, Buildings, Construction"
- Ward Curningham i Kent Beck – pierwsze wzorce dla interfejsu graficznego "Using Pattern Languages for Object-Oriented Programms" 1987
- W roku 1994 publikacja najważniejszej książki opisującej i klasyfikującej zbiór wzorców projektowych dla zagadnień informatycznych "Design Patterns" - Erich Gamma, Richard Helma, Ralph Johnson, John Vlissides (nazywana również "Gang of Four book" lub "GoF book")

Opis wzorca projektowego

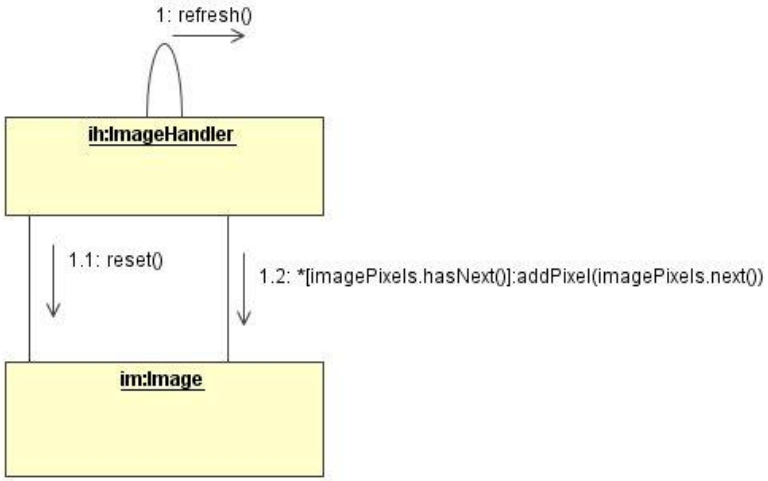
- Nazwa wzorca
- Skrótowy opis wzorca
- Opis problemu – może również zawierać konkretny przykład rozwiązania problemu oraz opis kontekstu
- Ogólne rozwiązanie stanowiące wzorzec
- Konsekwencje wykorzystania wzorca
- Lista wzorców związanych z opisywanym wzorcem

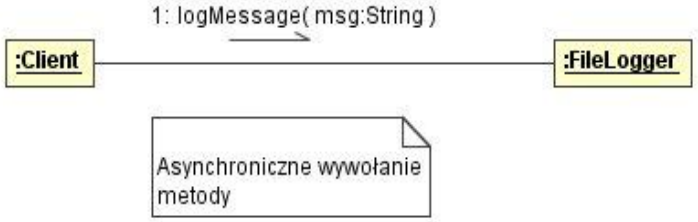
Opisy wzorców projektowych różnią się między sobą w zależności od katalogu wzorców, w którym dany wzorzec został opisany i opublikowany

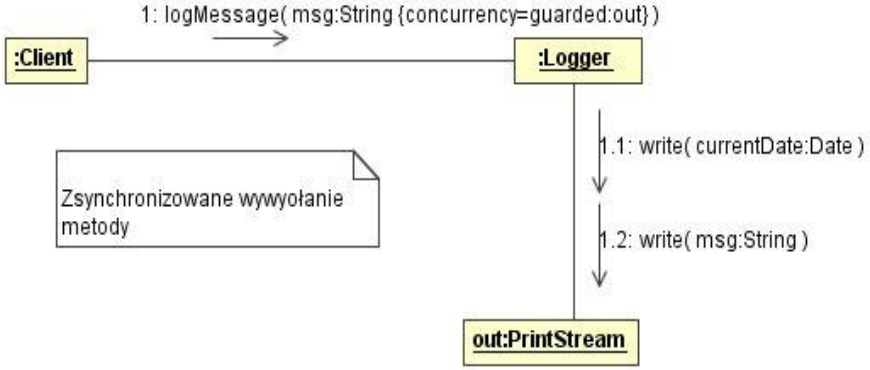
Notacja UML – diagramy klas



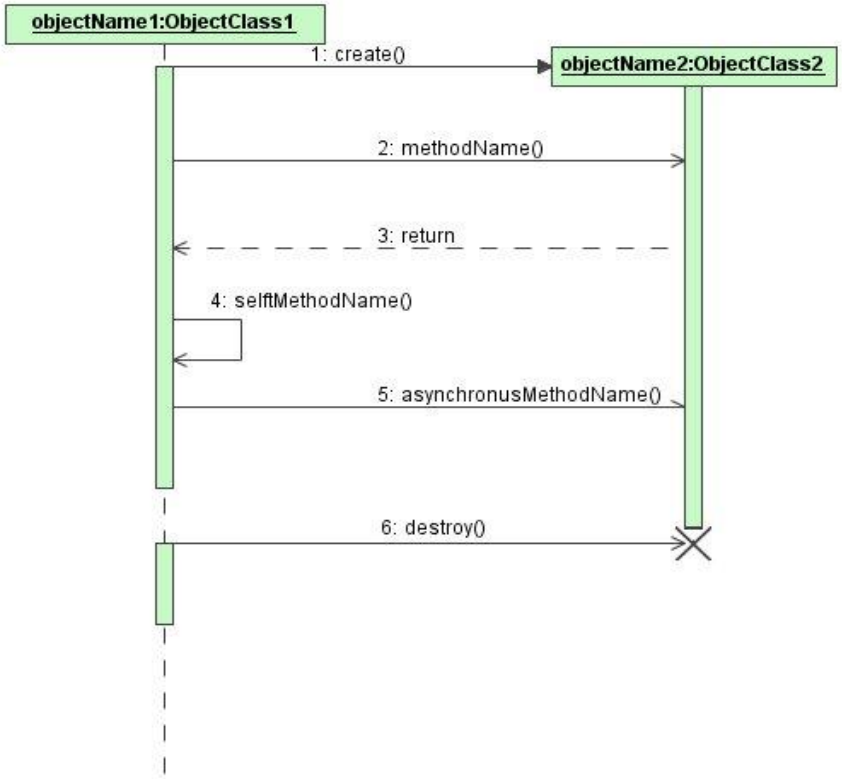
Notacja UML – diagramy kolaboracji (przypomnienie)







