Diagramy klas, diagramy sekwencji

Zofia Kruczkiewicz

Składnia elementów na diagramach UML

1. W prezentacji składni diagramów sekwencji (str.8-18) 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. 21, 34, 44, 58, 70 oraz diagramów sekwencji UML na str. 22-79 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): **https://sparxsystems.com/resources/tutorials/uml2/index.html**
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. **Diagramy sekwencji UML**
   https://sparxsystems.com/resources/tutorials/uml2/sequence-diagram.html
2. **Przykłady diagramów sekwencji i klas – kontynuacja przykładu 2 z wykładu 2 i wykładu 3**
3. **Modelowanie zachowania obiektów za pomocą diagramów sekwencji i aktywności - porównanie**

Diagramy klas, diagramy sekwencji

1. **Diagramy sekwencji UML**

   https://sparxsystems.com/resources/tutorials/uml2/sequence-diagram.html

Diagramy UML 2 – część czwarta

Na podstawie

UML 2.0 Tutorial

https://sparxsystems.com/resources/tutorials/uml2/sequence-diagram.html

Dwa rodzaje diagramów UML 2

Diagramy UML <u>modelowania strukturalnego</u>

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

Diagramy UML <u>modelowania zachowania</u>

- ***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

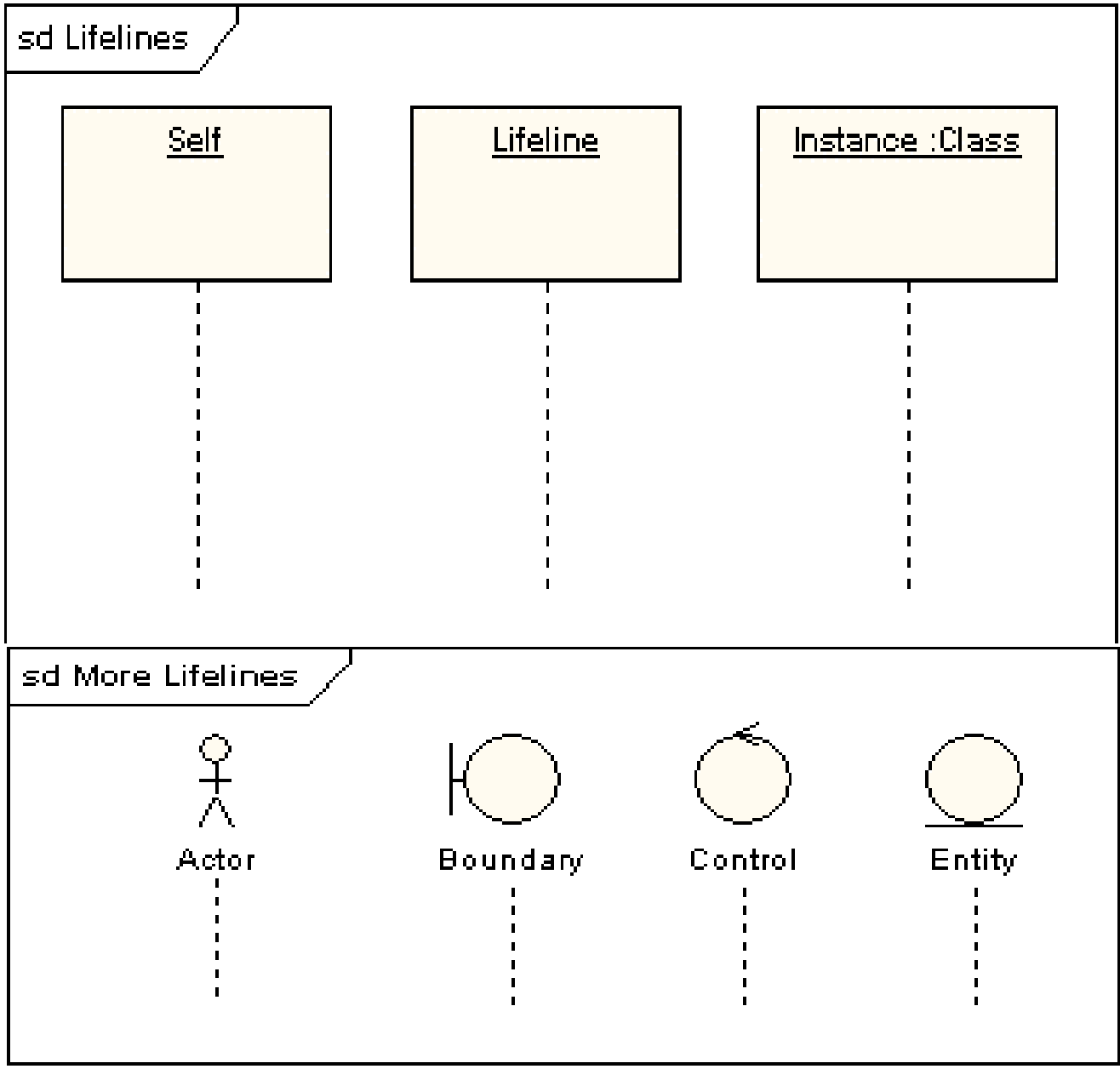
- **nie są używane do wyrażania złożonej logiki proceduralnej**

