Diagramy klas, diagramy sekwencji

Zofia Kruczkiewicz

Składnia elementów na diagramach UML

1. W prezentacji składni diagramów sekwencji (str. 25-35) o charakterze tutorialowym sposób definiowania składowych klas (atrybuty, operacje, parametry operacji) jest jednym z przyjętych sposobów interpretowania specyfikacji języka UML w tutorialach – często odbiegająca od syntaktyki znanych języków obiektowych (Java, C++) i zazwyczaj uproszczona.
2. W prezentacji przykładów diagramów klas UML na str. 18, 19, 50, 61, 75, 85 oraz diagramów sekwencji UML na str. 38-83 sposób definiowania składowych klas jest jednym z kolejnych przyjętych sposobów interpretowania specyfikacji języka UML w narzędziach UML. Składnia tych diagramów różni się od prezentowanych w tutorialach (p.1) i jest zbliżona do składni języka Java. **Diagramy klas i sekwencji uzyskano generując diagramy z kodu Javy.**

Wniosek: W wielu narzędziach UML sposób definiowania elementów diagramów oparty na tej samej specyfikacji UML różni się. W prezentowanych materiałach przedstawiono te różnice, stosując dwa różne sposoby definiowania oparte na:

1) tutorialach (p.1): **http://sparxsystems.com.au/resources/uml2_tutorial/**
2) narzędziu z serii Visual Paradigm VP CE (np instrukacja do lab1: **http://zofia.kruczkiewicz.staff.iiar.pwr.wroc.pl/wyklady/IO_UML/Instrukcja_1_2.pdf**

W mat. 2) diagramy klas i sekwencji zostały wygenerowane z kodu Javy (inzynieria odwrotna), w celu zwrócenia uwagi, że te różnice są naturalnym zjawiskiem, ale zawsze wspierającym programistów.

Diagramy klas, diagramy sekwencji

1. **Identyfikacja elementów diagramów klas – część 2**

   [Shalloway A.,Trott James R.,Projektowanie zorientowane obiektowo. Wzorce projektowe. Gliwice, Helion, 2005]

2. **Diagramy sekwencji UML**

   http://sparxsystems.com.au/resources/uml2_tutorial/

3. **Przykłady diagramów sekwencji – kontynuacja przykładu 1 z wykładu 2 i wykładu 3**

Diagramy klas, diagramy sekwencji

1. **Identyfikacja elementów diagramów klas – część 2**

   [Shalloway A.,Trott James R.,Projektowanie zorientowane obiektowo. Wzorce projektowe. Gliwice, Helion, 2005]

Dwa rodzaje diagramów UML 2

Diagramy UML modelowania strukturalnego

- Diagramy pakietów
- *Diagramy klas*
- Diagramy obiektów
- Diagramy mieszane
- Diagramy komponentów
- Diagramy wdrożenia

Diagramy UML modelowania zachowania

- *Diagramy przypadków użycia*
- *Diagramy czynności*
- Diagramy stanów
- Diagramy komunikacji
- Diagramy sekwencji
- Diagramy czasu
- Diagramy interakcji

Modele analizy i projektu – typy klas na diagramach klas

Produkt	Opis produktu (reprezentowanego w języku UML)
a) klasy typu „Control” Warstwy: prezentacji **biznesowa,** integracji	•reprezentują koordynację (*coordination*), sekwencje (*sequencing*), transakcje (*transactions*), sterowanie (*control*) •często są używane do hermetyzacji sterowania odniesionego do przypadku użycia dla każdej warstwy tzn hermetyzują warstwę biznesową dla warstwy prezentacji oraz warstwę integracji dla warstwy biznesowej; •klasy te modelują dynamikę systemu czyli główne akcje (*actions*) i przepływ sterowania (*control flows*) i przekazują działania do klas warstwy prezentacji, biznesowej oraz integracji ;

b) klasy typu „Entity” - formalnie obiekty realizowane przez system, często przedstawiane jako logiczne struktury danych (*logical data structure*) Warstwa biznesowa	• używane do modelowania informacji o długim okresie istnienia i często niezmiennej (*persistent*); • klasy realizowana jako obiekty typu „real-life” lub zdarzenia typu „real-life”; • są wyprowadzane z modelu analizy • mogą zawierać specyfikację złożonego zachowania reprezentowanej informacji
c) klasy typu „Boundary” Warstwa klienta	• klasy te reprezentują abstrakcje: okien, formularzy, interfejsów komunikacyjnych, interfejsów drukarek, sensorów, terminali i API (również nieobiektowych); • jedna klasa odpowiada jednemu użytkownikowi typu aktor • używane do modelowania interakcji między systemem i aktorami czyli użytkownikami (*users*) lub zewnętrznymi systemami;

Identyfikacja klas

(wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)

- Zidentyfikuj zbiór klas, które współpracują ze sobą w celu wykonania poszczególnych czynności
- Określ zbiór zobowiązań każdej klasy
- Rozważ zbiór klas jako całość: podziel na mniejsze te klasy, które mają zbyt wiele zobowiązań; scal w większe te klasy, które mają zbyt mało zobowiązań
- Rozpatrz sposoby wzajemnej kooperacji tych klas i porozdzielaj ich zobowiązania tak, aby żadna z nich była ani zbyt złożona ani zbyt prosta
- Elementy nieprogramowe (urządzenia) przedstaw w postaci klasy i odróżnij go za pomocą własnego stereotypu; jeśli ma on oprogramowanie, może być traktowany jako węzeł diagramu klas w celu rozwijania tego oprogramowania
- Zastosuj typy pierwotne (tabele, wyliczenia, typy proste np. boolean itp)

Identyfikacja związków: zależność (Dependency)

(wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)

Modelowanie zależności

- Utworzyć zależności między klasą z operacją, a klasą użytą jako parametr tej operacji
- Stosuj zależności tylko wtedy, gdy modelowany związek nie jest strukturalny

Identyfikacja związków: generalizacja czyli dziedziczenie (Generalization)

(wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)

- Ustaliwszy zbiór klas poszukaj zobowiązań, atrybutów i operacji wspólnych dla co najmniej dwóch klas
- Przenieś te wspólne zobowiązania, atrybuty i operacje do klasy bardziej ogólnej; jeśli to konieczne, utwórz nową klasę, do której zostaną przypisane te właśnie byty (uważaj z wprowadzaniem zbyt wielu poziomów generalizacji)
- Zaznacz, że klasy szczegółowe dziedziczą po klasie ogólnej, to znaczy uwzględnij uogólnienia biegnące od każdego potomka do bardziej ogólnego przodka
- Stosuj uogólnienia tylko wtedy, gdy masz do czynienia ze związkiem „jest rodzajem”; dziedziczenie wielobazowe często można zastąpić agregacją
- Wystrzegaj się wprowadzania cyklicznych uogólnień
- Utrzymuj uogólnienia w pewnej równowadze; krata dziedziczenia nie powinna być zbyt głęboka (pięć lub więcej poziomów już budzi wątpliwości) ani zbyt szeroka (lepiej wprowadzić pośrednie klasy abstrakcyjne)

Identyfikacja związków strukturalnych: powiązanie (Association) , agregacja (Aggregation)

(wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)

- Rozważ, czy w wypadku każdej pary klas jest konieczne przechodzenie od obiektów jednej z nich do obiektów drugiej
- Rozważ, czy w wypadku każdej pary klas jest konieczna inna interakcja między obiektami jednej z nich a obiektami drugiej niż tylko przekazywanie ich jako parametrów; jeśli tak, uwzględnij powiązanie między tymi klasami, w przeciwnym wypadku jest to zależność użycia. Ta metoda identyfikacji powiązań jest oparta na zachowaniu
- Dla każdego powiązania określ liczebność (szczególnie wtedy, kiedy nie jest to 1 - wartość domyślna) i nazwy ról (ponieważ ułatwiają zrozumienie modelu)
- Jeśli jedna z powiązanych klas stanowi strukturalną lub organizacyjną całość w porównaniu z klasami z drugiego końca związku, które wyglądają jak części, zaznacz przy niej specjalnym symbolem, że chodzi o agregację.
- Stosuj powiązania głównie wtedy, kiedy między obiektami zachodzą związki strukturalne

Identyfikacja wzorców projektowych (wstęp do wykładu 5)

- Dobrze zbudowany system obiektowy jest pełen wzorców obiektowych
- Wzorzec to zwyczajowo przyjęte rozwiązanie typowego problemu w danym kontekście
- Strukturę wzorca przedstawia się w postaci diagramu klas
- Zachowanie się wzorca przedstawia się za pomocą diagramu sekwencji
- Wzorce projektowe: Wzorzec reprezentuje powiązanie problemu z rozwiązaniem **(wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)**

- Każdy wzorzec składa się z trzech części, które wyrażają związek między konkretnym kontekstem, problemem i rozwiązaniem **(Christopher Aleksander)**
- Każdy wzorzec to trzyczęściowa reguła, która wyraża związek między konkretnym kontekstem, rozkładem sił powtarzającym się w tym kontekście i konfiguracją oprogramowania pozwalająca na wzajemne zrównoważenie się tych sił w celu rozwiązania zadania. **(Richar Gabriel)**
- Wzorzec to pomysł, który okazał się użyteczny w jednym rzeczywistym kontekście i prawdopodobnie będzie użyteczny w innym. **(Martin Fowler)**

Identyfikacja klas
Analiza wspólności (perspektywa koncepcji, model analizy – wykład 1)

Przykład 1 z wykładu 2 i jego kontynuacja z wykładu 3

Wykryto **trzy główne klasy** typu **„Entity”** ze względu na odpowiedzialność:

- **Rachunek** (PU: **Wstawianie nowego rachunku, Wstawianie nowego zakupu, Obliczanie wartosci rachunku**),
- **Zakup** (PU: **Wstawianie nowego zakupu, Obliczanie wartosci rachunku**),
- **ProduktBezPodatku** (PU: **Wstawianie nowego produktu, Wstawianie nowego zakupu, Obliczanie wartosci rachunku**)

Analiza zmienności (perspektywa specyfikacji, model projektowy – wykład 1)

Przykład 1 z wykładu 2 i jego kontynuacja z wykładu 3

Wykryto **dziedziczenie** w właściwościach produktów, które podają cenę jednostkową podawaną jako cenę netto, jeśli produkt nie posiada atrybutu podatek lub cenę brutto, jeśli posiada atrybut podatek. Zdefiniowano klasę pochodną:

- **ProduktZPodatkiem** typu **„Entity”,** która dziedziczy od klasy **ProduktBezPodatku** (PU: **Wstawianie nowego produktu, Obliczanie wartosci rachunku**)

Analiza zmienności (c.d)

Wykryto **strategię zmniejszania ceny jednostkowej wynikającej z promocji** powiązaną z produktem zarówno z podatkiem, jak i bez podatku:

- Zdefiniowano klasę **Promocja** typu **„Entity”**
- Zdefiniowano **związek typu asocjacja (lub słaba agregacja)** między klasami **ProduktBezPodatku i Promocja**, który jest dziedziczony przez pozostałe typy produktu tzn. **ProduktZPodatkiem**. Ponieważ jednak promocja nie musi dotyczyć każdego produktu, jest **w związku asocjacji (lub agregacji słabej) 0..1 do 0..*** z bazowym (głównym) produktem typu **ProduktBezPodatku**
- Dzięki temu produkty typu **ProduktBezPodatku** i **ProduktZPodatkiem** powinny podawać uogólnioną cenę detaliczną: **bez podatku, z podatkiem oraz w razie potrzeby z uwzględnieniem scenariusza dodawania promocji do ceny detalicznej produktu dla dwóch pierwszych przypadków** (stąd cztery typy ceny detalicznej)
- Podstawą identyfikacji są PU: **Wstawianie nowego produktu, Wstawianie nowego zakupu, Obliczanie wartosci rachunku.**

Analiza zmienności (c.d)

Wykryto związki:

- **silnej agregacji** między obiektem typu **Rachunek** i obiektami typu **Zakup** (rachunek posiada kolekcję zakupów)
- oraz **słabej agregacji** między obiektem typu **Zakup** a obiektem typu **ProduktBezPodatku** (zakup składa się z produktu bazowego lub jego następców)
- Podstawą identyfikacji są PU: **Wstawianie nowego zakupu, Obliczanie wartosci rachunku.**

Zastosowano:

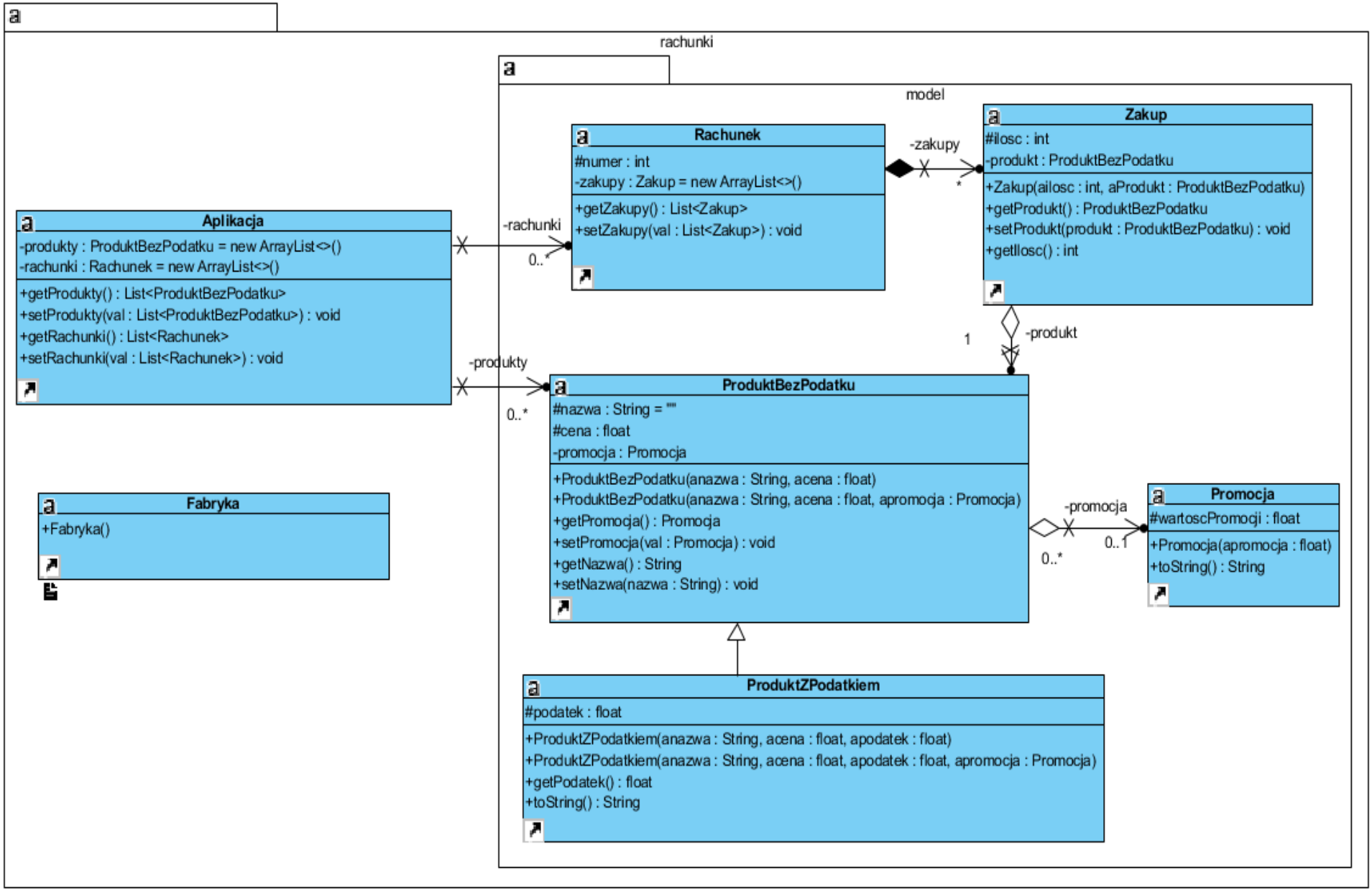
- klasę fasadową **Aplikacja** typu **„Control”** do oddzielenia przetwarzania obiektów typu „**Entity**” od pozostałej części systemu

Analiza zmienności (c.d)

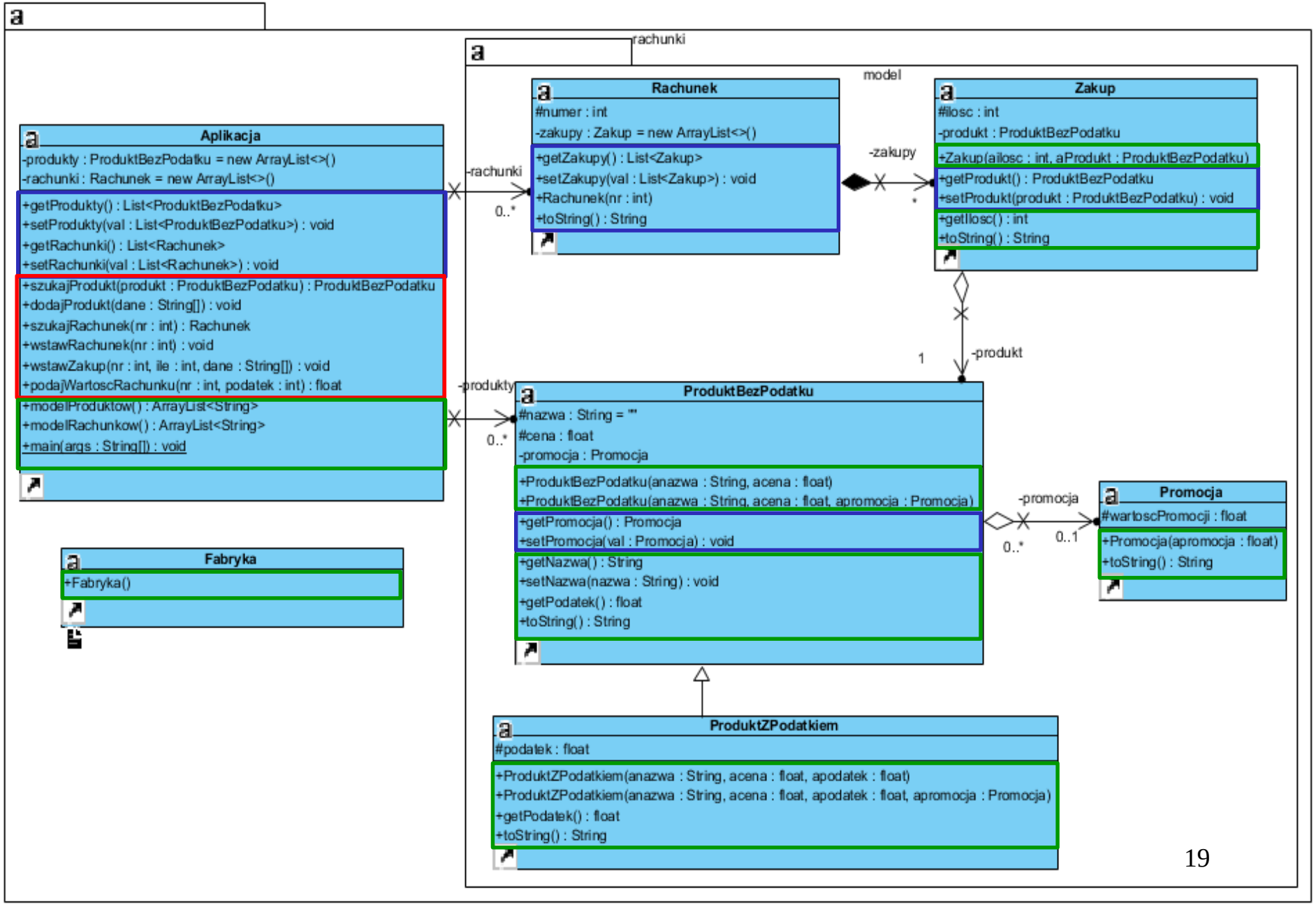
Zastosowano:

- klasę fasadową **Aplikacja** typu **„Control”** do oddzielenia przetwarzania obiektów typu „**Entity**” od pozostałej części systemu
- klasę typu **„Control”** jako fabrykę obiektów (**Fabryka**) do tworzenia różnych typów produktów – czyli obiektów typu **ProduktBezPodatku** i **ProduktZPodatkiem**.