Notacja UML – diagramy sekwencji(przypomnienie)



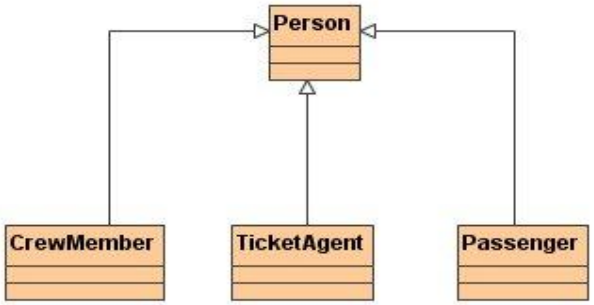
Klasyfikacje wzorców projektowych

- Design Patterns – podstawowa klasyfikacja:
  - Fundamental – wzorce podstawowe wykorzystywane przez wiele innych wzorców
  - Creational – tworzenie instancji obiektów w przypadku gdy proces ten realizowany jest w oparciu o pewne decyzje
  - Partitioning – podział złożonych zagadnień/koncepcji na szereg mniejszych wzajemnie współpracujących klas
  - Structural – wzajemna współpraca różnych typów obiektów
  - Behavioral – organizacja i zarządzanie zachowaniem obiektów
  - Concurrency – zarządzanie i koordynacja równoległego wykonania zadań

Wzorce projektowe – delegation (fundamental pattern) 1/4

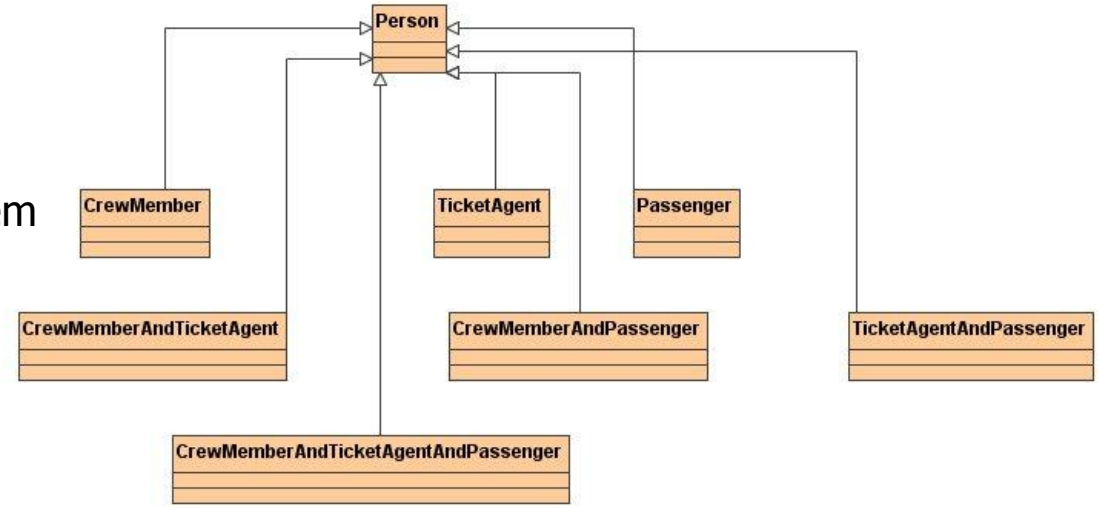
- **Nazwa wzorca:** delegation
- **Opis:** wzorzec *delegation* opisuje metodę rozszerzania i reużywania funkcjonalności klasy bazowej poprzez tworzenie nowych klas z dodatkową funkcjonalnością, które wykorzystują instancje bazowej klasy w celu dostarczenia bazowej funkcjonalności.