- **są używane do modelowania 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.

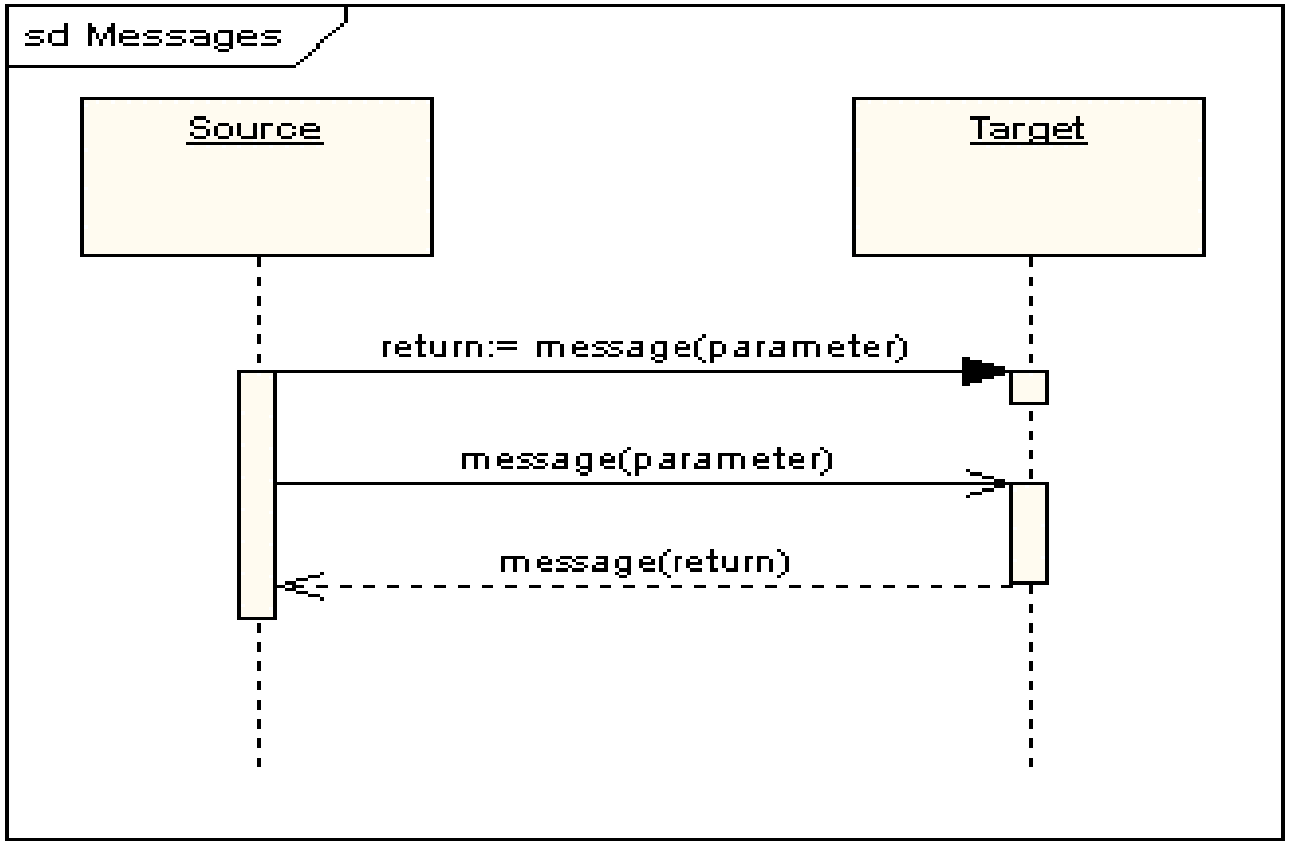


Wiadomości (Messages)