Analiza wspólności i zmienności - rezultat



Projekt powiązań | Metody przypadków użycia | Decyzja projektowa



```
package rachunki;
import java.util.ArrayList;
import java.util.List;
import rachunki.model.*;
public class Aplikacja
{   private List<ProduktBezPodatku> produkty = new ArrayList<>();
    private List<Rachunek> rachunki = new ArrayList<>();
    List<ProduktBezPodatku> getProdukty ()                    { return null;  }
    void setProdukty (ArrayList<ProduktBezPodatku> val)       {   }
    List<Rachunek> getRachunki ()                             { return null;  }
    public void setRachunki (ArrayList<Rachunek> val)         {   }
    public void wstawZakup (int nr, int ile, String dane[])   {   }
    public Rachunek szukajRachunek (int nr)                   { return null;  }
    public void wstawRachunek (int nr)                        {   }
    public float podajWartoscRachunku (int nr, int podatek_)  { return 0.0f;  }
    public Produkt1 szukajProdukt (ProduktBezPodatku produkt) { return null;  }
    public void dodajProdukt (String[] dane)                  {   }
     public ArrayList<String> modelProduktow ()               {return null;  }
     public ArrayList<String> modelRachunkow ()               {return null;  }
     public static void main (String[] args)                  {   }
}
```

Diagramy klas, diagramy sekwencji

1. **Identyfikacja elementów diagramów klas**

   [Shalloway A.,Trott James R.,Projektowanie zorientowane obiektowo. Wzorce projektowe. Gliwice, Helion, 2005]

2. **Diagramy sekwencji UML**

   http://sparxsystems.com.au/resources/uml2_tutorial/

Diagramy UML 2 – część czwarta

Na podstawie

UML 2.0 Tutorial

http://sparxsystems.com.au/resources/uml2_tutorial/

Dwa rodzaje diagramów UML 2

Diagramy UML modelowania strukturalnego

- **Diagramy pakietów**
- ***Diagramy klas***
- **Diagramy obiektów**
- **Diagramy mieszane**
- **Diagramy komponentów**
- **Diagramy wdrożenia**

Diagramy UML modelowania zachowania

- ***Diagramy przypadków użycia***
- ***Diagramy aktywności***
- **Diagramy stanów**
- **Diagramy komunikacji**
- ***Diagramy sekwencji***
- **Diagramy czasu**
- **Diagramy interakcji**

Diagramy sekwencji (Sequence Diagrams)

- wyrażają **interakcje w czasie** (wiadomości wymieniane między obiektami jako poziome strzałki wychodzące od linii życia jednego obiektu i wchodzące do linii życia drugiego obiektu)

- wyrażają dobrze **komunikację** między obiektami i zarządzanie przesyłaniem wiadomości

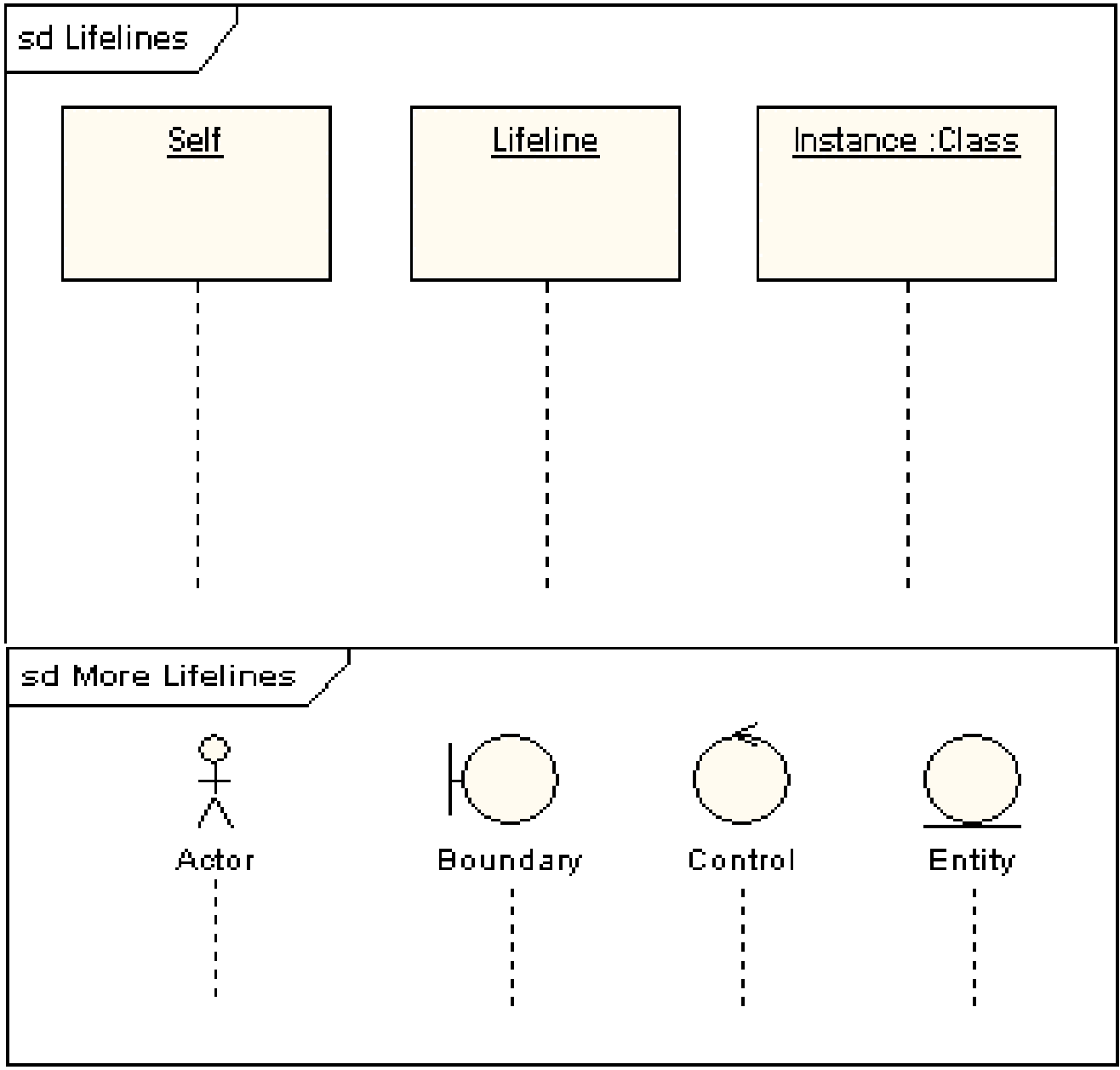
- **nie są używane do wyrażania złożonej logiki proceduralnej**

- **są używane do modelowanie scenariusza przypadku użycia**

Linie życia (Lifelines)

Linie życia reprezentują indywidualne uczestniczenie obiektu w diagramie. Posiadają one często prostokąty zawierające nazwę i typ obiektu.

Czasem diagram sekwencji zawiera **linię życia aktora**. Oznacza to, że właścicielem diagramu sekwencji jest **przypadek użycia**. Elementy oznaczające ***obiekty typu „boundary”, „control”*** **,** ***„Entity”*** mają również swoje linie życia.

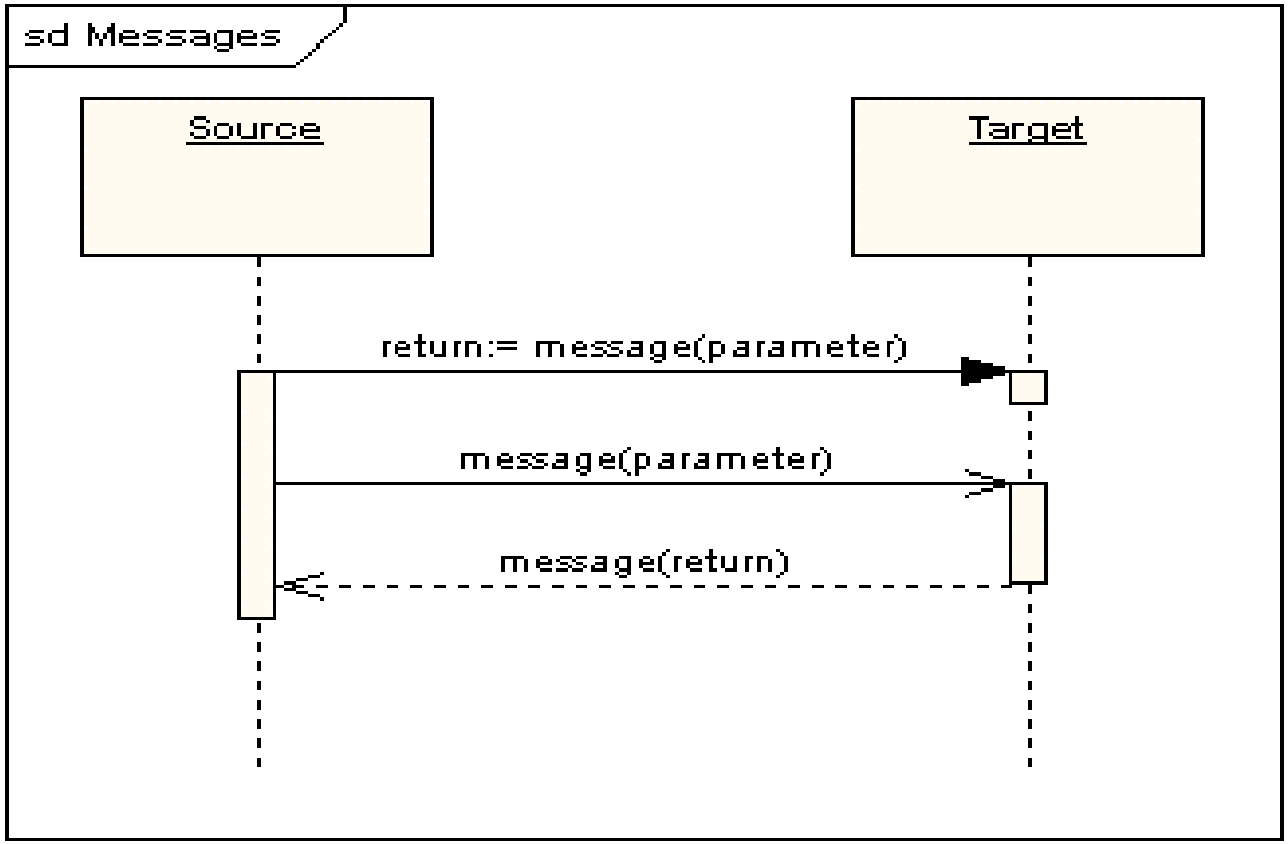


Wiadomości (Messages)

- są wyświetlane jako strzałki.
- mogą być ***kompletne, zgubione i znalezione;***
- mogą być ***synchroniczne i asychroniczne***
- Mogą być typu wywołanie operacji **(*call)*** lub sygnał (***signal***)
- **dla wywołań operacji (call) wyjście strzałki z linii życia oznacza, że obiekt ten wywołuje metodę obiektu, do którego strzałka dochodzi**

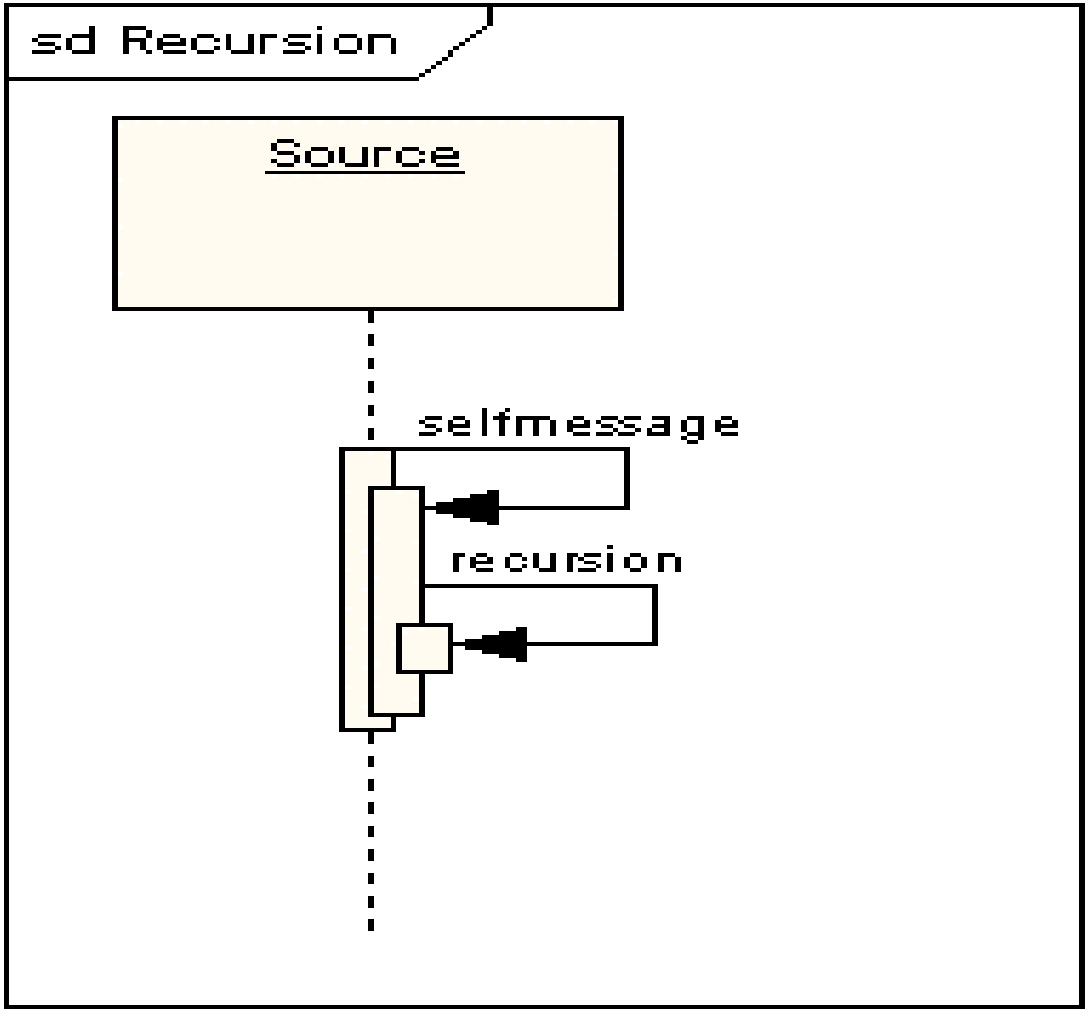
Wykonywanie interakcji (Execution Occurrence)

1. pierwsza wiadomość jest synchroniczna, kompletna i posiada return (wywołanie metody obiektu Target przez obiekt przez Source),
2. druga wiadomość jest asynchroniczna (wywołanie metody obiektu Target przez obiekt przez Source),
3. trzecia wiadomość jest asynchroniczną wiadomością typu return (przerywana linia – return metody asynchronicznej obiektu Target).



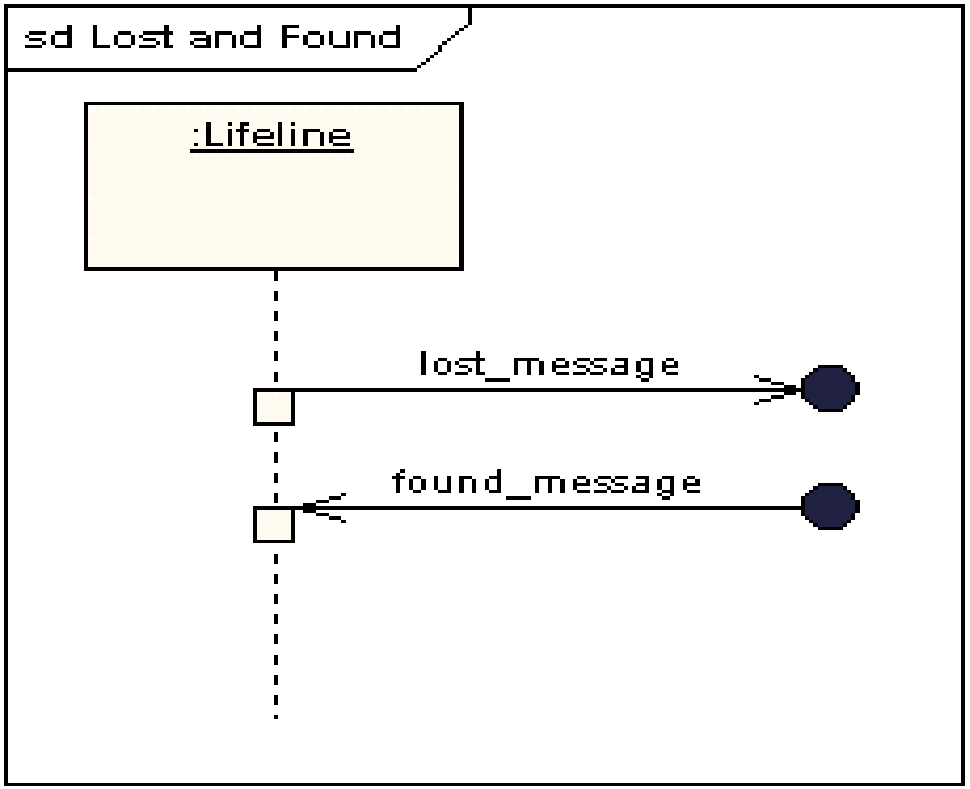
Własne wiadomości (Self Message)

Własne wiadomości reprezentują rekursywne wywoływanie operacji albo jedna operacja wywołuje inną operację należącą do tego samego obiektu.



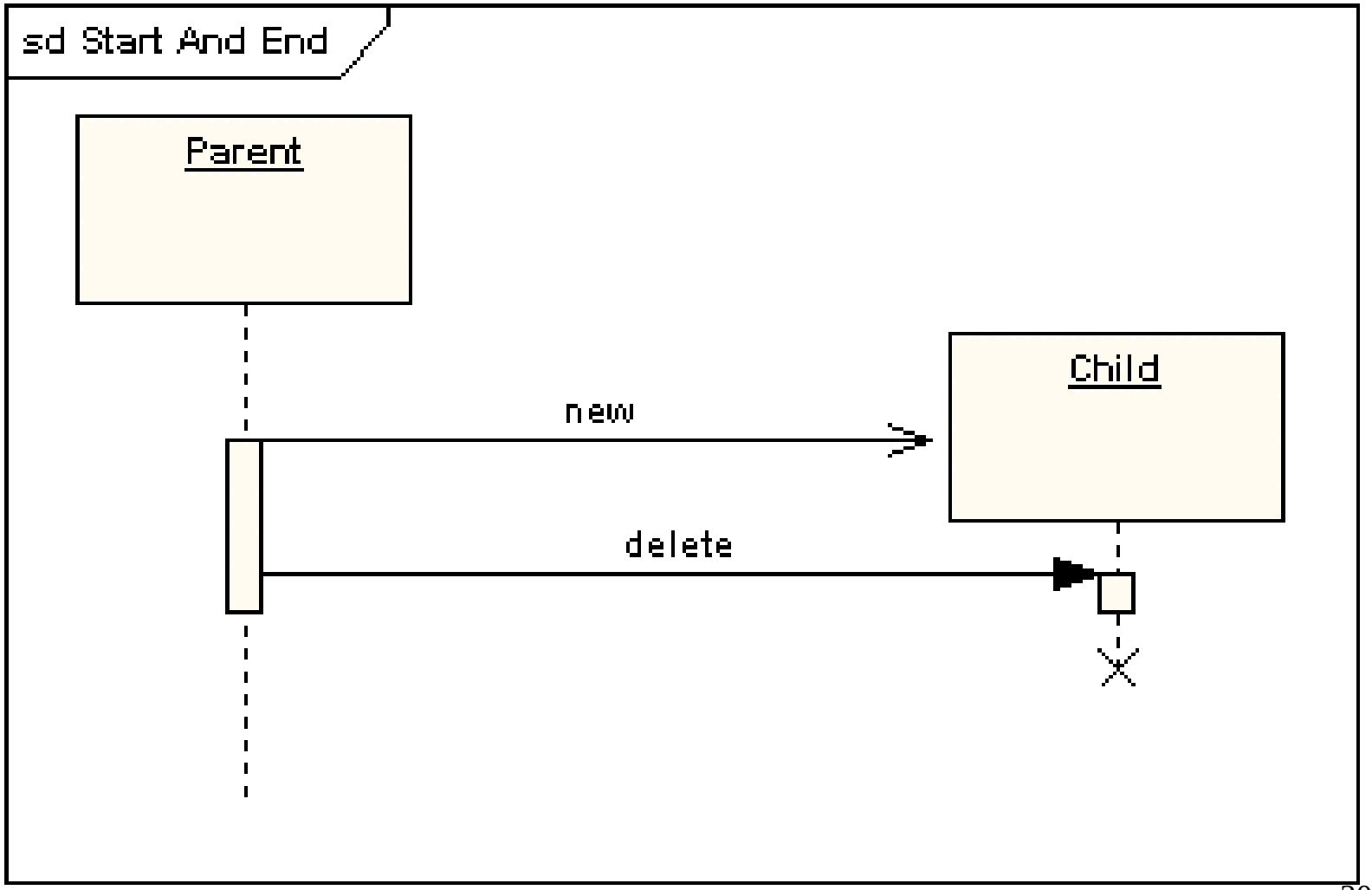
Zgubione i znalezione wiadomości (Lost and Found Messages)

- **Zgubione wiadomości** są wysłane i nie docierają do obiektu docelowego lub nie są pokazane na bieżącym diagramie.
- **Znalezione wiadomości** docierają od nieznanego nadawcy albo od nadawcy, który nie jest pokazany na bieżącym diagramie.