Wzorce projektowe – delegation(fundamental pattern) 2/4

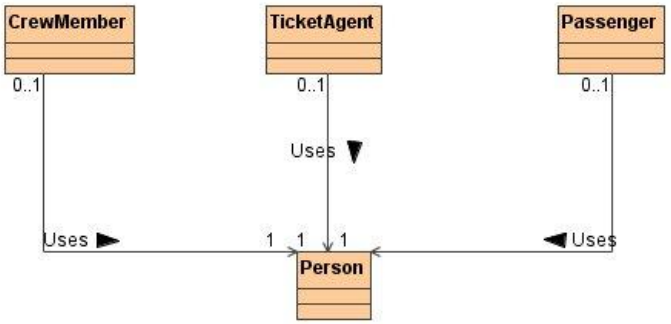


Modelowanie ról z wykorzystaniem dziedziczenia

Modelowanie wielu ról z wykorzystaniem dziedziczenia



Wzorce projektowe – delegation(fundamental pattern) 3/4



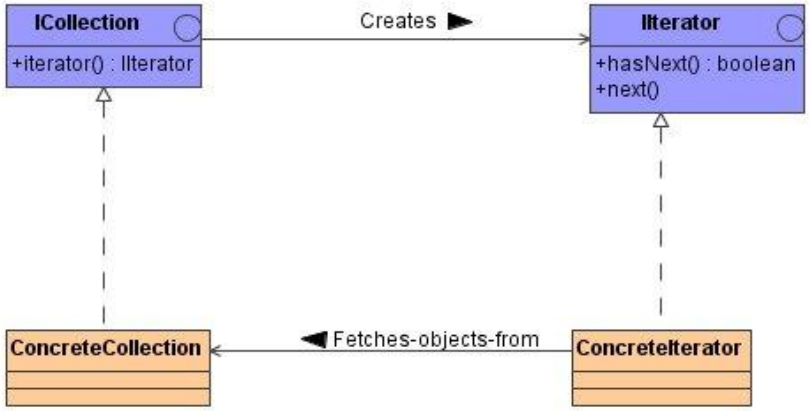
Modelowanie ról z wykorzystaniem delegacji

Wzorce projektowe – delegation(fundamental pattern) 4/4

- **Konsekwencje:**
  - Możliwość dynamicznej zmiany zachowania klasy w trakcie wykonania
  - Zależności między klasami mniej strukturalne w porównaniu do relacji dziedziczenia
- **Związane wzorce:**
  - Znaczna część innych wzorców wykorzystuje wzorzec delegacji (dla przykładu: Decorator, Proxy)

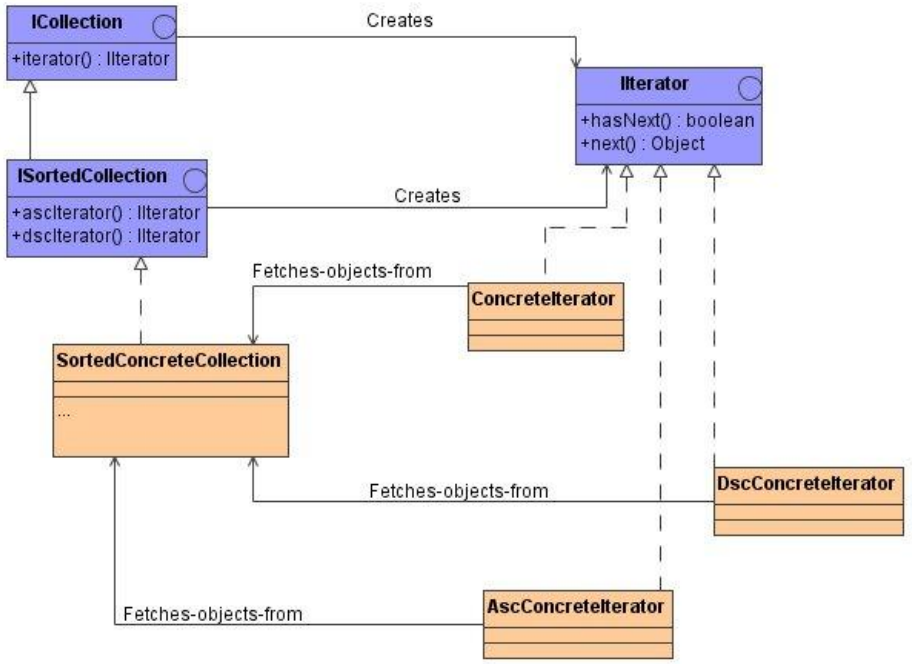
Wzorce projektowe – iterator (structural pattern) 1/3

- **Nazwa wzorca:** iterator
- **Opis:** wzorzec *iterator* definiuje interfejs zawierający deklaracje metod umożliwiających sekwencyjny dostęp do kolejnych wartości zawartych w kolekcji. Klasa uzyskująca dostęp do kolekcji poprzez taki interfejs uniezależnia się od szczegółów implementacji kolekcji.



Wzorce projektowe – iterator (structural pattern) 2/3

- **Konsekwencje:**
  - Możliwość dostępu do obiektów kolekcji bez znajomości źródła danych
  - Możliwość definicji różnych rodzajów iteratorów dla tego samego typu kolekcji
- **Związane wzorce:**
  - Adapter
  - Factory Method
  - Null Object



Wzorce projektowe – iterator (structural pattern) 3/3

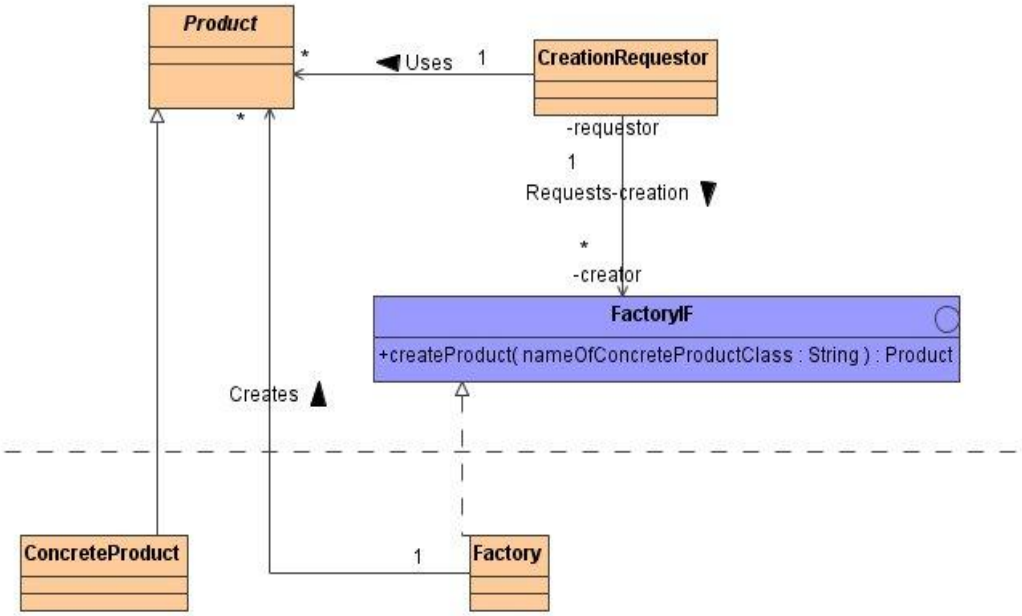
```
/**
 * An iterator over a collection.  Iterator takes the place of Enumeration in
 * the Java collections framework.  Iterators differ from enumerations in two
 * ways: <ul>
 *      <li> Iterators allow the caller to remove elements from the
 *           underlying collection during the iteration with well-defined
 *           semantics.
 *      <li> Method names have been improved.
 * </ul>
 *
 *
 */
public interface Iterator {
    /**
     * Returns <tt>true</tt> if the iteration has more elements. (In other
     * words, returns <tt>true</tt> if <tt>next</tt> would return an element
     * rather than throwing an exception.)
     *
     * @return <tt>true</tt> if the iterator has more elements.
     */
    boolean hasNext();

    /**
     * Returns the next element in the iteration.
     *
     * @return the next element in the iteration.
     * @exception NoSuchElementException iteration has no more elements.
     */
    Object next();

    /**
     *
     * Removes from the underlying collection the last element returned by the
     * iterator (optional operation).  This method can be called only once per
     * call to <tt>next</tt>.  The behavior of an iterator is unspecified if
     * the underlying collection is modified while the iteration is in
     * progress in any way other than by calling this method.
     *
     * @exception UnsupportedOperationException if the <tt>remove</tt>
     *              operation is not supported by this Iterator.
     * @exception IllegalStateException if the <tt>next</tt> method has not
     *              yet been called, or the <tt>remove</tt>
     *                                  method has already
     *              been called after the last call to the
     *                                  <tt>next</tt> method.
     */
    void remove();
}
```

Wzorce projektowe – factory method (creational pattern) 1/2

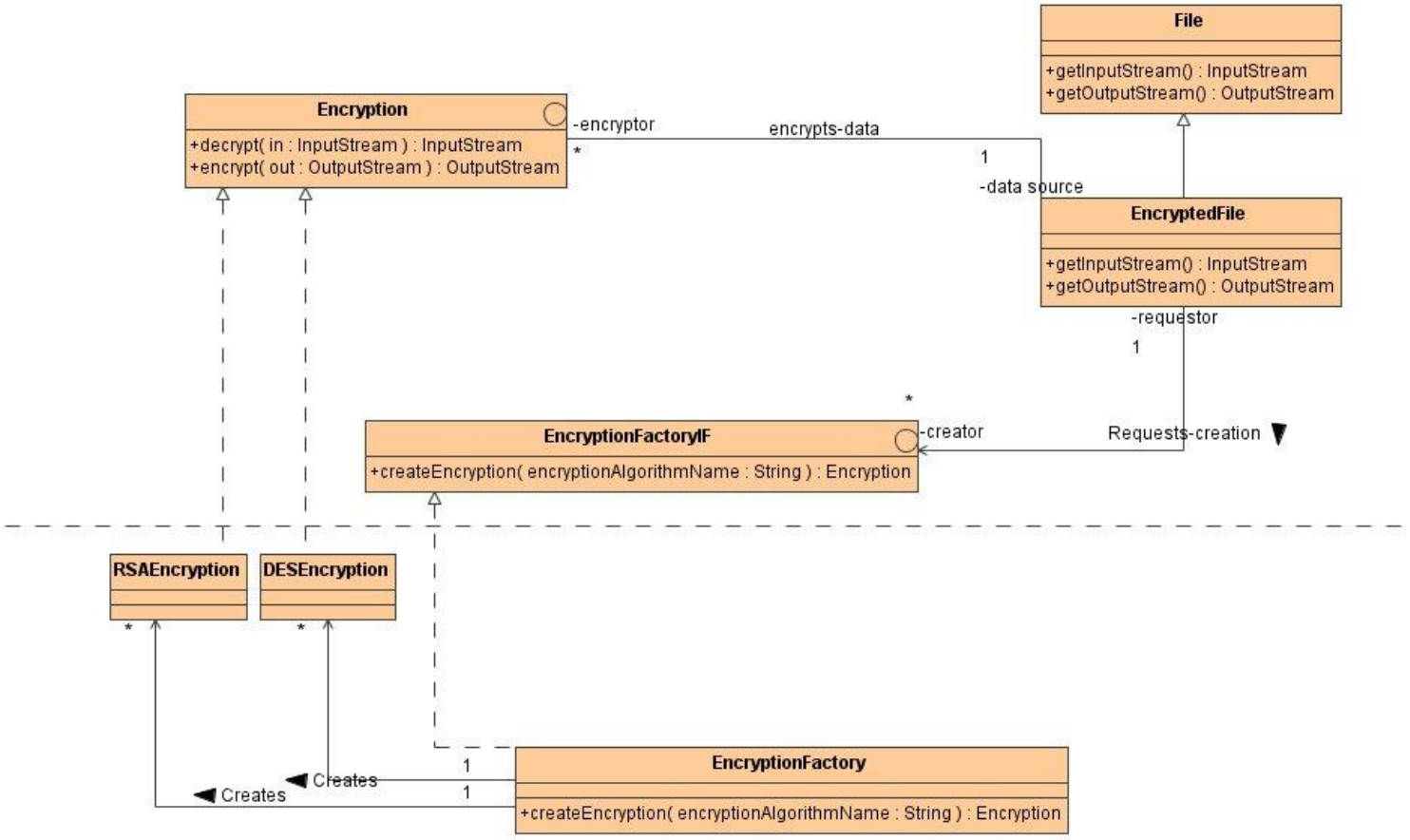
- **Nazwa wzorca: factory method**
- **Opis:** umożliwienie tworzenia instancji klas bez potrzeby znajomości definicji klas. Klasa korzystająca z factory method pozostaje niezależna od implementacji klas, które tworzy odwołując się do nich przez wspólny, dobrze zdefiniowany interfejs dostępowy.