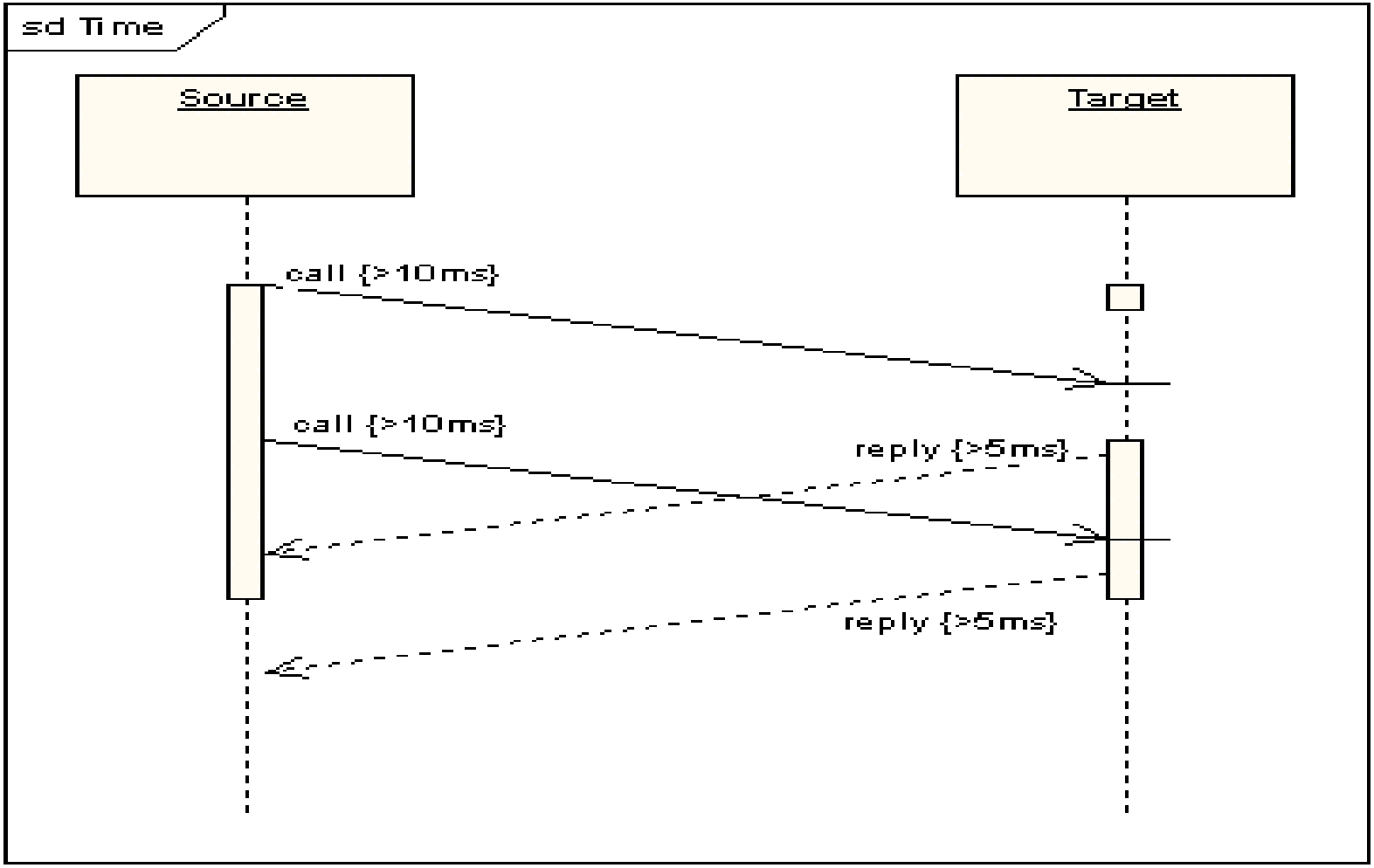
Start linii życia i jej koniec (Lifeline Start and End)

Oznacza to tworzenie (typu **Create Message**) i usuwanie obiektu (symbol **X**)



Ograniczenia czasowe (Duration and Time Constraints)

Domyślnie, wiadomość jest poziomą linią. W przypadku, gdy należy ukazać opóźnienia czasu wynikające z czasu podjętych akcji przez obiekt po otrzymaniu wiadomości, wprowadza się ukośne linie wiadomości.

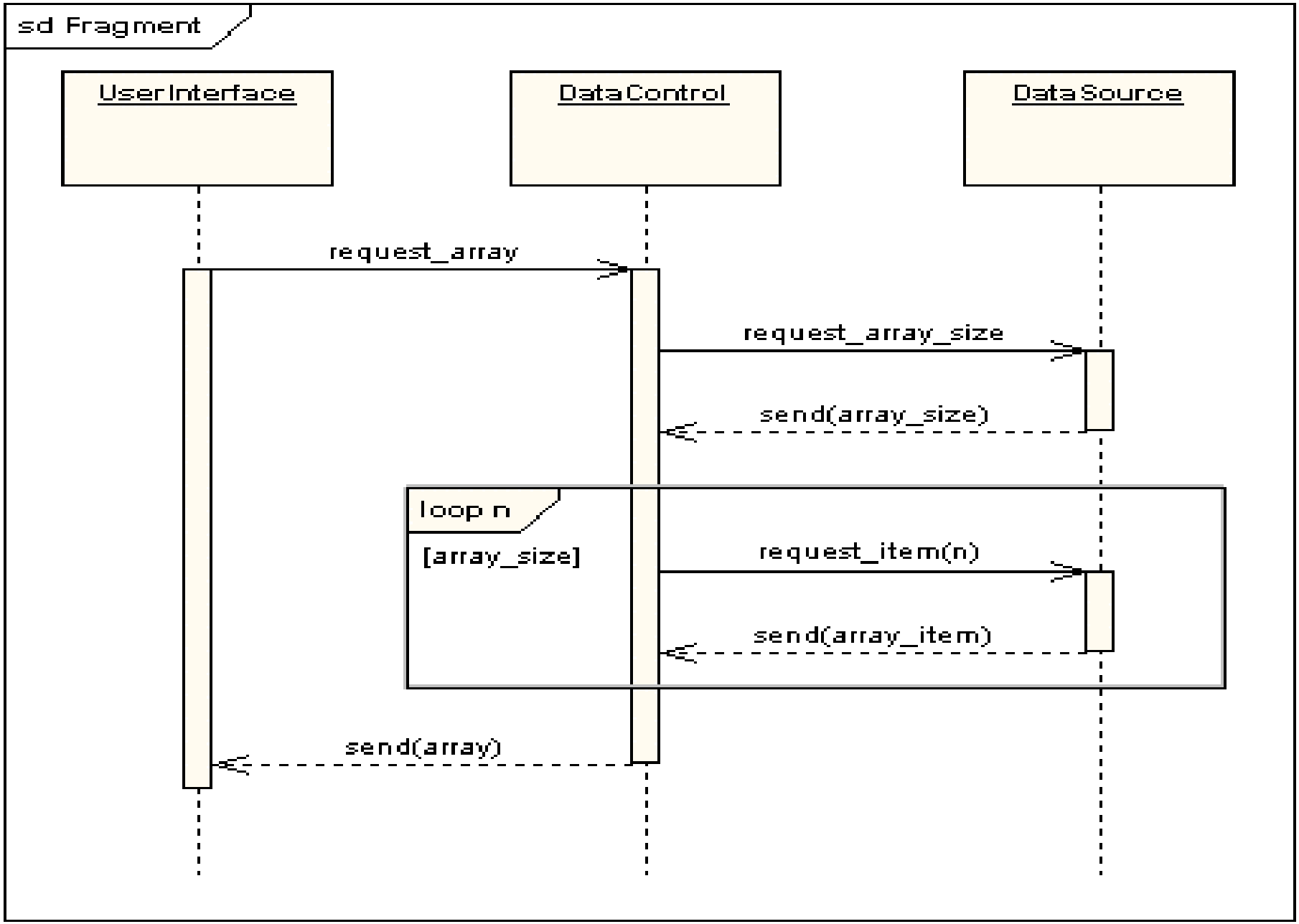


Złożone modelowanie sekwencji wiadomości

Fragmenty ujęte w ramki umożliwiają:

1. **fragmenty alternatywne** (oznaczone "**opt**") modelują konstrukcje **if…then…else**
2. **fragmenty opcjonalne** (oznaczone "**alt**") modelują konstrukcje **switch**.
3. **fragment Break** modeluje alternatywną sekwencję zdarzeń dla pozostałej części diagramu.
4. **fragment równoległy** (oznaczony "**par**") modeluje proces równoległy.
5. **słaba sekwencja** (oznaczona "**seq**") zamyka pewna liczbę sekwencji, w której wszystkie wiadomości muszą być wykonane przed rozpoczęciem innych wiadomości z innych fragmentów, z wyjątkiem tych wiadomości, **które nie dzielą linii życia oznaczonego fragmentu**.
6. **dokładna sekwencja** (oznaczona jako "**strict**") zamyka wiadomości, które muszą być wykonane w określonej kolejności
7. **fragment negatywny** (oznaczony "**neg**") zamyka pewną liczbę niewłaściwych wiadomości
8. **fragment krytyczny** (oznaczony jako „**critical**") zamyka sekcję krytyczną.
9. **fragment ignorowany** (oznaczony jako "**ignored**") deklaruje wiadomość/ci nieistotne
10. **fragment rozważany-** tylko ważne są wiadomości w tym fragmencie
11. **fragment asercji** (oznaczony "**assert**") eliminuje wszystkie sekwencje wiadomości, które są objęte danym operatorem, jeśli jego wynik jest fałszywy
12. **pętla** (oznaczony "**loop**") oznacza powtarzanie interakcji we fragmencie.

Pętla Wykonanie w pętli fragmentu diagramu sekwencji

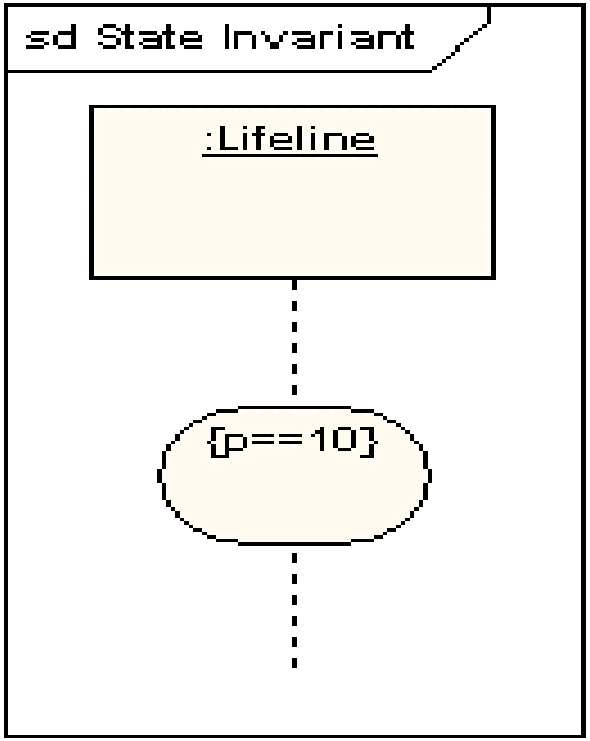


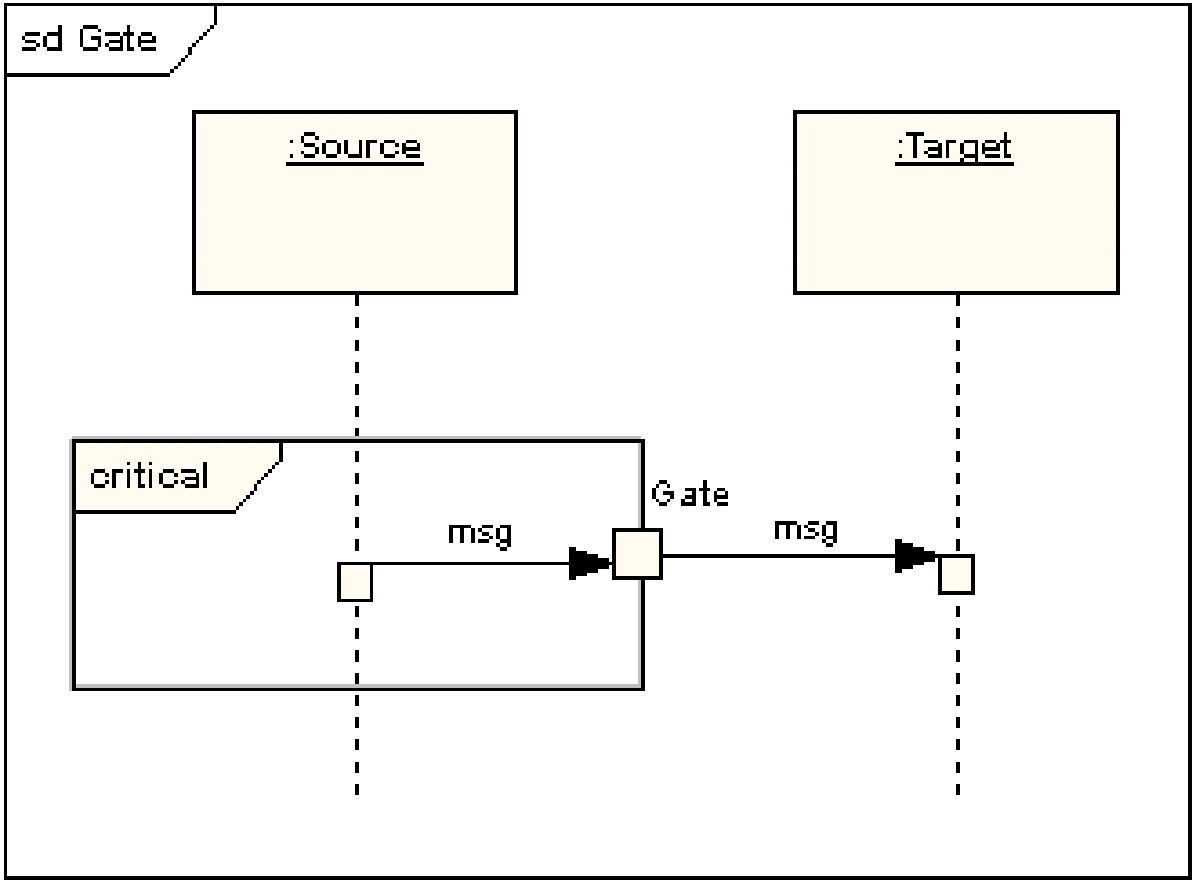
Stan niezmienny lub ciągły (State Invariant /Continuations)

- **Stan niezmienny** jest oznaczany symbolem prostokąta z zaokrąglonymi wierzchołkami.
- **Stany ciągłe** są oznaczone takim samym symbolem, obejmującym kilka linii życia

Brama (Gate)

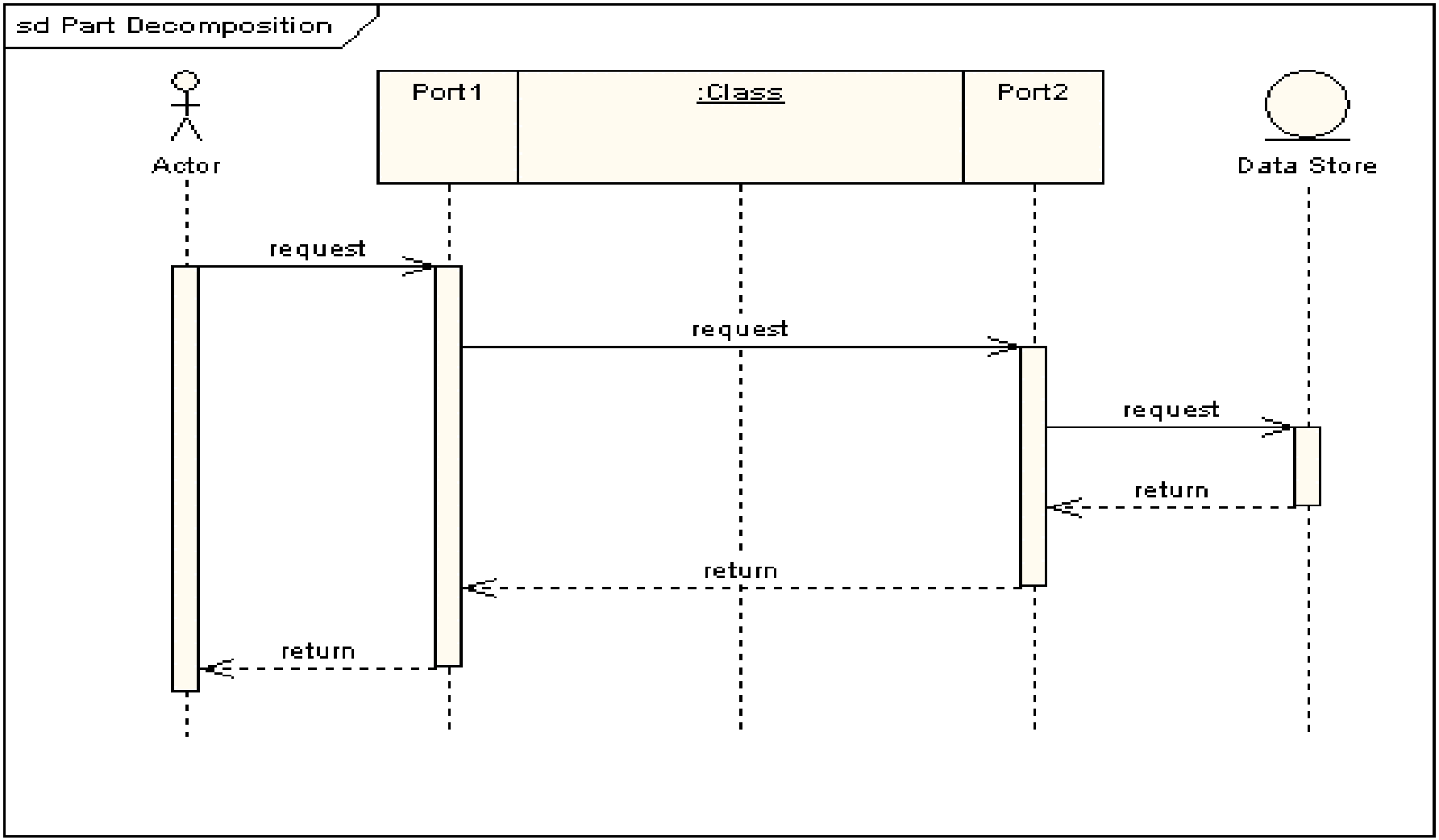
Oznacza przekazywanie wiadomości na zewnątrz między fragmentem i pozostałą częścią diagramu (linie życia, inne fragmenty)





Dekompozycja (Part Decomposition)

Obiekt ma więcej niż jedną linę życia (np. typu Class). Pozwala to pokazać zagnieżdżone protokoły przekazywanych wiadomości np. wewnątrz obiektu i na zewnętrz (w przykładzie typu Class)



Diagramy klas, diagramy sekwencji

1. **Identyfikacja elementów diagramów klas**

   [Shalloway A.,Trott James R.,Projektowanie zorientowane obiektowo. Wzorce projektowe. Gliwice, Helion, 2005]

2. **Diagramy sekwencji UML**

   http://sparxsystems.com.au/resources/uml2_tutorial/

3. **Przykłady diagramów sekwencji – kontynuacja przykładu 1 z wykładów: 2 i 3**

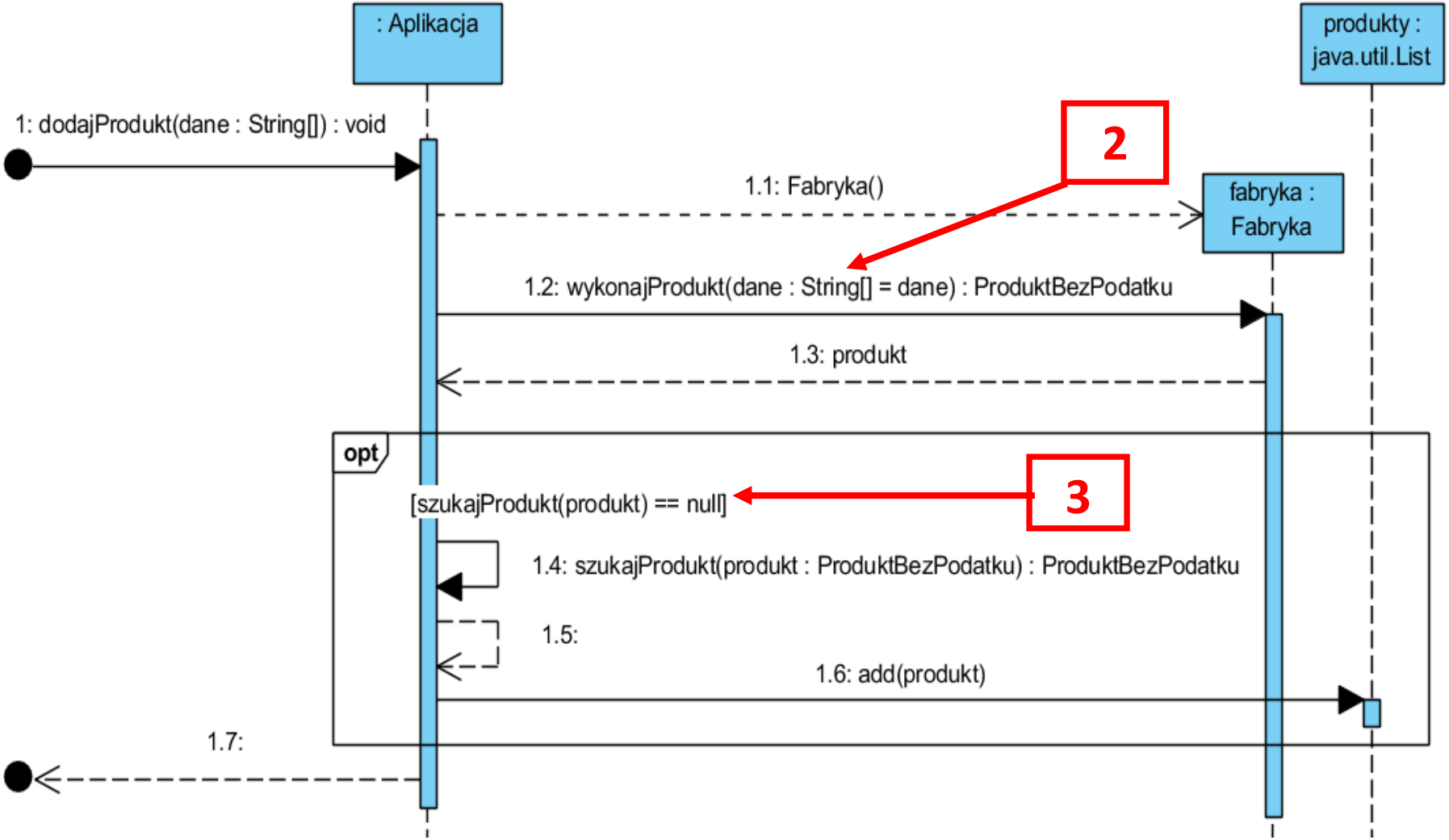
Iteracja 1

Projekt przypadku użycia

„ **Wstawianie nowego produktu**”