Wzorce projektowe – factory method (creational pattern) 2/2

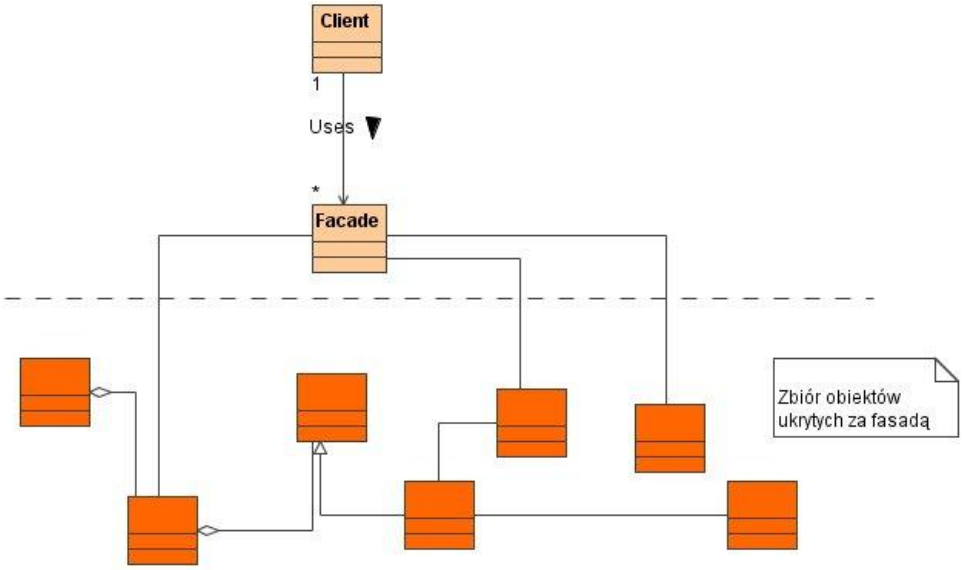
- **Konsekwencje:**
  - Klasa "kliencka" wykorzystująca factory method (CreationRequestor) nie jest zależna od konkretnych implementacji klas, które są dynamicznie tworzone
  - Zbiór tworzonych klas może być dynamicznie zmieniany
- **Związane wzorce:**
  - Abstract Factory
  - Prototype
  - Template Method

Wzorce projektowe – factory method przykład



Wzorce projektowe – façade (structural pattern) 1/2

- **Nazwa wzorca: façade**
- **Opis:** uproszczenie dostępu do zbioru wzajemnie powiązanych obiektów poprzez jeden wspólny obiekt. Obiekt ten wykorzystywany jest przez wszystkie inne zewnętrzne obiekty do komunikacji ze zbiorem obiektów "ukrytych" za fasadą.



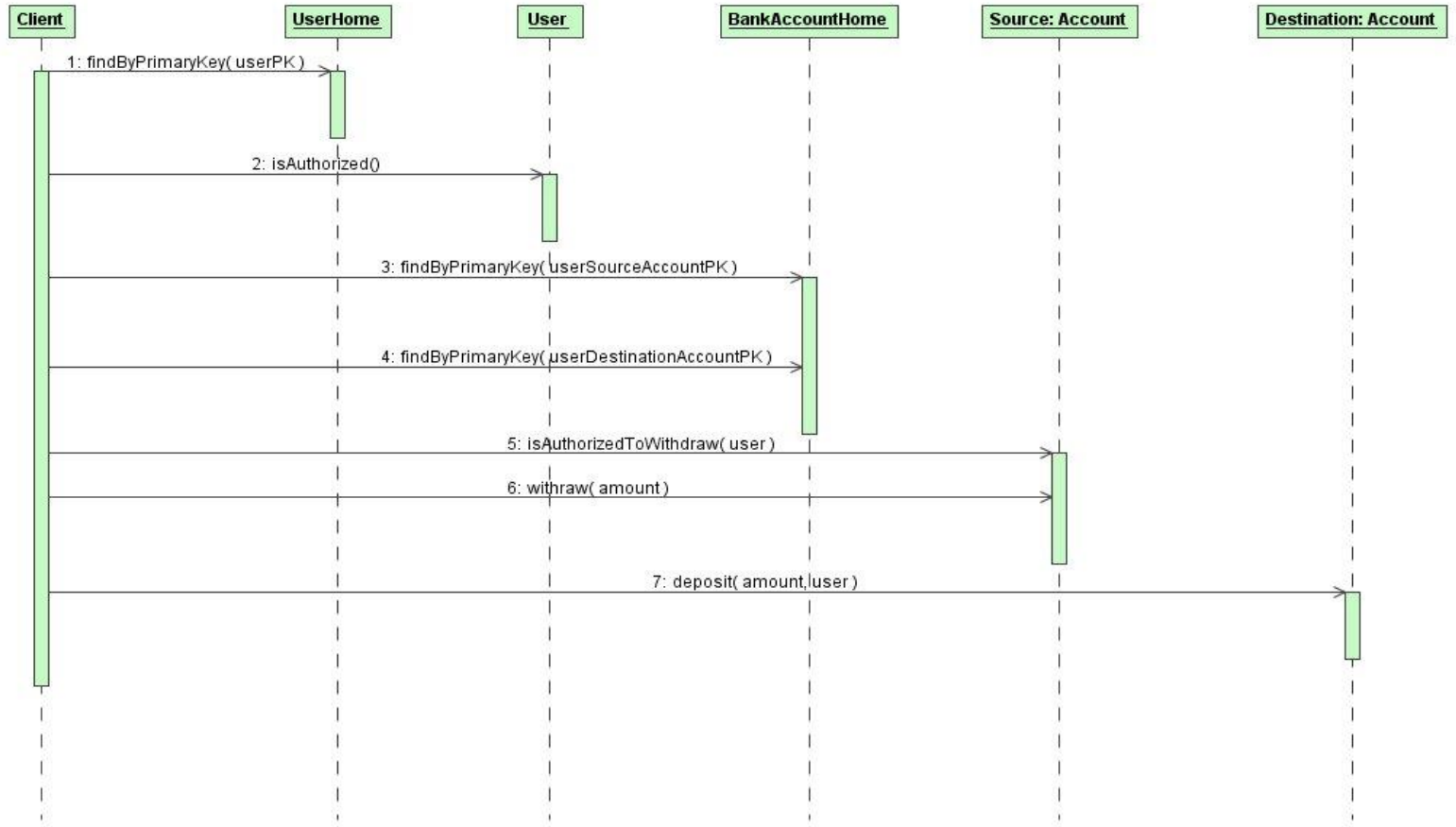
Wzorce projektowe – façade (structural pattern) 2/2

- **Konsekwencje:**
  - Zewnętrzni klienci obiektu fasady nie muszą "znać" żadnej z klas obiektów ukrytych za fasadą
  - Zmniejszenie lub nawet eliminacja powiązania pomiedzy klasami implementującymi funkcjonalność udostępnianą przez klasę obiektu fasady powoduje, że zmiana sposobu implementacji funkcjonalności nie wpływa na zewnętrznych klientów fasady
- **Związane wzorce:**
  - Interface
  - Don't talk to strangers

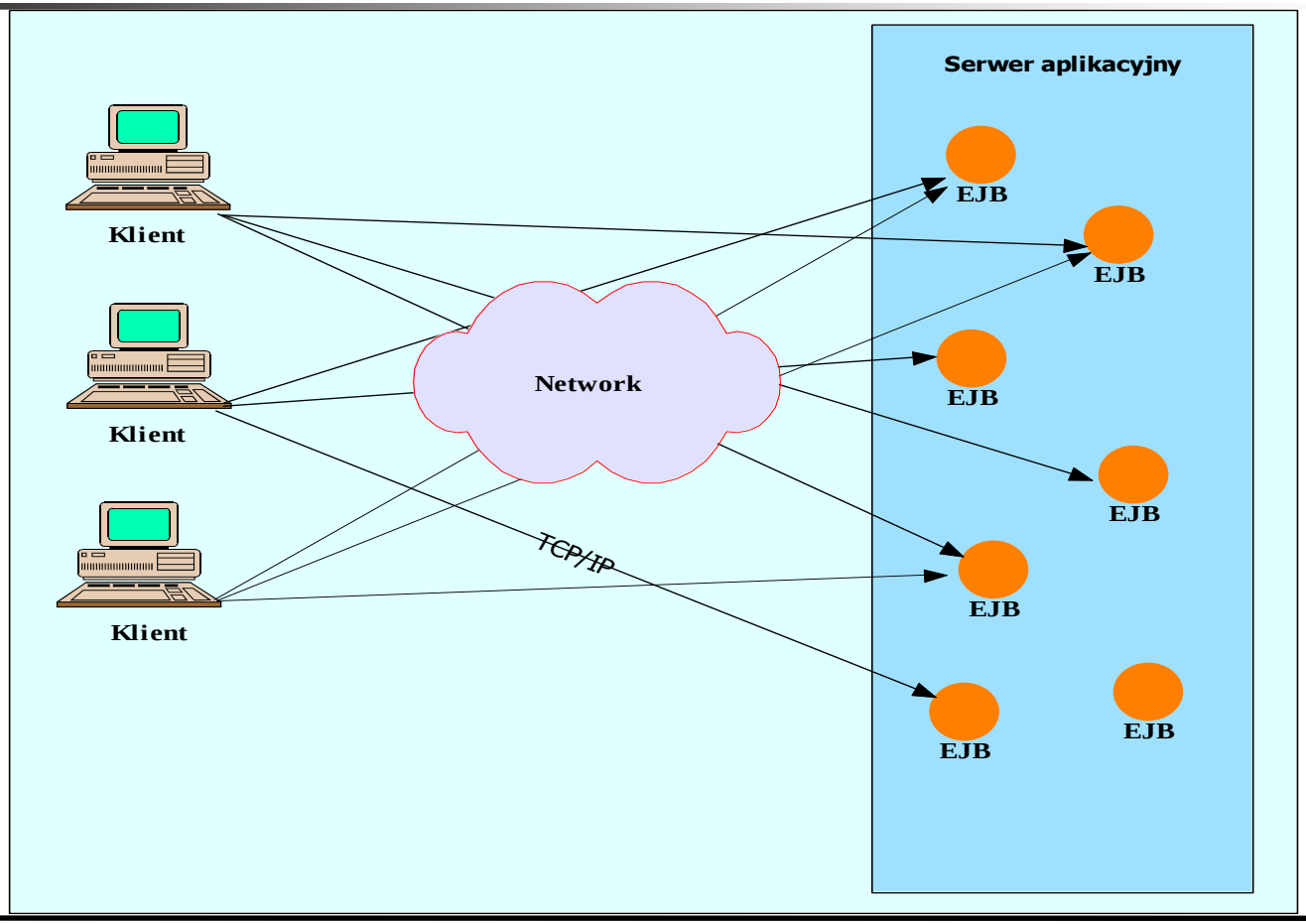
Przykład adaptacji wzorca façade – EJB session façade 1/7