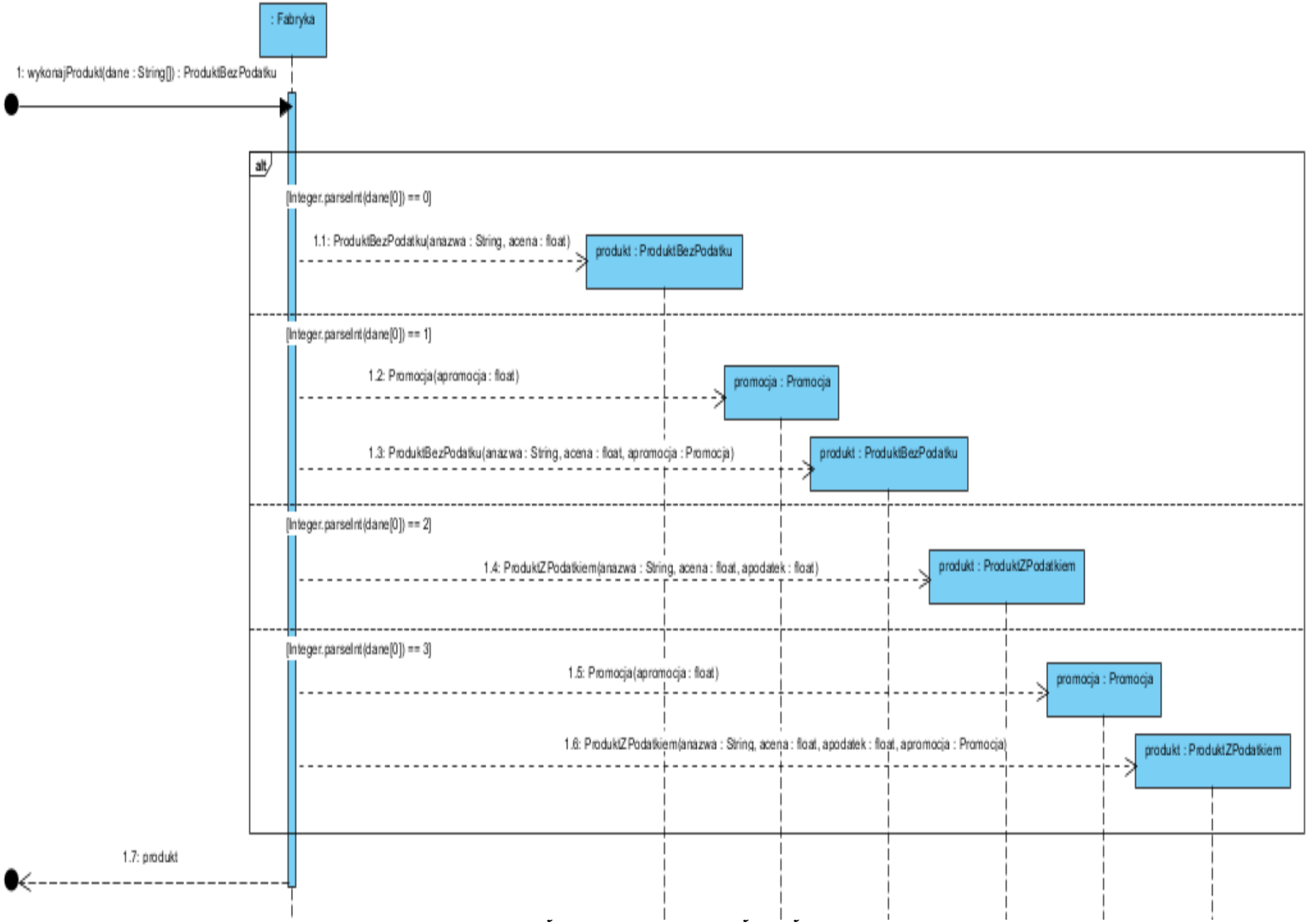
za pomocą diagramu sekwencji i diagramu klas. Diagram klas jest uzupełniany metodami zidentyfikowanymi podczas projektowania scenariusza przypadku użycia za pomocą diagramu sekwencji.

PU Wstawianie nowego produktu
(1) void dodajProdukt(String [] dane)



```
                                                    //class Aplikacja

 private List <ProduktBezPodatku> produkty = new ArrayList <>();

public void dodajProdukt (String dane[])
  {
     Fabryka fabryka = new Fabryka();
     ProduktBezPodatku produkt = fabryka.wykonajProdukt(dane);
      if (szukajProdukt(produkt) == null)
        produkty.add(produkt);
  }
```

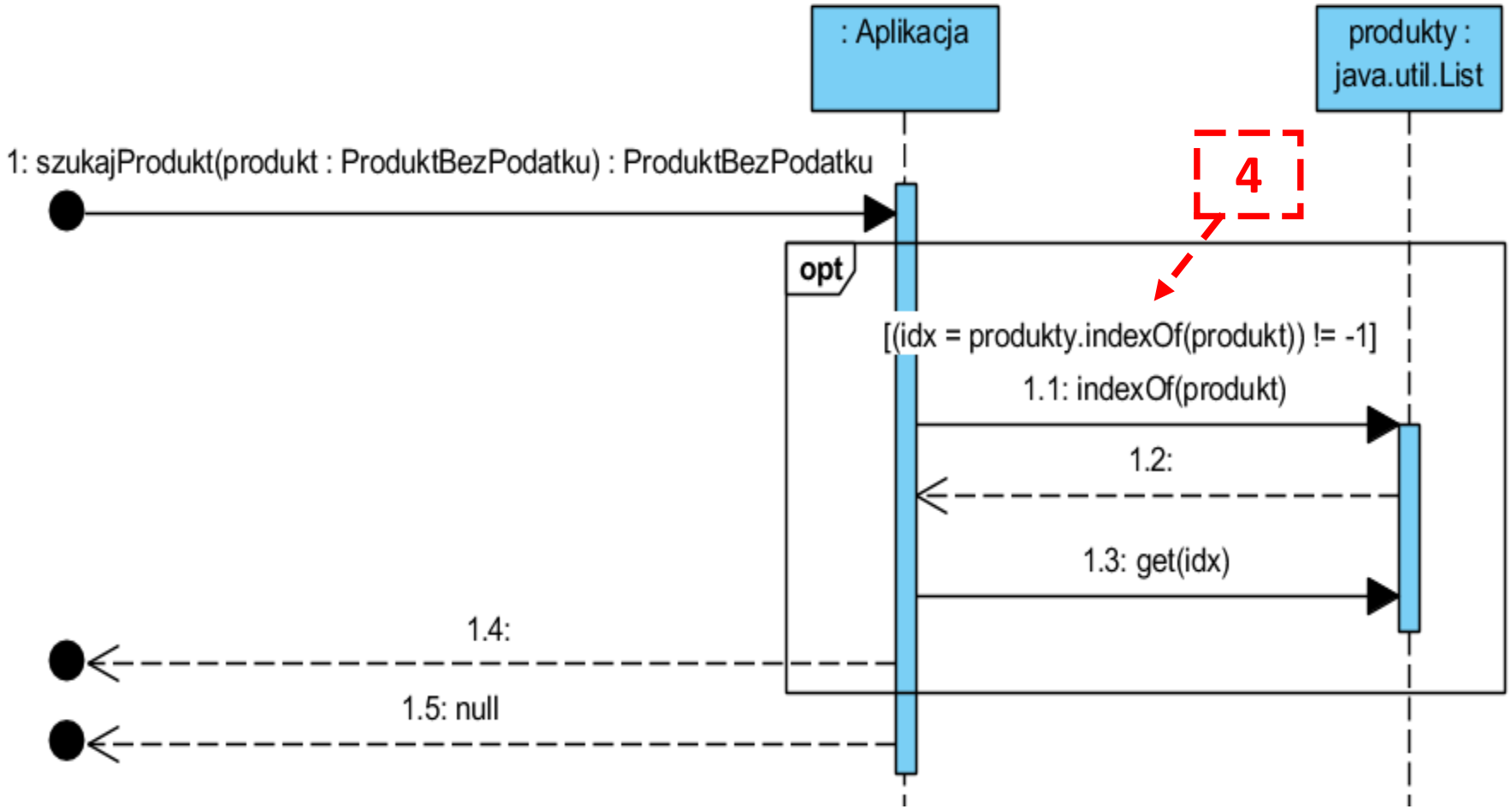
(2) ProduktBez Podatku wykonajProdukt(String dane[])



```
public class Fabryka                    //Fabryka -decyzje na poziomie tworzenia kodu
 { public Fabryka() { }
   public ProduktBezPodatku wykonajProdukt(String dane[])
   { ProduktBezPodatku produkt = null;
     Promocja promocja;
     switch ( Integer.parseInt(dane[0]) )
    { case 0: produkt= new ProduktBezPodatku(dane[1], Float.parseFloat(dane[2]));
              break;
      case 1: promocja = new Promocja(Float.parseFloat(dane[3]));
               produkt = new ProduktBezPodatku (dane[1],
                                    Float.parseFloat(dane[2]),promocja);
              break;
      case 2: produkt = new ProduktZPodatkiem (dane[1], Float.parseFloat(dane[2]),
                                     Float.parseFloat(dane[3]));
               break;
      case 3: promocja = new Promocja(Float.parseFloat(dane[4]));
               produkt= new ProduktZPodatkiem(dane[1], Float.parseFloat(dane[2]),
                                    Float.parseFloat(dane[3]),promocja);
              break;
     }
    return produkt; }
}
```

PU Szukanie produktu

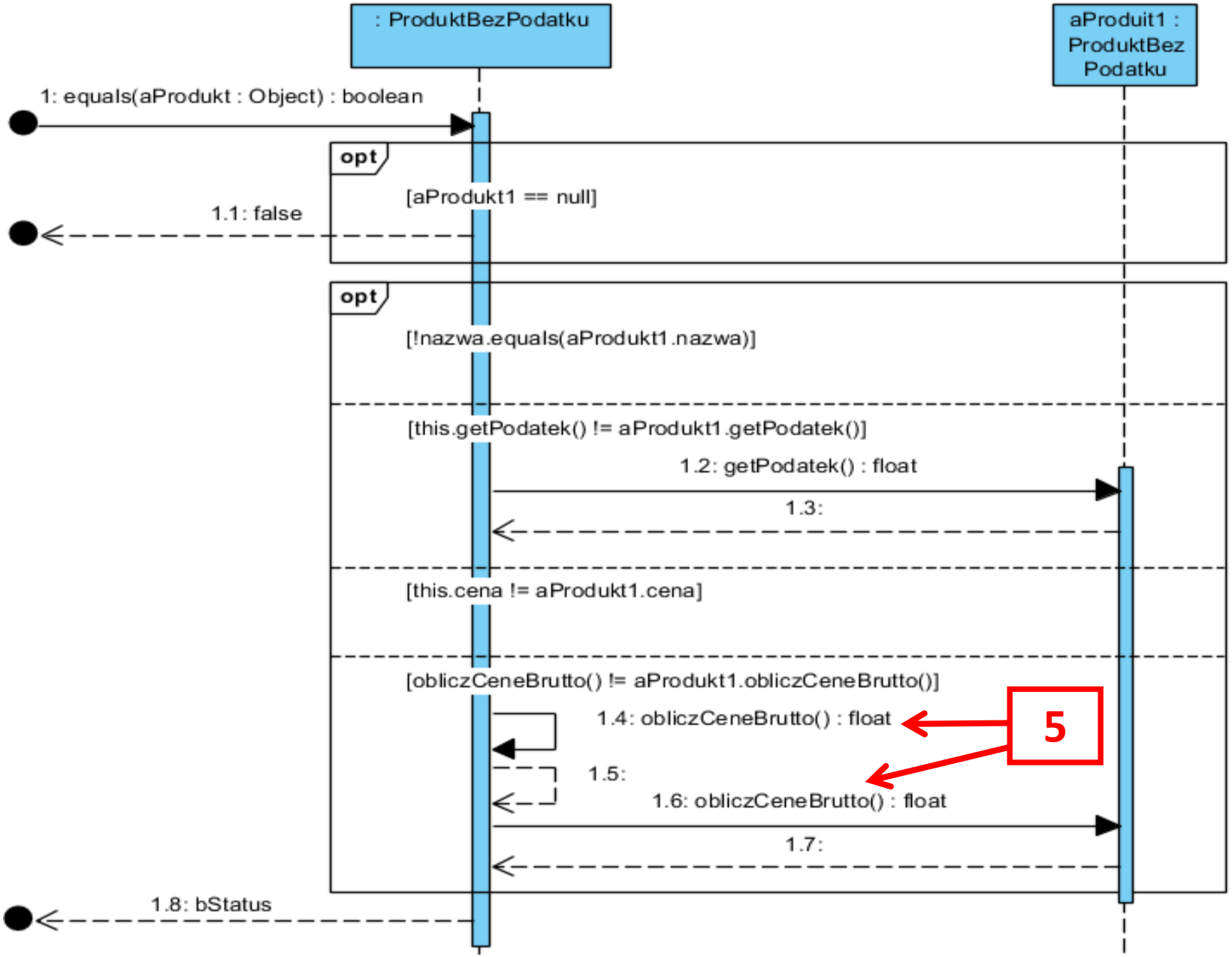
(3) ProduktBezPodatku szukajProdukt(ProduktBezPodatku produkt)



```
                                                    //Aplikacja
private List <ProduktBezPodatku> produkty = new ArrayList <>();

ProduktBezPodatku szukajProdukt (ProduktBezPodatku produkt)
  {
    int idx;
    if ((idx=produkty.indexOf(produkt))!=-1 )
     {
       return  produkty.get(idx);
     }
    return null;
  }
```

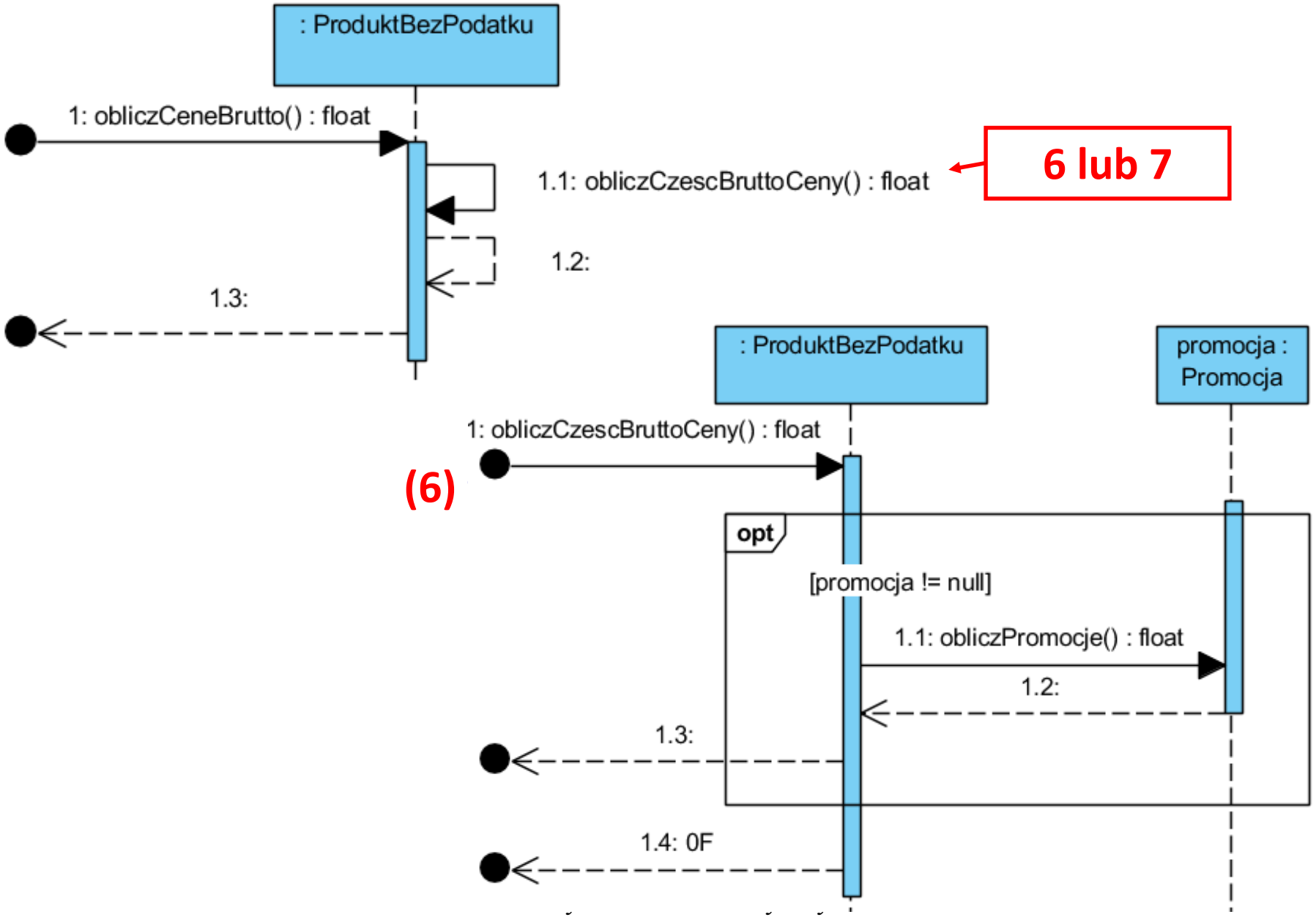
```
public int indexOf(Object o) {
      if (o == null) {
        for (int i = 0; i < size; i++)
          if (elementData[i]==null)
            return i;
      } else {
        for (int i = 0; i < size; i++)
          if (o.equals(elementData[i]))
            return i;  }
      return -1;  }
```

(4) boolean equals(Object aProdukt)

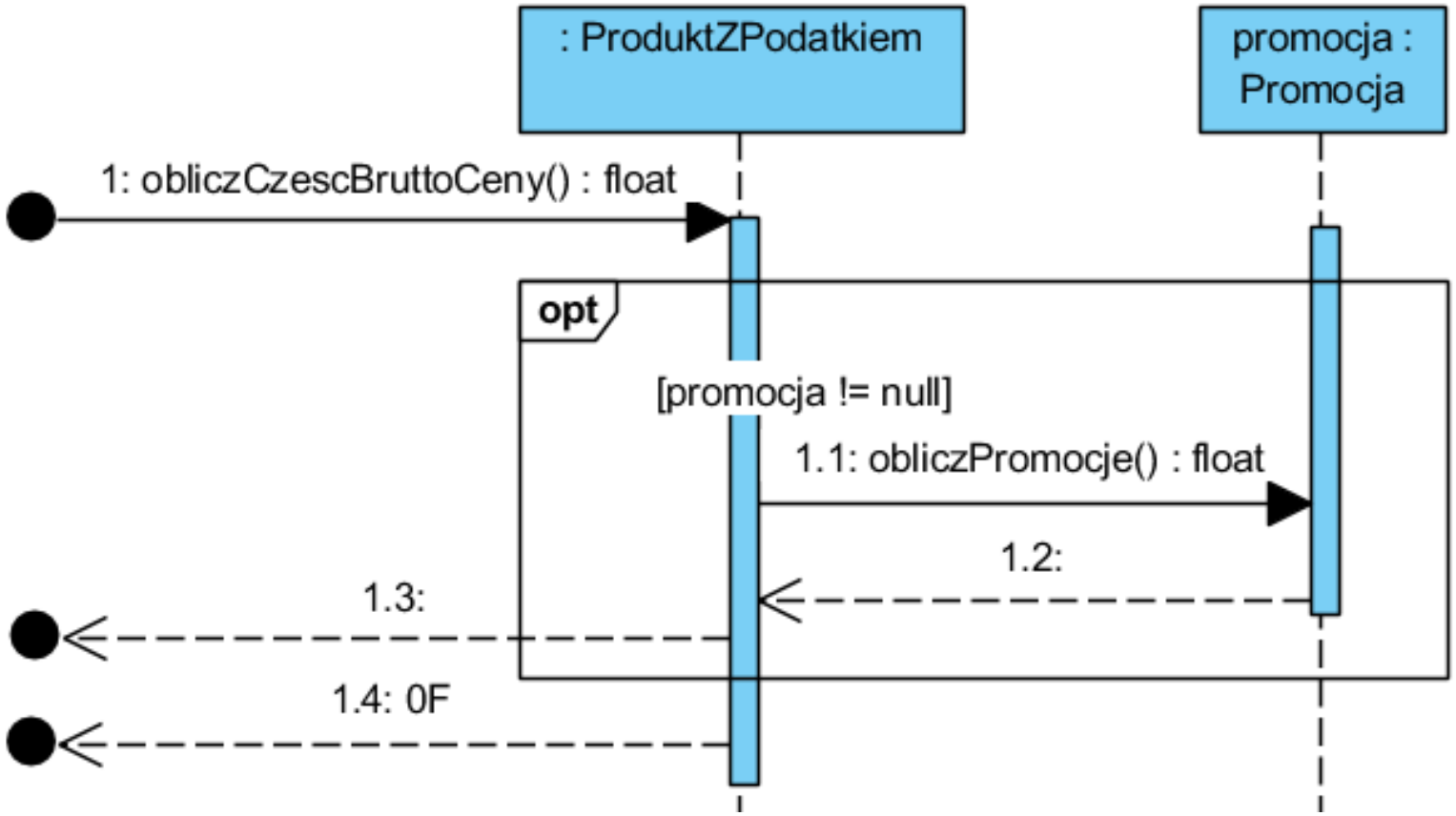


```
//class ProduktBezPodatku
public boolean equals (Object aProdukt)
 {
    ProduktBezPodatku aProdukt1=(ProduktBezPodatku)aProdukt;
    if ( aProdukt1 == null ) return false;
    boolean bStatus = true;
   if ( !nazwa.equals(aProdukt1.nazwa)) bStatus = false;
   else
    if (this.getPodatek()!=aProdukt1.getPodatek())
       bStatus = false;
    else
      if (this.cena!=aProdukt1.cena)
        bStatus = false;
      else
        if (this.obliczCeneBrutto() != aProdukt1.obliczCeneBrutto())
          bStatus = false;
    return bStatus;
 }
```

(5) float obliczCeneBrutto()



(7) float obliczCzescBruttoCeny()

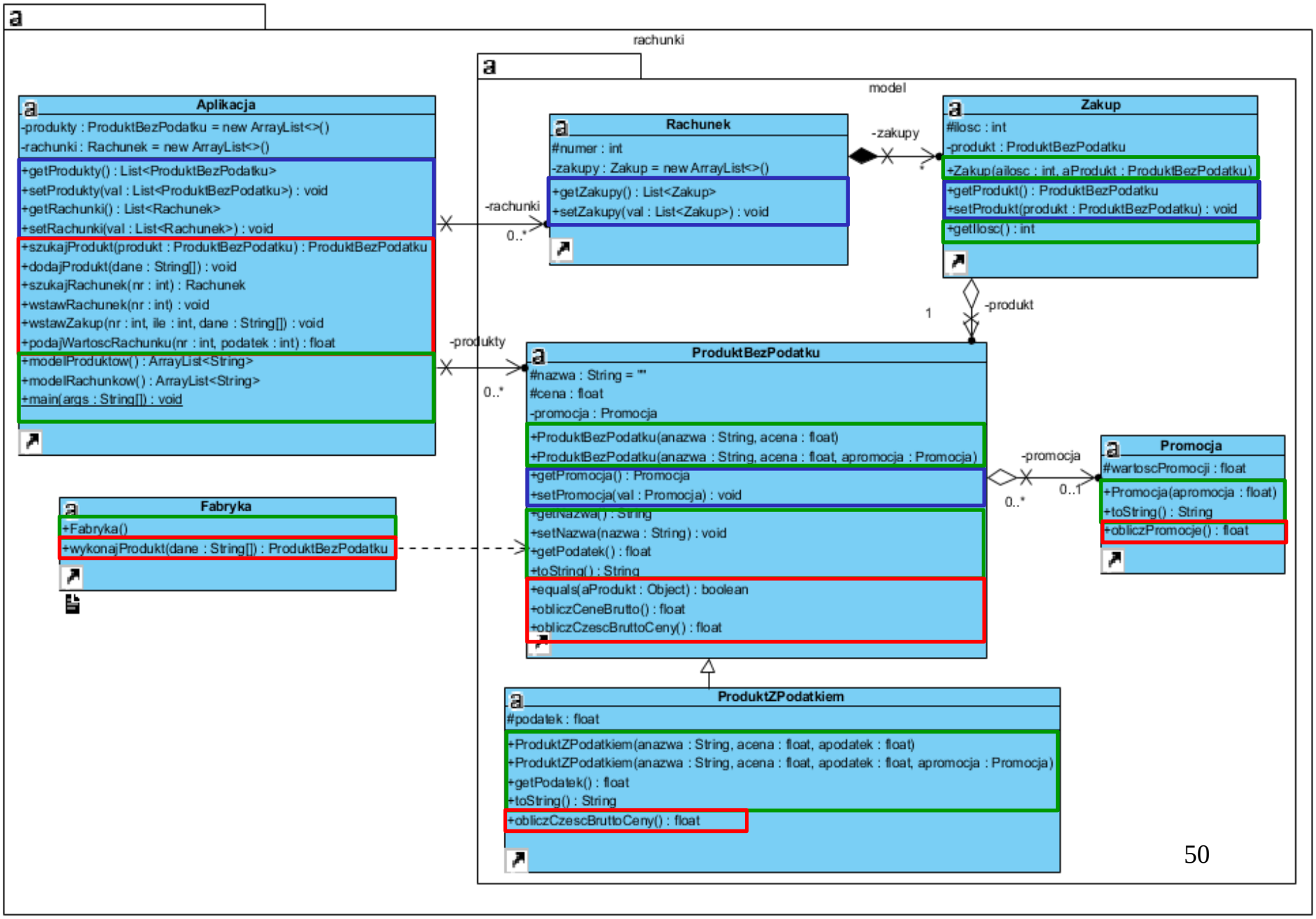


```
//class ProduktBezPodatku
public float obliczCeneBrutto ()
  {
      return cena + obliczCzescBruttoCeny();
  }

 public float getPodatek ()
   {
     return -1;
   }

public float obliczCzescBruttoCeny()
{
   if (promocja != null)
       return cena * (-promocja.obliczPromocje()/100);
   return 0F;
 }
```

```
@Override
public  float obliczCzescBruttoCeny ()     //class ProduktZPodatkiem
   { float dodatek = 0;
     if (promocja  != null)
        dodatek= cena*(-promocja.obliczPromocje()/100);
      return  cena*podatek/100 + dodatek;
   }

 @Override
public  float getPodatek ()
   {    return podatek;    }
```

```
                                          //class Promocja lub dowolny jej następca
public  float obliczPromocje ()
   {  if (wartoscPromocji<50)                    //jakiś algorytm obliczania promocji
          return wartoscPromocji;
      return wartoscPromocji *1.1F;
   }
```

Projekt powiązań | Metody przypadków użycia | Decyzja projektowa



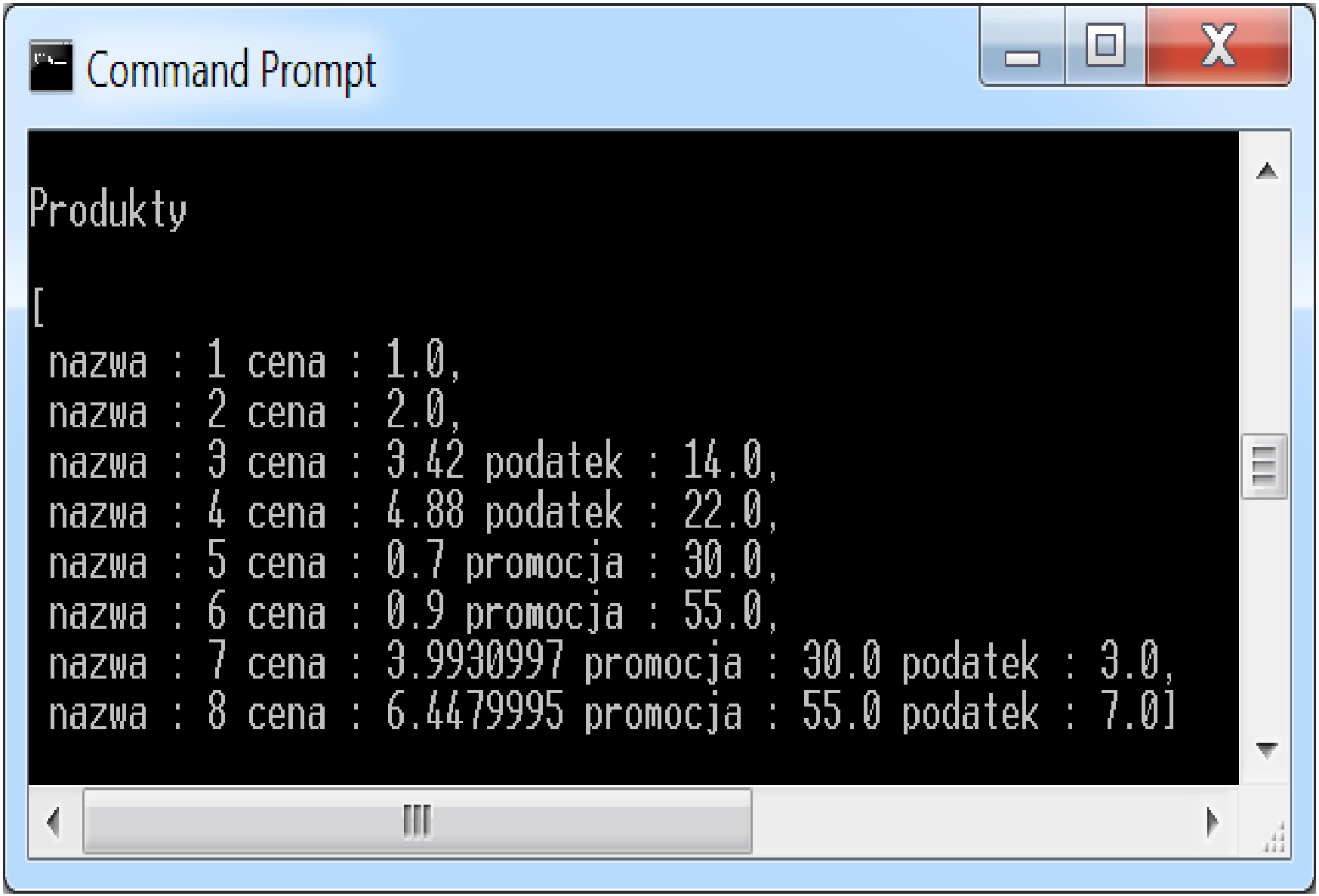
```java
public ArrayList<String> modelProduktow()        //Aplikacja
 {     ArrayList<String> modelProduktow = new ArrayList();
       for (ProduktBezPodatku produkt : produkty)
          modelProduktow.add("\n" + produkt.toString());
       return modelProduktow;
   }
```

```java
 @Override          //ProduktBezPodatku
public String toString()
 {  StringBuilder sb = new StringBuilder ();
    sb.append(" nazwa : ");
    sb.append(nazwa);
    sb.append(" cena : ");
    sb.append(obliczCeneBrutto());
    if (promocja != null)
       sb.append(promocja.toString());
   return sb.toString();
 }
```

```java
 @Override          //ProduktZPodatkiem
public String toString()
 { StringBuilder sb = new StringBuilder ();
   sb.append(super.toString());
   sb.append (" podatek : " );
   sb.append ( podatek );
   return sb.toString ();
 }
```

```java
 @Override          //Promocja
public String toString()
 {    StringBuilder sb = new StringBuilder();
      sb.append(" promocja : ");
      sb.append(obliczPromocje());
      return sb.toString();
}
```

```
public static void main(String args[])                                    //Aplikacja
{ Aplikacja app=new Aplikacja();
  String dane1[]={"0","1","1"};              String dane2[]={"0","2","2"};
  app.dodajProdukt(dane1);
  app. dodajProdukt(dane2);
  app. dodajProdukt(dane1);
  String dane3[]={"2","3","3","14"};         String dane4[]={"2","4","4","22"};
  app. dodajProdukt(dane3);
  app.dodajProdukt(dane4);
  app.dodajProdukt(dane3);
  String dane5[]={"1","5","1","30"};         String dane6[]={"1","6","2","50"};
  String dane7[]={"3","7","5.47","3","30"};
  String dane8[]={"3","8", "12.46","7","50"};
  app.dodajProdukt(dane5);
  app.dodajProdukt(dane6);
  app.dodajProdukt(dane5);
  app.dodajProdukt(dane7);
  app.dodajProdukt(dane8);
  app.dodajProdukt(dane7);
  System.out.println("\nProdukty\n");
  System.out.println(app.modelProduktow());}
```


Command Prompt
Produkty
[
nazwa : 1 cena : 1.0,
nazwa : 2 cena : 2.0,
nazwa : 3 cena : 3.42 podatek : 14.0,
nazwa : 4 cena : 4.88 podatek : 22.0,
nazwa : 5 cena : 0.7 promocja : 30.0,
nazwa : 6 cena : 0.9 promocja : 55.0,
nazwa : 7 cena : 3.9930997 promocja : 30.0 podatek : 3.0,
nazwa : 8 cena : 6.4479995 promocja : 55.0 podatek : 7.0]

Dodatek – kolejne iteracje rozwoju tworzenia oprogramowania
(kontynuacja przykładu 1 z wykładu 2 i wykładu 3)

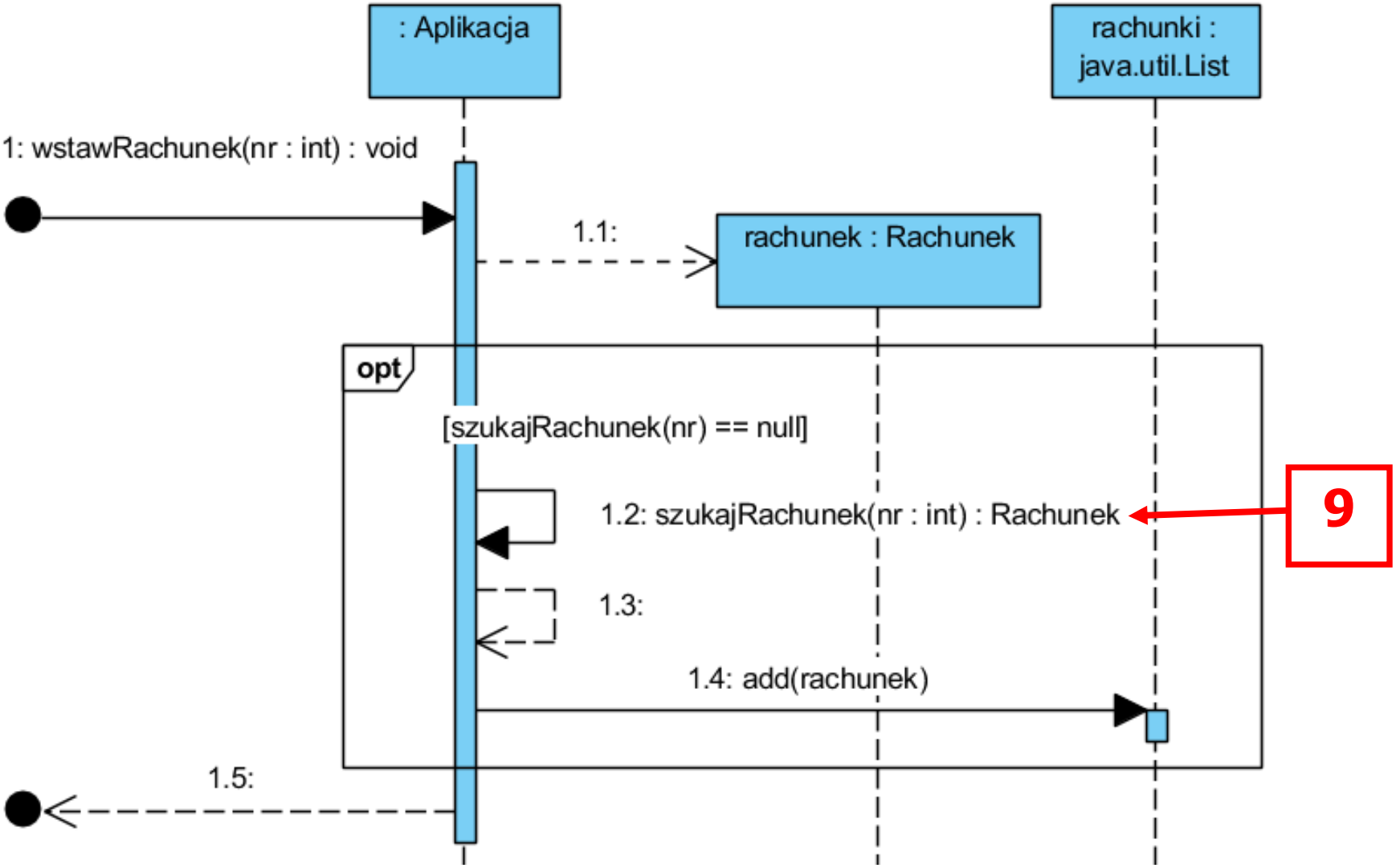
Iteracja 2

Projekt przypadku użycia

„ **Wstawianie nowego rachunku**”

za pomocą diagramu sekwencji i diagramu klas. Diagram klas jest uzupełniany metodami zidentyfikowanymi podczas projektowania scenariusza przypadku użycia za pomocą diagramu sekwencji.

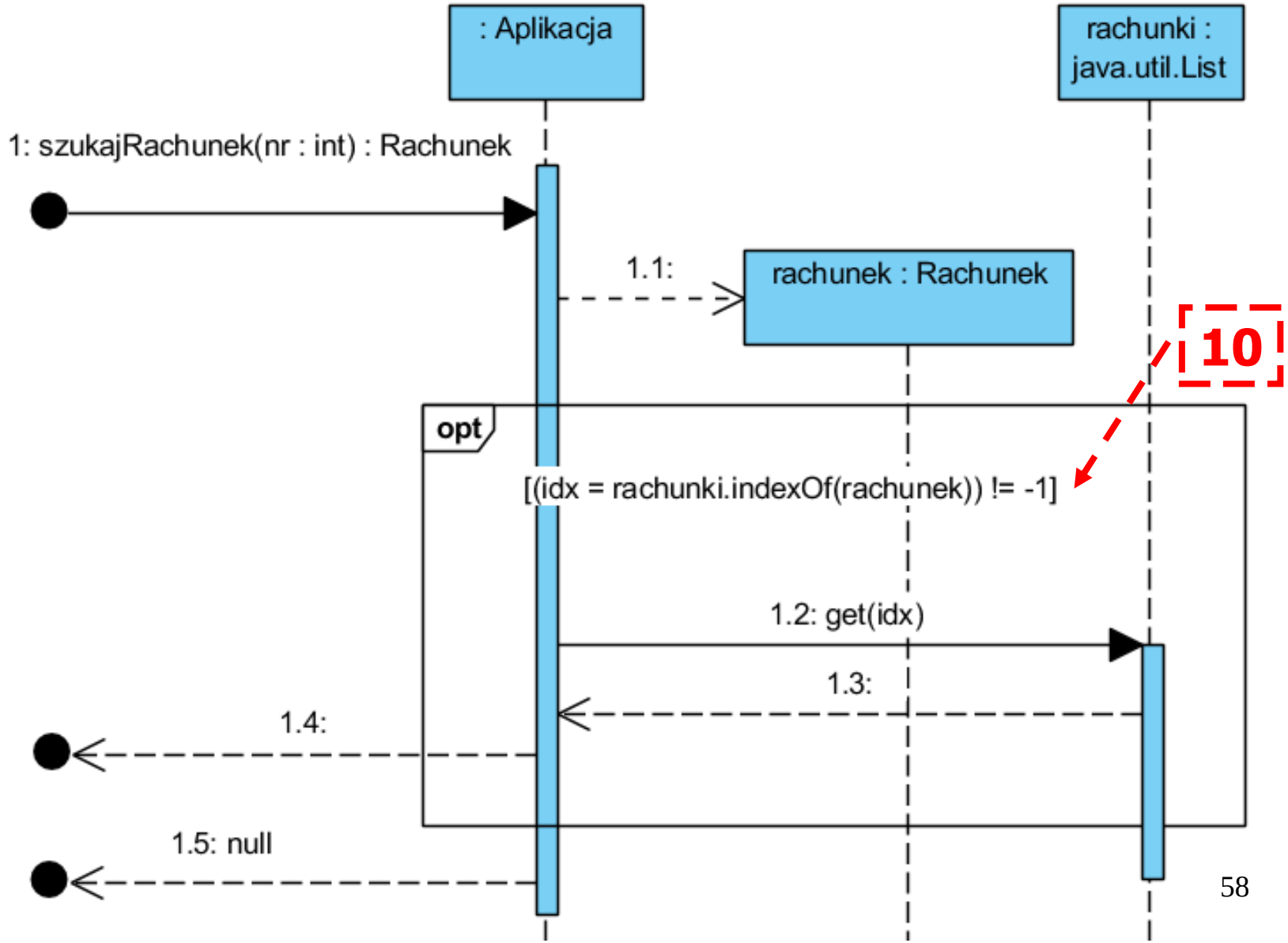
PU Wstawianie nowego rachunku
(8) void wstawRachunek(int nr)



```
                                                    //Aplikacja

private List <Rachunek> rachunki  = new ArrayList <>();

 public void wstawRachunek (int nr)
  {
     Rachunek rachunek=new Rachunek(nr);
      if (szukajRachunek(nr) == null)
           rachunki.add(rachunek);
  }
```