- Problem: Enterprise Java Beans umożliwia implementację logiki biznesowej z wykorzystaniem komponentów EJB (tzw. session i/lub entity beans) wykonywanych po stronie serwera. Wykonanie logiki biznesowej typowego usecase'u wymaga wielokrotnego dostępu do różnych beanów sesyjnych i/lub entity. Wielokrotne wywołania dają znaczący narzut związany z komunikacją poprzez sieć (RMI) jak również zarządzalność kodem staje się utrudniona gdyż każda z aplikacji klienckich musi implementować przepływ sterowania i logikę biznesową.

Przykład adaptacji wzorca façade – EJB session façade 2/7



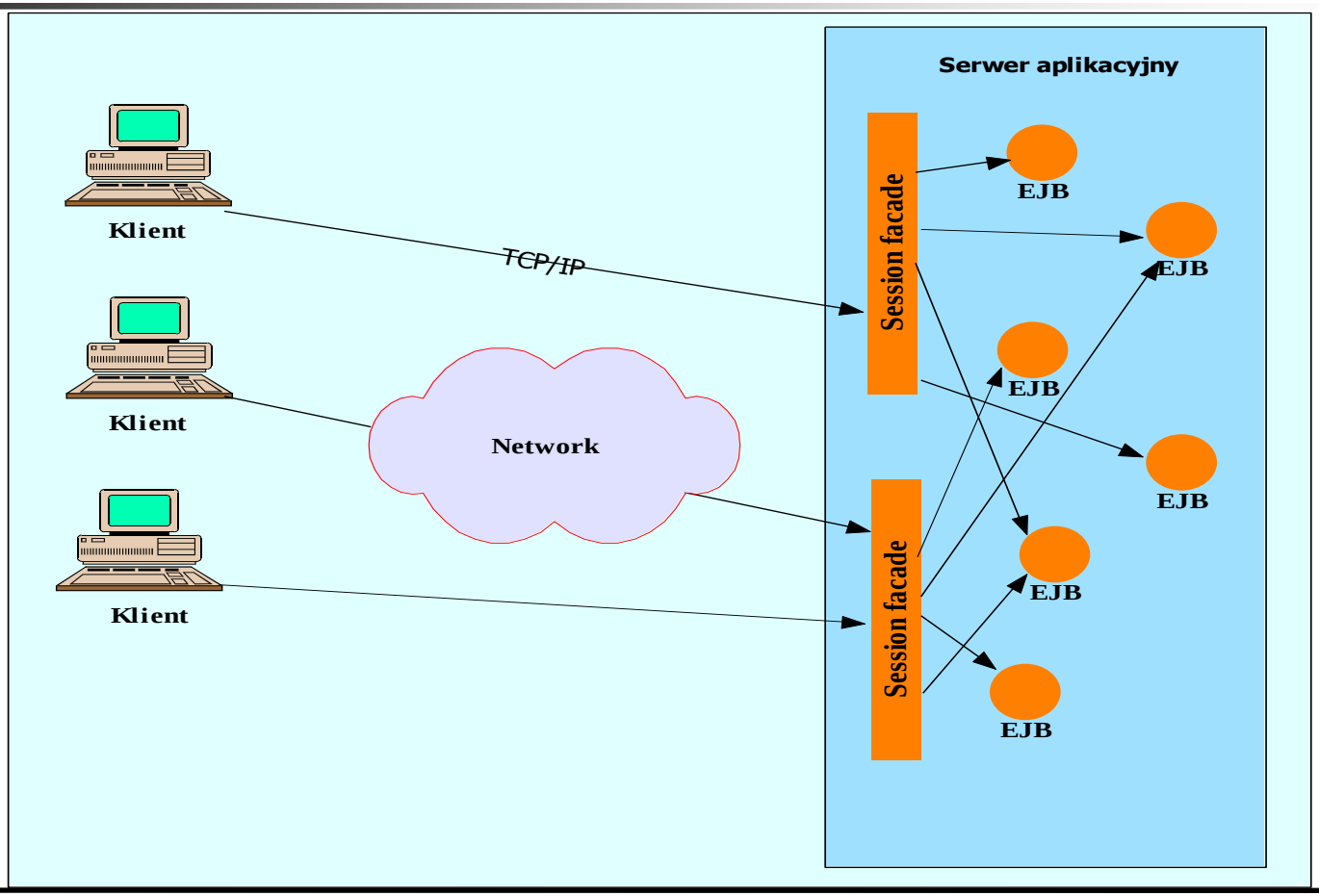
Przykład adaptacji wzorca façade – EJB session façade 3/7