PU Szukanie rachunku

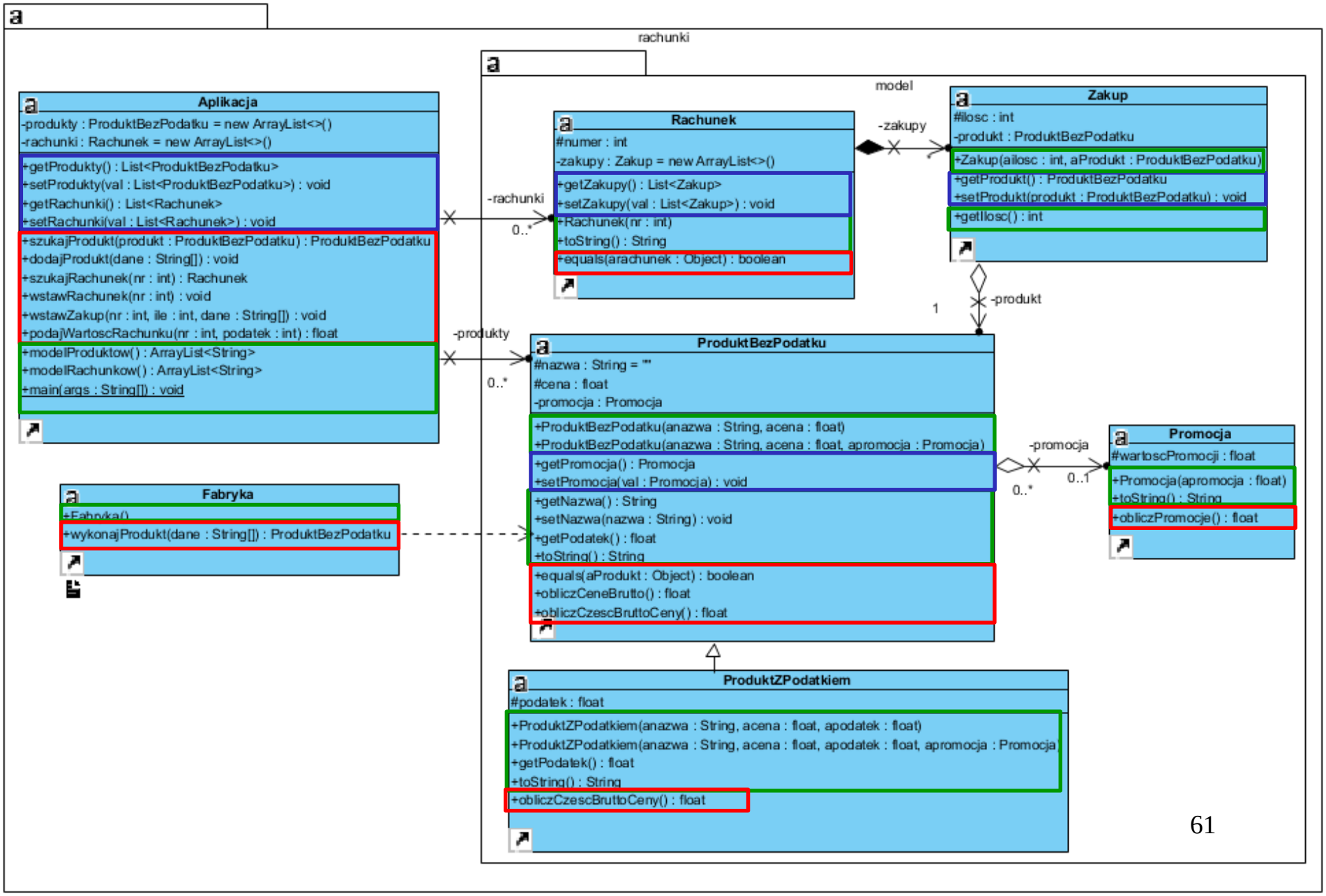
(9) Rachunek szukajRachunek(int nr)



```
                                                        //Aplikacja
private List <Rachunek> rachunki  = new ArrayList <>();

 public  Rachunek szukajRachunek (int nr)
 {
    Rachunek rachunek = new Rachunek(nr);
    int idx;
     if ((idx=rachunki.indexOf(rachunek)) != -1)
      {
         rachunek=rachunki.get(idx);
         return rachunek;
      }
     return null;
 }
```

```
                                                    //Rachunek

public boolean equals (Object aRachunek)
  {
    Rachunek rachunek= (Rachunek)aRachunek;
    return numer== rachunek.numer ;
  }
```

Projekt powiązań | Metody przypadków użycia | Decyzja projektowa

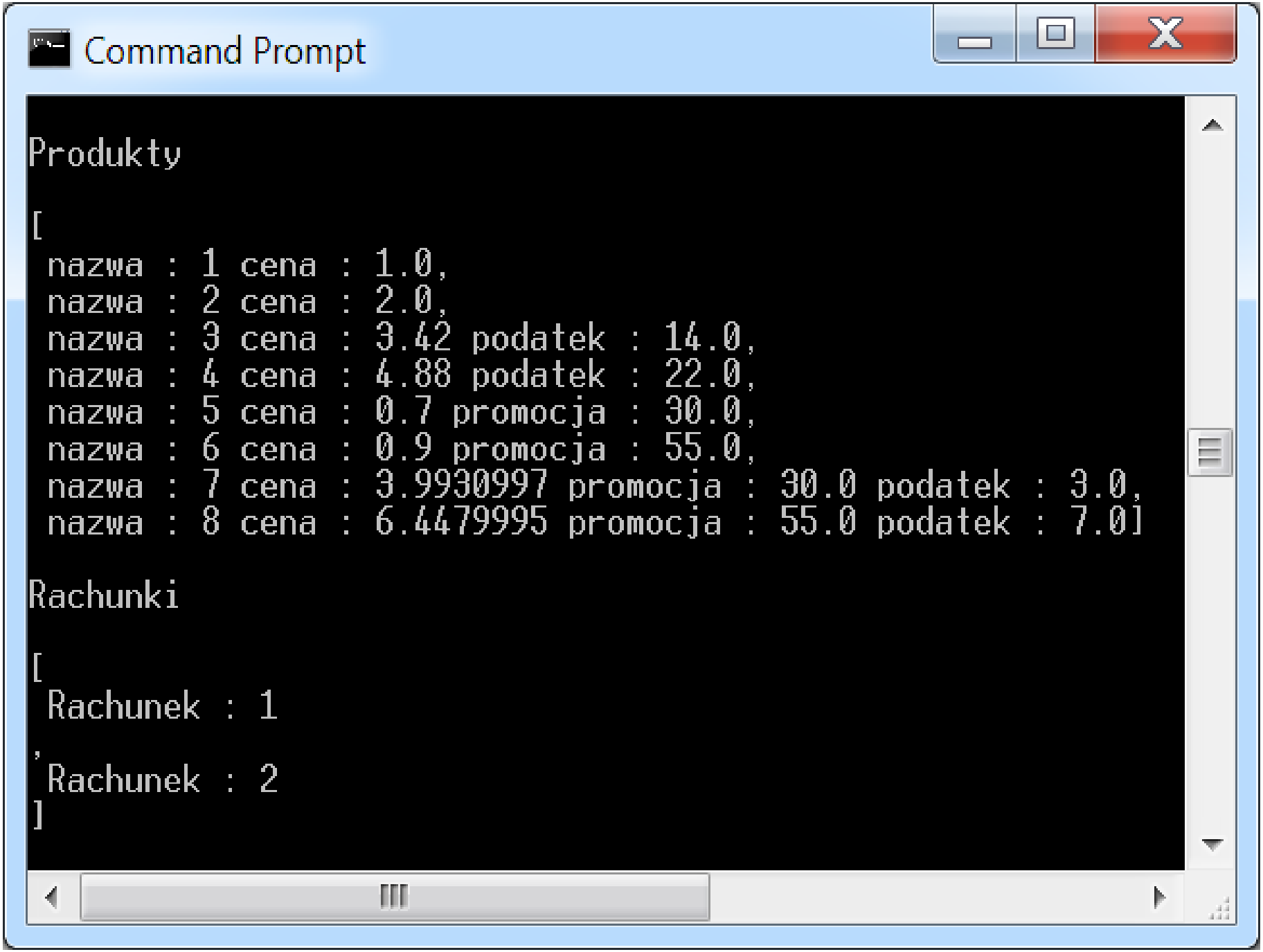


```
//Decyzje na poziomie tworzenia kodu
//Aplikacja
public ArrayList<String> modelRachunkow() {
      ArrayList<String> modelRachunkow = new ArrayList();
      for (Rachunek rachunek : rachunki) {
         modelRachunkow.add("\n" + rachunek.toString());
      }
      return modelRachunkow;
   }
```

```
//Rachunek
public String toString() {
 Zakup z;
 StringBuilder sb = new StringBuilder();
 sb.append(" Rachunek : ");
 sb.append(numer).append("\n");
 for (Zakup zakup:zakupy)
  sb.append(zakup.toString()).append("\n");
 return sb.toString();
}
```

```
//Zakup
public String toString() {
  StringBuilder sb =new StringBuilder();
   sb.append(" ilosc : ");
   sb.append(ilosc);
   sb.append(" Produkt : ");
   sb.append(produkt.toString());
   return sb.toString();
 }
```

```
//c.d. kodu metody main po implementacji przypadków użycia:
// Szukanie rachunku i Wstawianie nowego rachunku

        app.wstawRachunek(1);
        app.wstawRachunek(1);
        app.wstawRachunek(2);
        System.out.println("\nRachunki\n");
        System.out.println("\nRachunki\n");
        System.out.println(app.modelRachunkow());
    }
}
```


Command Prompt
Produkty
[
nazwa : 1 cena : 1.0,
nazwa : 2 cena : 2.0,
nazwa : 3 cena : 3.42 podatek : 14.0,
nazwa : 4 cena : 4.88 podatek : 22.0,
nazwa : 5 cena : 0.7 promocja : 30.0,
nazwa : 6 cena : 0.9 promocja : 55.0,
nazwa : 7 cena : 3.9930997 promocja : 30.0 podatek : 3.0,
nazwa : 8 cena : 6.4479995 promocja : 55.0 podatek : 7.0]
Rachunki
[
Rachunek : 1
,Rachunek : 2
]

Iteracja 3

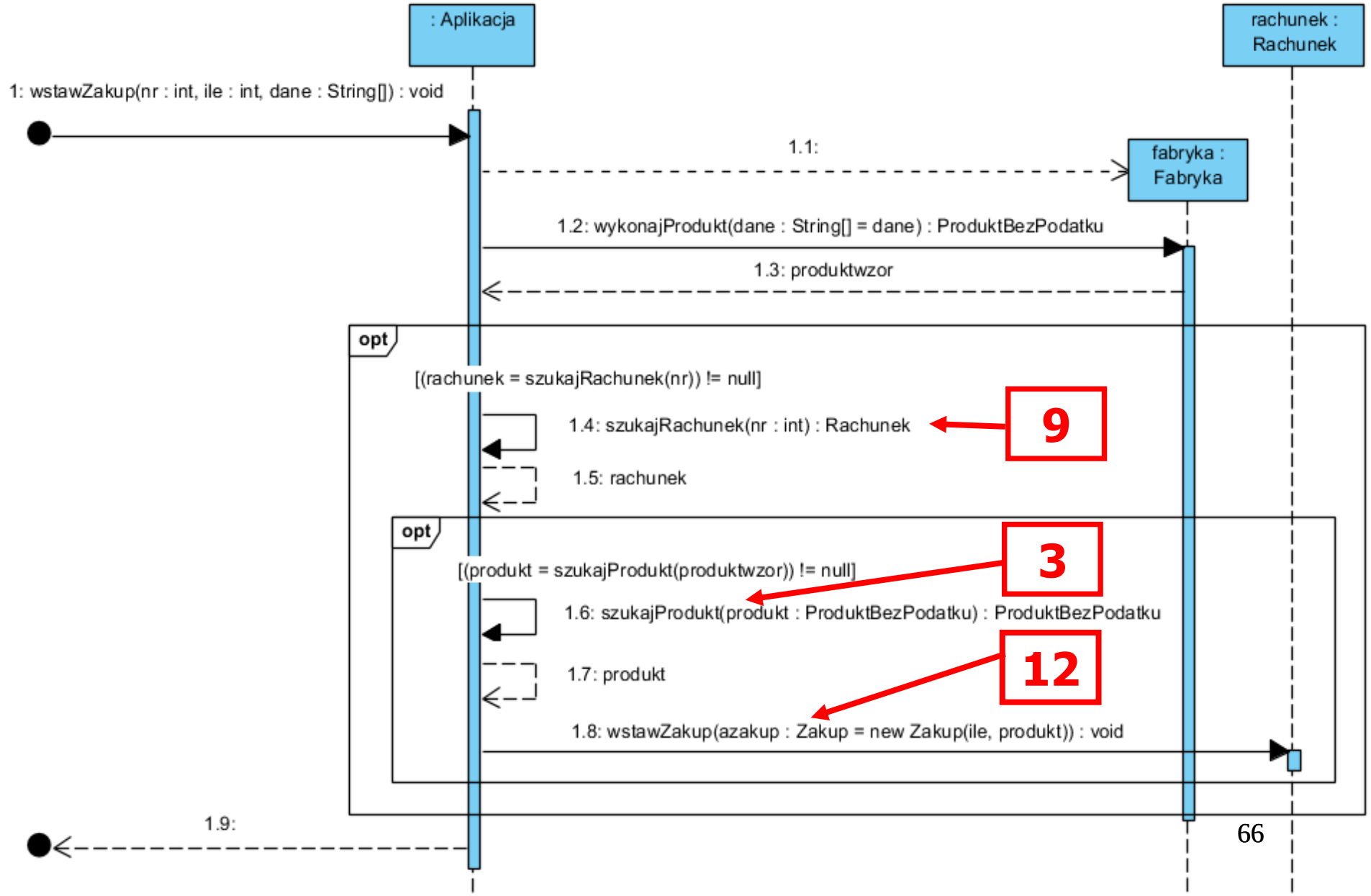
Projekt przypadku użycia

„**Wstawianie nowego zakupu**”

za pomocą diagramu sekwencji i diagramu klas. Diagram klas jest uzupełniany metodami zidentyfikowanymi podczas projektowania scenariusza przypadku użycia za pomocą diagramu sekwencji.

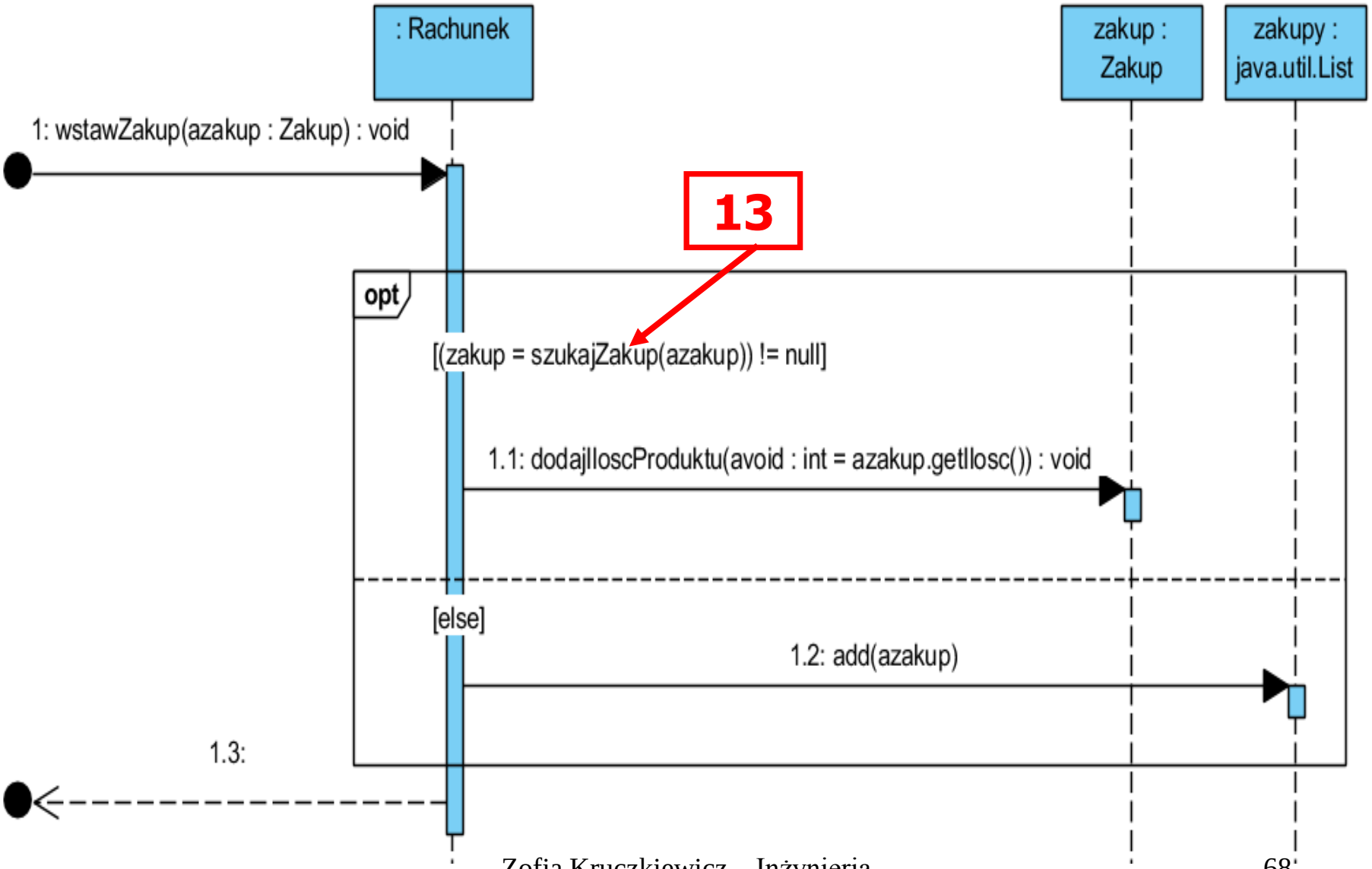
PU Wstawianie nowego zakupu

(11) void wstawZakup (int nr, int ailosc, String dane[])



```
                                                          //Aplikacja

public   void wstawZakup (int nr, int ile, String dane[])
{
  Rachunek rachunek;
   Fabryka fabryka = new Fabryka();
   ProduktBezPodatku produkt1 = fabryka.wykonajProdukt(dane);
                                                          // 1-a iteracja
   if ((rachunek=szukajRachunek(nr)) != null)            // 2-a iteracja
     if ((produkt1=szukajProdukt(produkt1)) != null)  // 1-a iteracja
       rachunek.wstawZakup(new Zakup(ile, produkt1));
 }
```

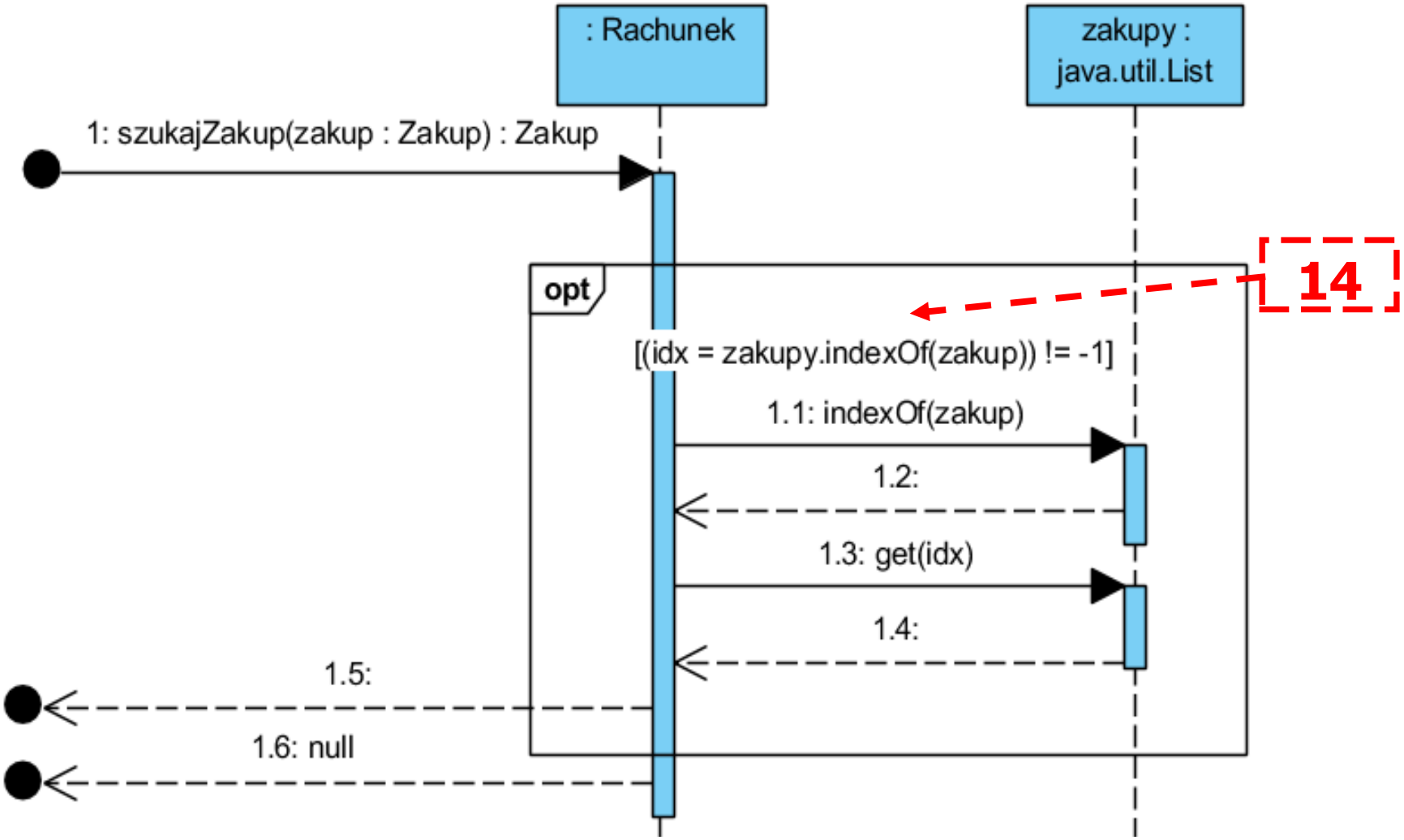
(12) void wstawZakup(Zakup azakup)



```
                                                    //Rachunek

 private List<Zakup> zakupy = new ArrayList<>();

public void wstawZakup (Zakup azakup)
    {
      Zakup zakup;
      if ((zakup = szukajZakup(azakup)) != null)
            zakup.dodajIloscProduktu(azakup.getIlosc());
      else
            zakupy.add(azakup);
  }
```

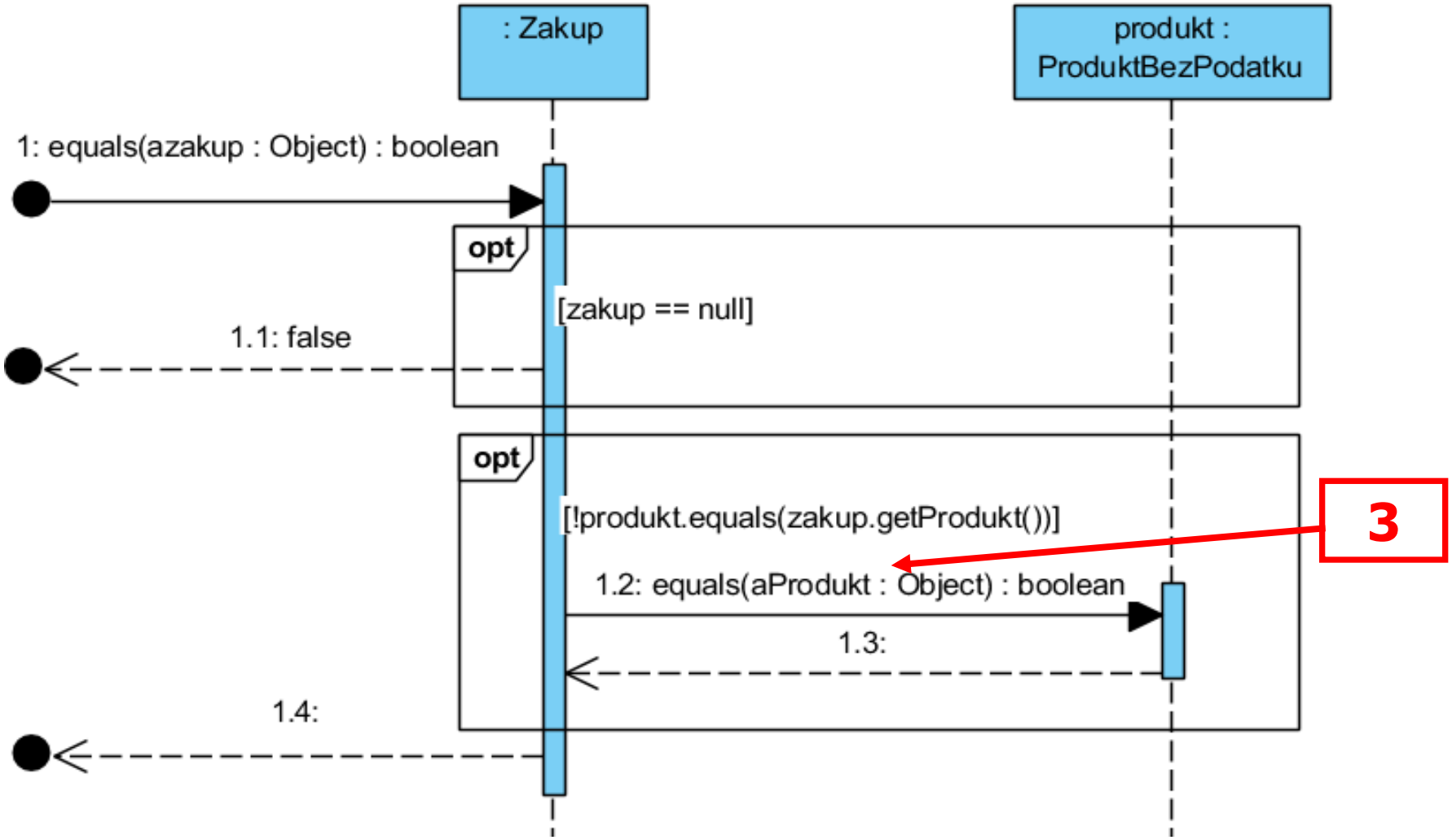
```
                                                    //Zakup

public  void dodajIloscProduktu ( int avoid)
  {
     ilosc+=avoid;
  }

public  int getIlosc ()
  {
     return ilosc;
 }
```

(13) Zakup szukajZakup(Zakup zakup)



```
//Rachunek

private List<Zakup> zakupy = new ArrayList<>();

public Zakup szukajZakup (Zakup zakup)
 {
   int idx;
   if ((idx=zakupy.indexOf(zakup))!=-1)
      {
       return zakupy.get(idx);
      }
    return null;
 }
```

(14) boolean equals(Object zakup)

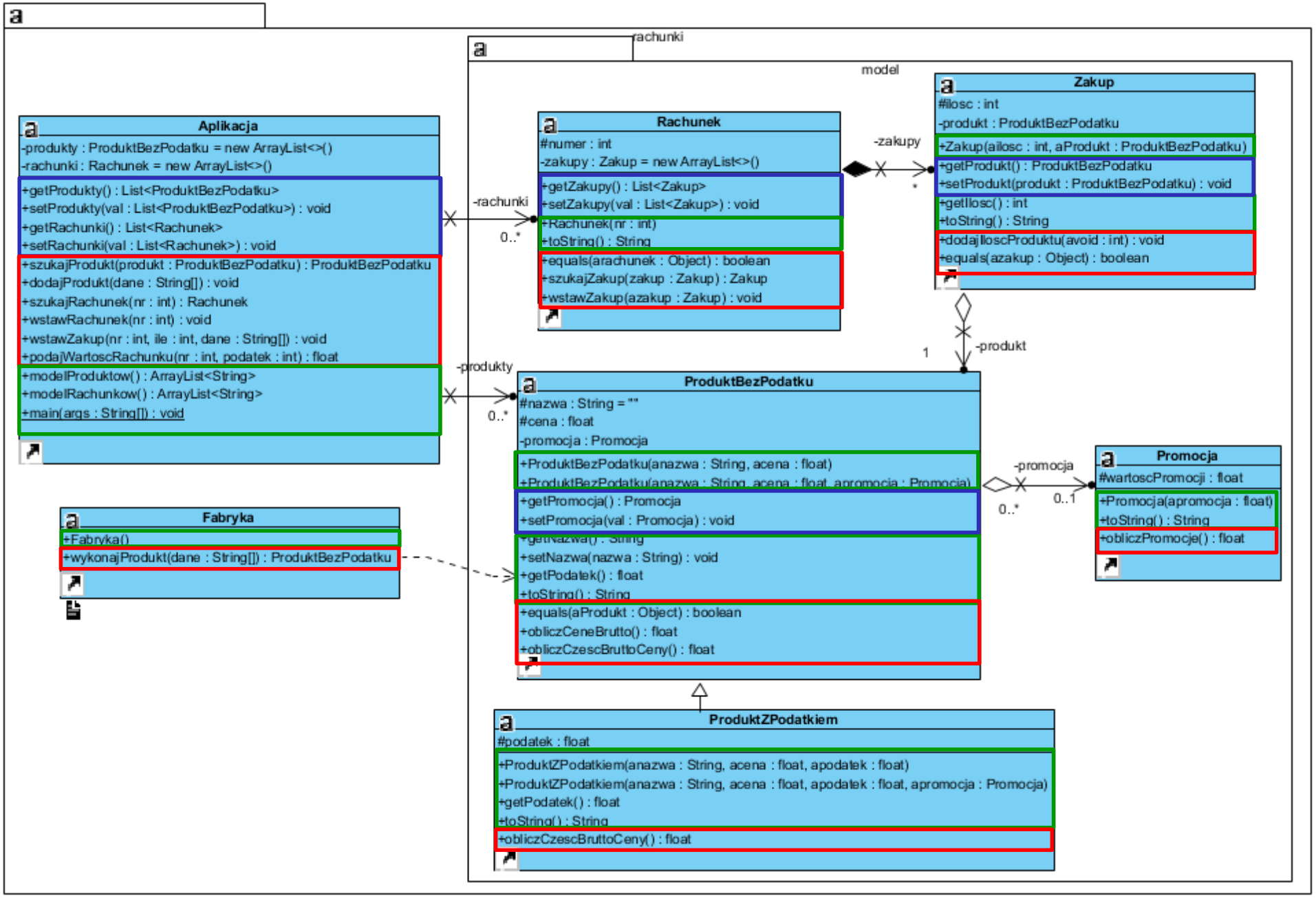


```
                                                        //Zakup

private   ProduktBezPodatku produkt;

 public boolean equals ( Object azakup )
  {
   Zakup zakup=(Zakup)azakup;
    if ( zakup == null )
          return false;
    return produkt.equals(zakup.produkt);    // 1-a iteracja
  }
```

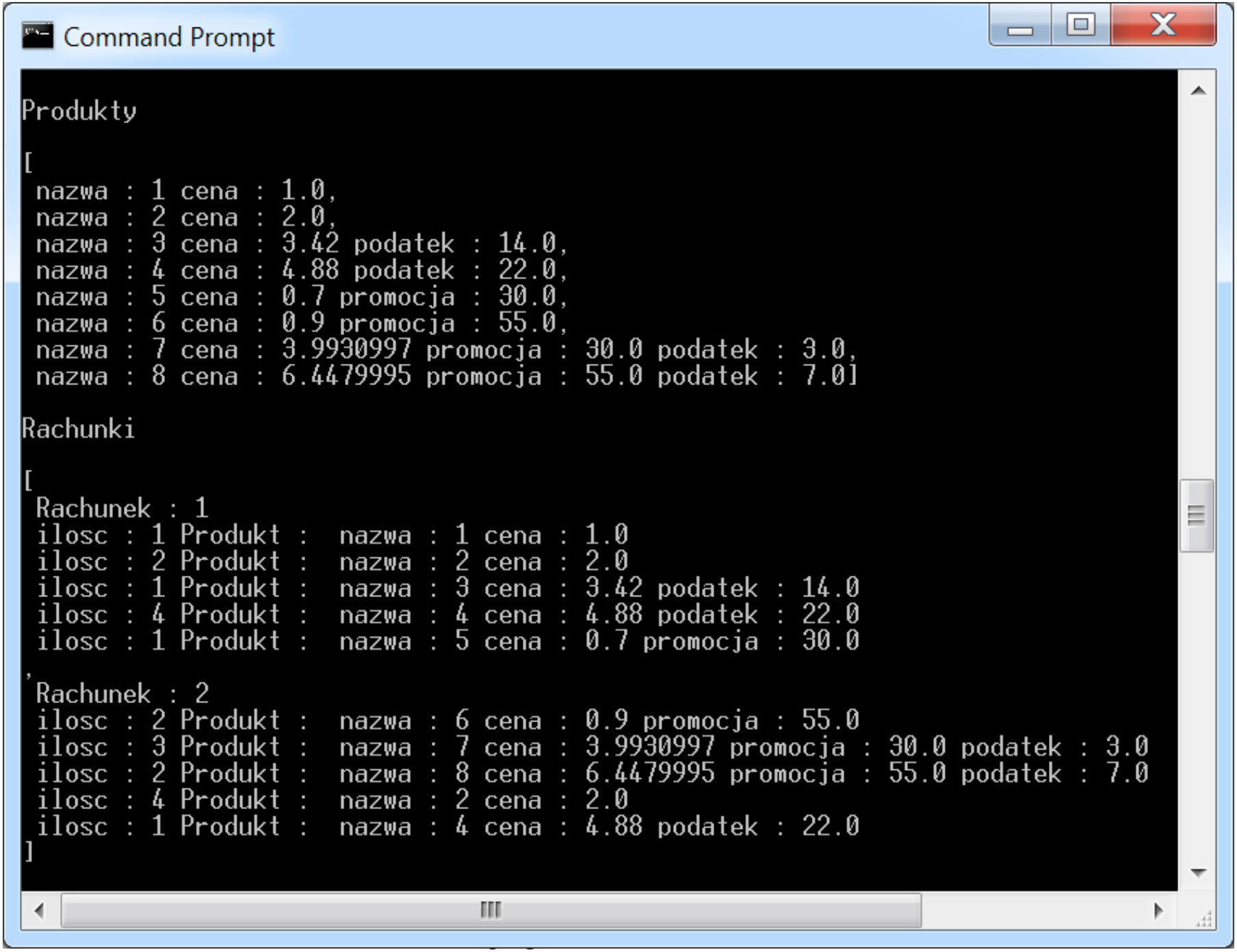
Projekt powiązań

Metody przypadków użycia

Decyzja projektowa



```
//c.d. kodu metody main po implementacji przypadków użycia:
// Wstawianie nowego zakupu
        app.wstawZakup(1, 1, dane1);
        app.wstawZakup(1, 2, dane2);
        app.wstawZakup(1, 1, dane3);
        app.wstawZakup(1, 4, dane4);
        app.wstawZakup(1, 1, dane5);
        app.wstawZakup(2, 1, dane6);
        app.wstawZakup(2, 3, dane7);
        app.wstawZakup(2, 1, dane8);
        app.wstawZakup(2, 4, dane2);
        app.wstawZakup(2, 1, dane4);
        app.wstawZakup(2, 1, dane6);
        app.wstawZakup(2, 1, dane8);
       System.out.println("\nRachunki\n");
       System.out.println(app.modelRachunkow());
    }
}
```


Command Prompt
Produkty
[
nazwa : 1 cena : 1.0,
nazwa : 2 cena : 2.0,
nazwa : 3 cena : 3.42 podatek : 14.0,
nazwa : 4 cena : 4.88 podatek : 22.0,
nazwa : 5 cena : 0.7 promocja : 30.0,
nazwa : 6 cena : 0.9 promocja : 55.0,
nazwa : 7 cena : 3.9930997 promocja : 30.0 podatek : 3.0,
nazwa : 8 cena : 6.4479995 promocja : 55.0 podatek : 7.0]
Rachunki
[
Rachunek : 1
ilosc : 1 Produkt : nazwa : 1 cena : 1.0
ilosc : 2 Produkt : nazwa : 2 cena : 2.0
ilosc : 1 Produkt : nazwa : 3 cena : 3.42 podatek : 14.0
ilosc : 4 Produkt : nazwa : 4 cena : 4.88 podatek : 22.0
ilosc : 1 Produkt : nazwa : 5 cena : 0.7 promocja : 30.0
,Rachunek : 2
ilosc : 2 Produkt : nazwa : 6 cena : 0.9 promocja : 55.0
ilosc : 3 Produkt : nazwa : 7 cena : 3.9930997 promocja : 30.0 podatek : 3.0
ilosc : 2 Produkt : nazwa : 8 cena : 6.4479995 promocja : 55.0 podatek : 7.0
ilosc : 4 Produkt : nazwa : 2 cena : 2.0
ilosc : 1 Produkt : nazwa : 4 cena : 4.88 podatek : 22.0
]

Iteracja 4

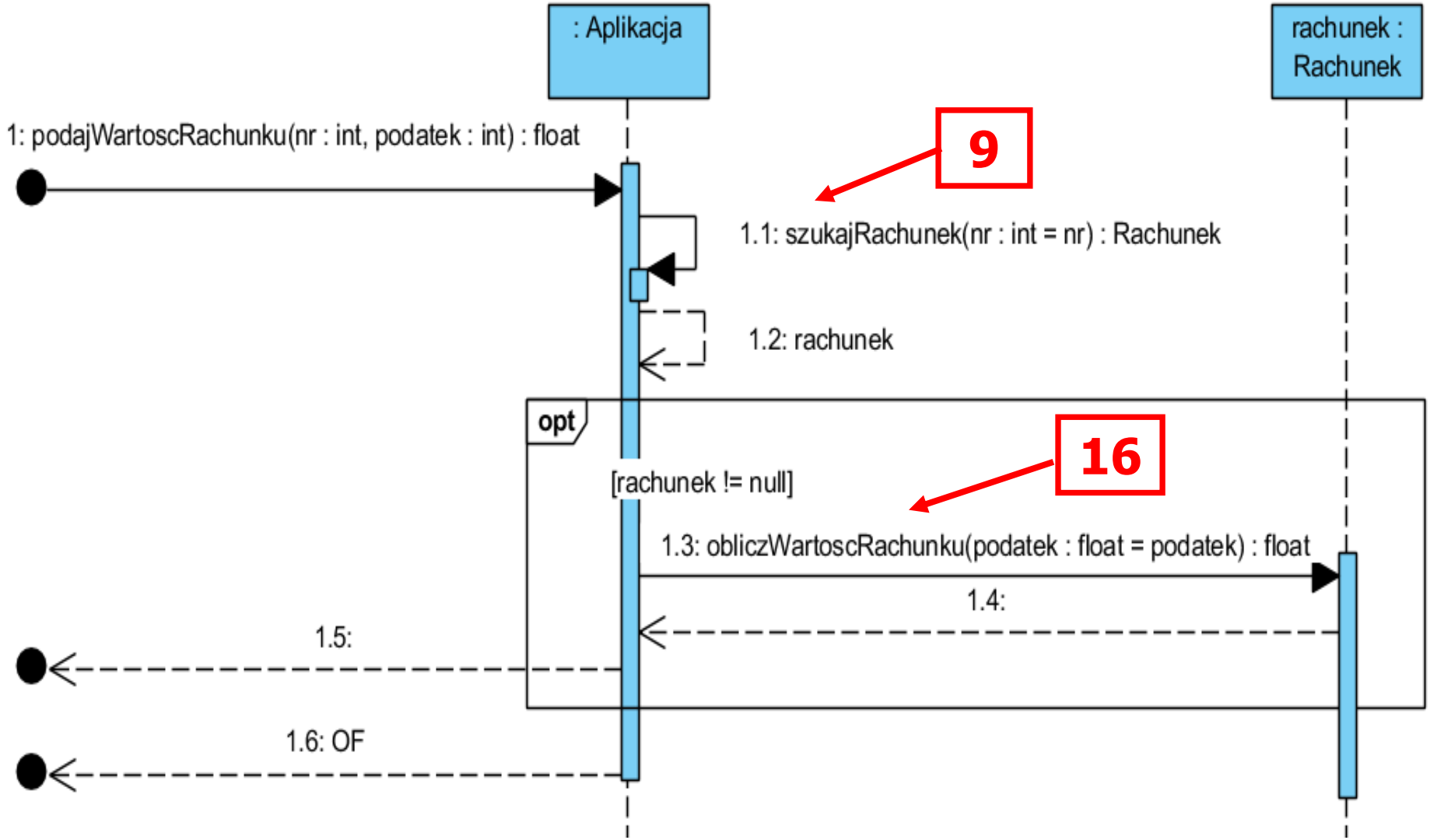
Projekt przypadku użycia

„**Obliczanie wartości rachunku**”

za pomocą diagramu sekwencji i diagramu klas. Diagram klas jest uzupełniany metodami zidentyfikowanymi podczas projektowania scenariusza przypadku użycia za pomocą diagramu sekwencji.