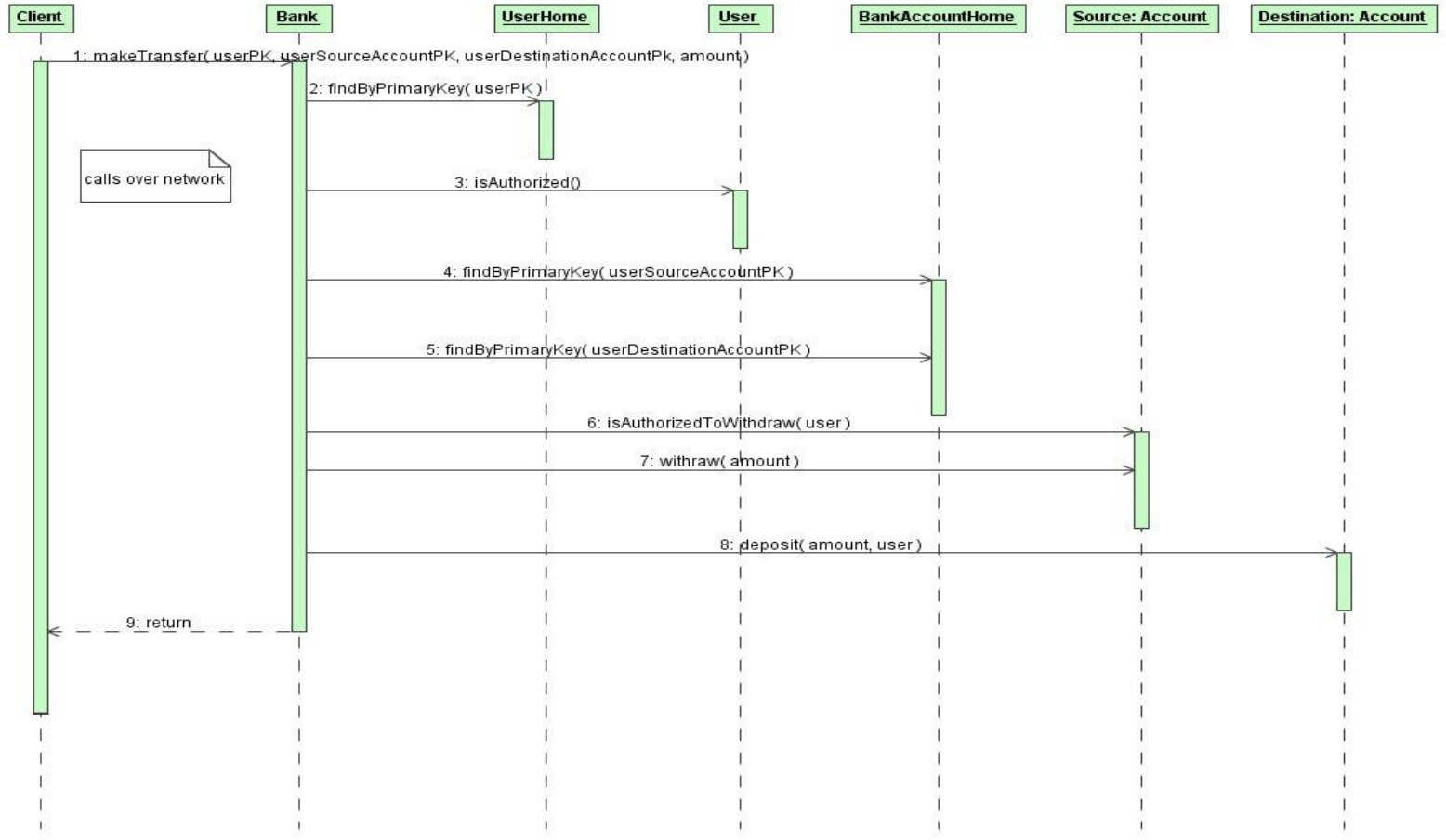
Przykład adaptacji wzorca façade – EJB session façade 4/7

- Rozwiązanie: Implementacja warstwy logiki biznesowej w postaci tzw. fasady beanów sesyjnych. Kod logiki biznesowej zostanie przeniesiony z aplikacji klienckich do warstwy fasady. Aplikacje klienckie mają dostęp tylko do funkcjonalności udostępnianej przez publiczny interfejs fasady.

Przykład adaptacji wzorca façade – EJB session façade 5/7



Przykład adaptacji wzorca façade – EJB session façade 6/7



Przykład adaptacji wzorca façade – EJB session façade 7/7

- Zalety:
  - Zmniejszenie narzutu związanego z komunikacją poprzez sieć
  - Zwiększenie wydajności
  - Zwiększenie czytelności architektury poprzez wyraźną separację warstwy logiki od prezentacji
  - Zmniejszenie powiązania pomiędzy warstwami
  - Wzrost reużywalności – logika biznesowa zaimplementowana w postaci fasady dostępna dla różnych typów aplikacji klienckich (np. JSP, servletów, apletów, aplikacji)
  - Poprawa zarządzalności

Bibliografia

- GoF, “Design Patterns”
- Bertrand Meyer, “Applying 'Design by Contract.'”, Computer (IEEE), 25(10):40-51. October 1992
- William H. Brown, Raphael C. Malveau “Anti Patterns – Refactoring Software, Architecture and Projects in Crisis”, Wiley Computer Publishing
- Frank Bauhman “A System of Patterns – Pattern Oriented Software Architecture”, Wiley Computer Publishing
- Mark Grand “Patterns in Java” vol. 1-2, Wiley Computer Publishing