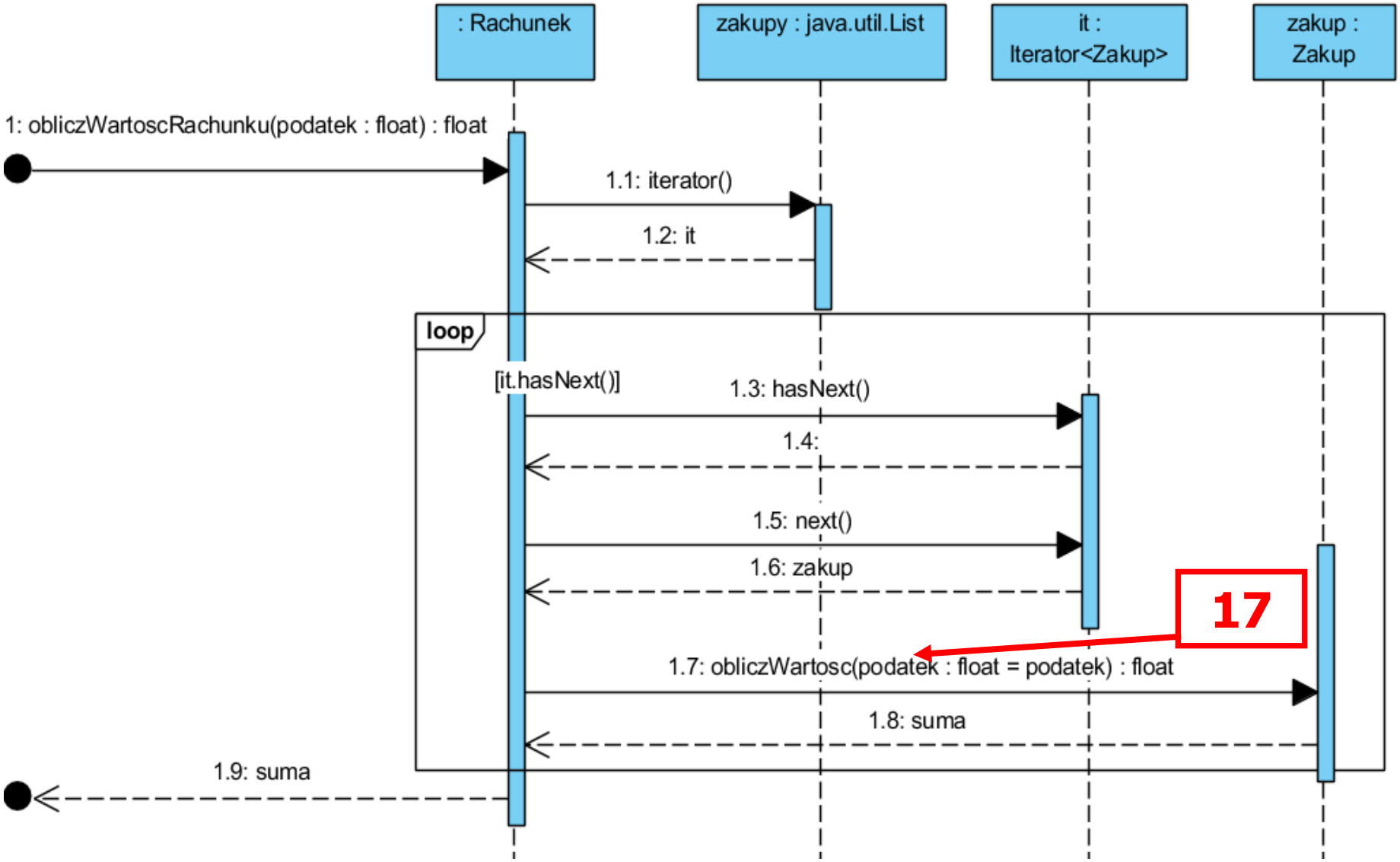
PU Obliczanie wartosci rachunku

(15) float podajWartoscRachunku(int nr, int podatek)



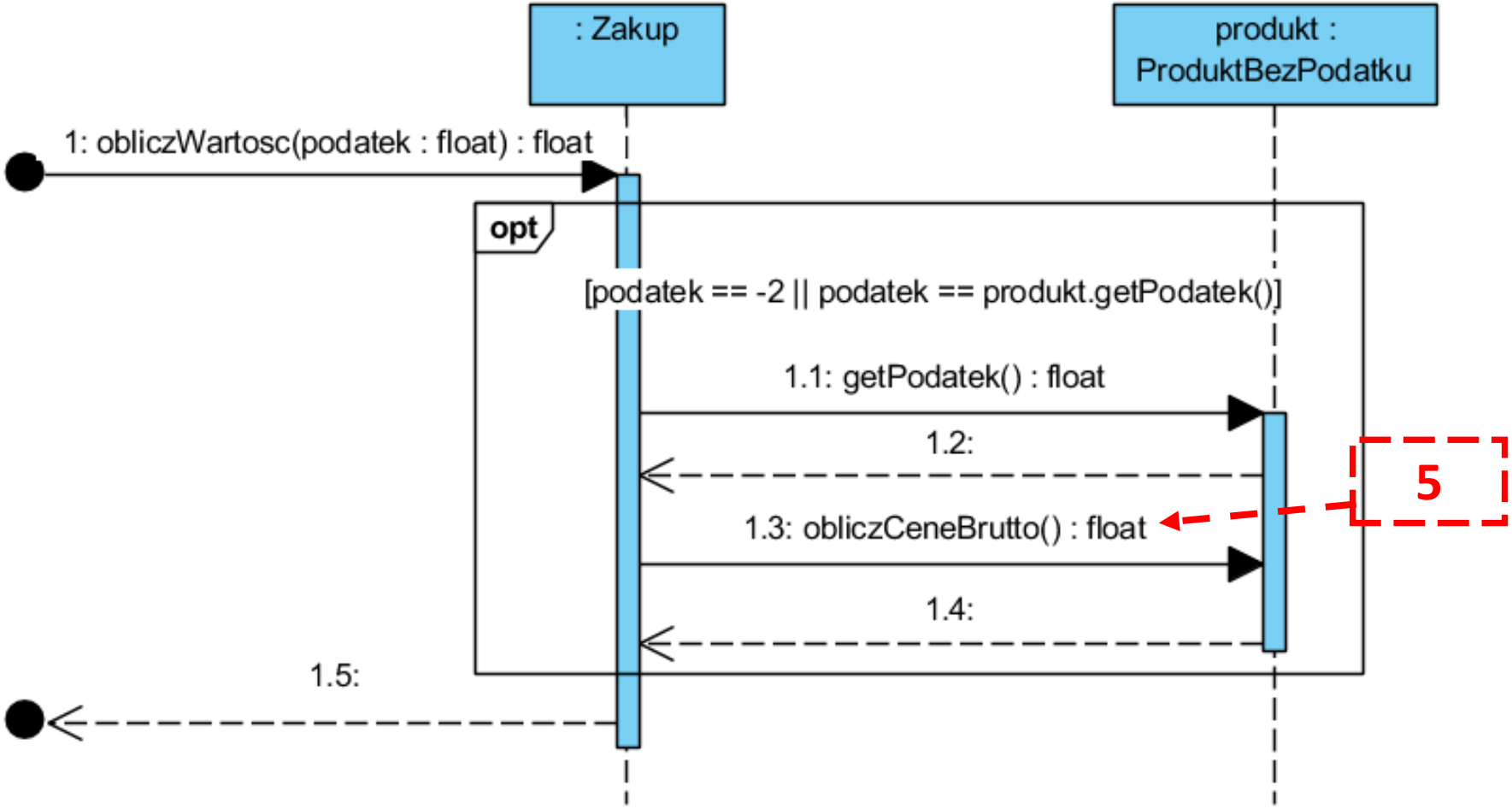
```
                                                    //Aplikacja

public  float podajWartoscRachunku (int nr, int podatek)
 {
   Rachunek rachunek;
   rachunek = szukajRachunek(nr);    // 2-a iteracja
   if (rachunek != null)
      return rachunek.obliczWartoscRachunku(podatek);
   return 0F;
 }
```

(16) float obliczWartoscRachunku(int podatek)



```
                                                    //Rachunek

private List<Zakup> zakupy = new ArrayList<>();

public float obliczWartoscRachunku (int podatek)
 {
   float suma=0;
   Zakup zakup;
   Iterator <Zakup> it=zakupy.iterator();
   while (it.hasNext())
    {  zakup = it.next();
       suma += zakup.obliczWartosc(podatek);
    }
   return suma;
  }
```

(17) float obliczWartosc(int podatek)

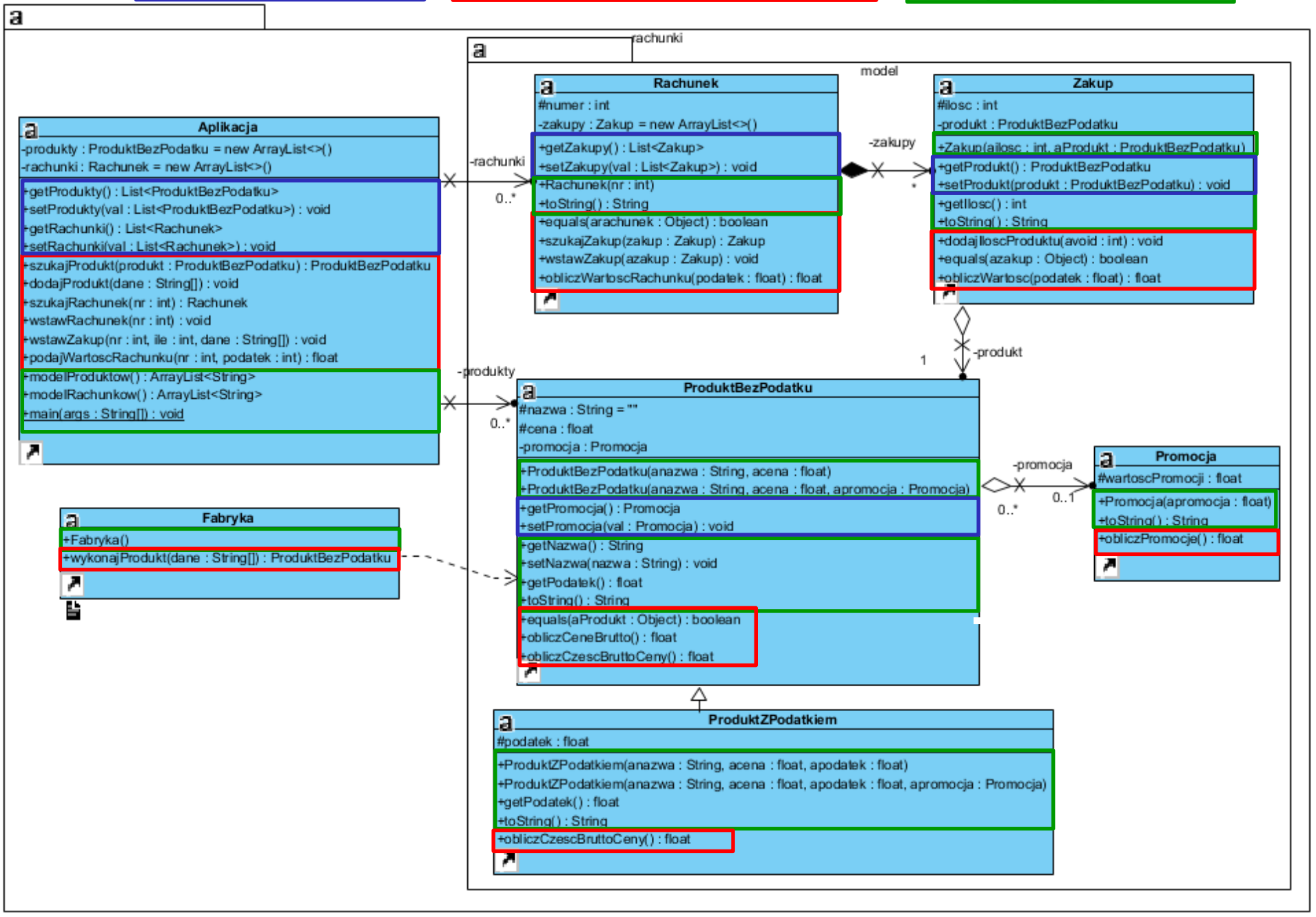


```
                                                          //Zakup


private   ProduktBezPodatku produkt = null;

public  float obliczWartosc (int podatek)
  {
   if (podatek==-2 ||podatek==produkt.getPodatek())
      return ilosc*produkt.obliczCeneBrutto();     // 1-a iteracja
   return 0F;
  }
```

Projekt powiązań

Metody przypadków użycia

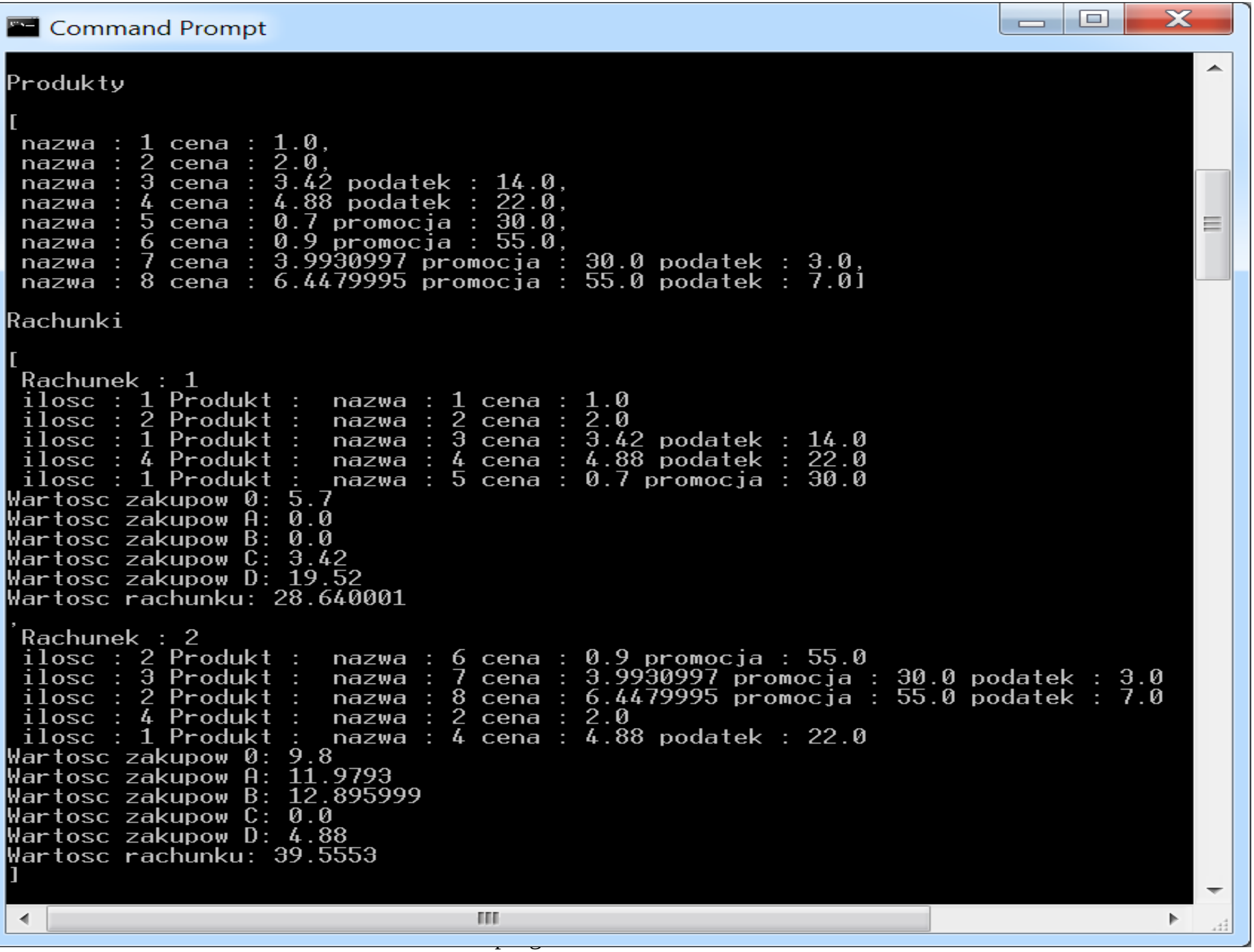
Decyzja projektowa



```
// Rachunek – zmiana kodu metody toString(),
// drukująca wartości rachunku w rożnych kategoriach

public String toString()
{
  StringBuilder sb = new StringBuilder();
  sb.append(" Rachunek : ");
  sb.append(numer).append("\n");
   for (Zakup zakup:Zakupy)
          sb.append(zakup.toString()).append("\n");
  sb.append("Wartosc zakupow 0: ").append(obliczWartoscRachunku(-1)).append("\n");
  sb.append("Wartosc zakupow A: ").append(obliczWartoscRachunku(3)).append("\n");
  sb.append("Wartosc zakupow B: ").append(obliczWartoscRachunku(7)).append("\n");
  sb.append("Wartosc zakupow C: ").append(obliczWartoscRachunku(14)).append("\n");
  sb.append("Wartosc zakupow D: ").append(obliczWartoscRachunku(22)).append("\n");
  sb.append("Wartosc rachunku: ").append(obliczWartoscRachunku(-2)).append("\n");
  return sb.toString();
}
```

```
public static void main(String args[])     //kod metody main po implementacji
 { Aplikacja app=new Aplikacja();          //6-u przypadków użycia
   String dane1[]={"0","1","1"};           // identyczny jak po implementacji
   String dane2[]={"0","2","2"};           // 5-go przypadku użycia
   app.dodajProdukt(dane1);
   app.dodajProdukt(dane2);
   app.dodajProdukt(dane1);
   String dane3[]={"2","3","3","14"};
   String dane4[]={"2","4","4","22"};
   app.dodajProdukt(dane3);
   app.dodajProdukt(dane4);
   app.dodajProdukt(dane3);
    String dane5[]={"1","5","1","30"};
   String dane6[]={"1","6","2","5"};
   String dane7[]={"3","7","5.47","3","30"};
   String dane8[]={"3","8","12.46","7","50"};
   app.dodajProdukt(dane5);
   app.dodajProdukt(dane6);
   app.dodajProdukt(dane5);
   app.dodajProdukt(dane7);
   app.dodajProdukt(dane8);
   app.dodajProdukt(dane7);
   System.out.println("\nProdukty\n");
   System.out.println(app.modelProduktow());
```

```
//c.d. kodu metody main po implementacji przypadków użycia:
// Wstawianie nowego zakupu
        app.wstawZakup(1, 1, dane1);
        app.wstawZakup(1, 2, dane2);
        app.wstawZakup(1, 1, dane3);
        app.wstawZakup(1, 4, dane4);
        app.wstawZakup(1, 1, dane5);
        app.wstawZakup(2, 1, dane6);
        app.wstawZakup(2, 3, dane7);
        app.wstawZakup(2, 1, dane8);
        app.wstawZakup(2, 4, dane2);
        app.wstawZakup(2, 1, dane4);
        app.wstawZakup(2, 1, dane6);
        app.wstawZakup(2, 1, dane8);
        System.out.println("\nRachunki\n");
       System.out.println(app.modelRachunkow());
    }
}
```


Command Prompt
Produkty
[
nazwa : 1 cena : 1.0,
nazwa : 2 cena : 2.0,
nazwa : 3 cena : 3.42 podatek : 14.0,
nazwa : 4 cena : 4.88 podatek : 22.0,
nazwa : 5 cena : 0.7 promocja : 30.0,
nazwa : 6 cena : 0.9 promocja : 55.0,
nazwa : 7 cena : 3.9930997 promocja : 30.0 podatek : 3.0,
nazwa : 8 cena : 6.4479995 promocja : 55.0 podatek : 7.0]
Rachunki
[
Rachunek : 1
ilosc : 1 Produkt : nazwa : 1 cena : 1.0
ilosc : 2 Produkt : nazwa : 2 cena : 2.0
ilosc : 1 Produkt : nazwa : 3 cena : 3.42 podatek : 14.0
ilosc : 4 Produkt : nazwa : 4 cena : 4.88 podatek : 22.0
ilosc : 1 Produkt : nazwa : 5 cena : 0.7 promocja : 30.0
Wartosc zakupow 0: 5.7
Wartosc zakupow A: 0.0
Wartosc zakupow B: 0.0
Wartosc zakupow C: 3.42
Wartosc zakupow D: 19.52
Wartosc rachunku: 28.640001
,
Rachunek : 2
ilosc : 2 Produkt : nazwa : 6 cena : 0.9 promocja : 55.0
ilosc : 3 Produkt : nazwa : 7 cena : 3.9930997 promocja : 30.0 podatek : 3.0
ilosc : 2 Produkt : nazwa : 8 cena : 6.4479995 promocja : 55.0 podatek : 7.0
ilosc : 4 Produkt : nazwa : 2 cena : 2.0
ilosc : 1 Produkt : nazwa : 4 cena : 4.88 podatek : 22.0
Wartosc zakupow 0: 9.8
Wartosc zakupow A: 11.9793
Wartosc zakupow B: 12.895999
Wartosc zakupow C: 0.0
Wartosc zakupow D: 4.88
Wartosc rachunku: 39.5553
]