- 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**

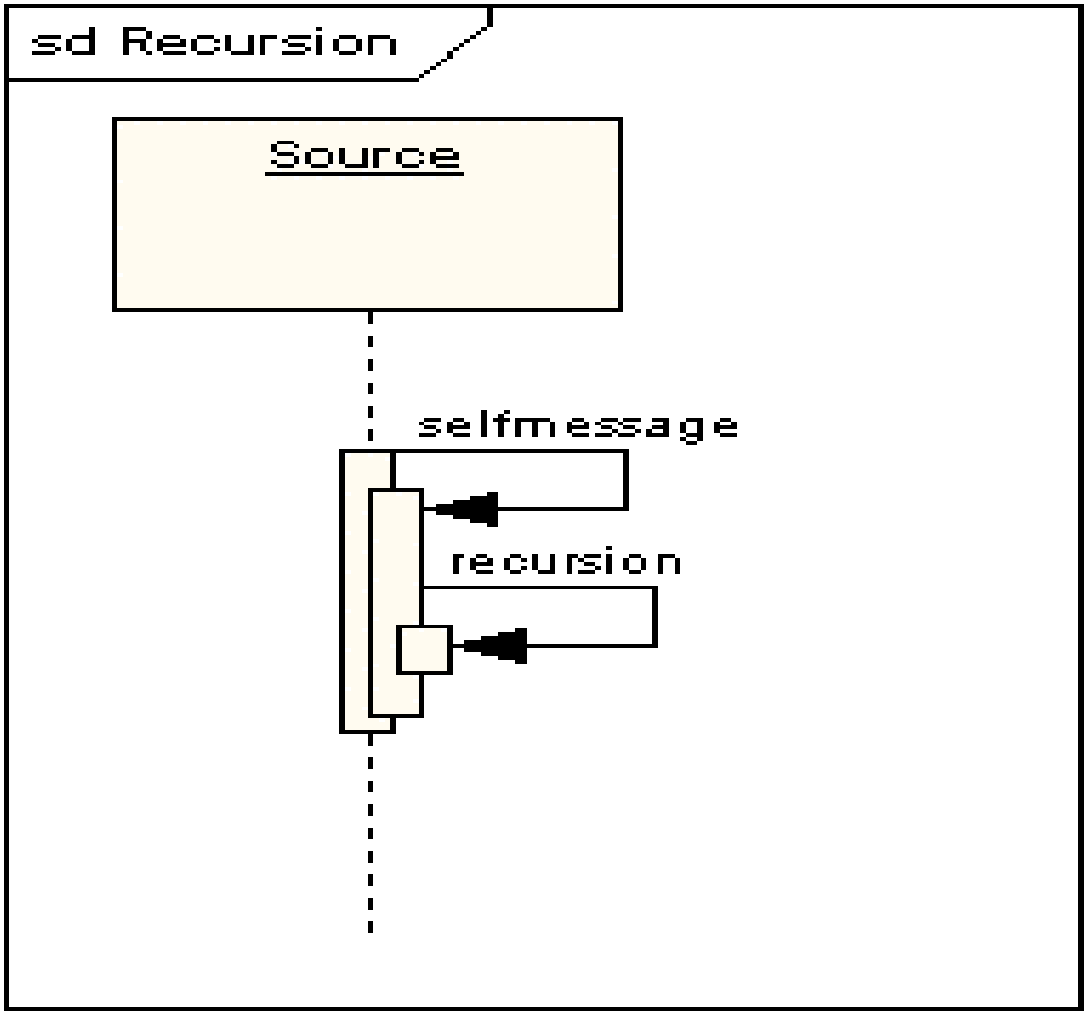
Wykonywanie interakcji (Execution Occurrence)

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).



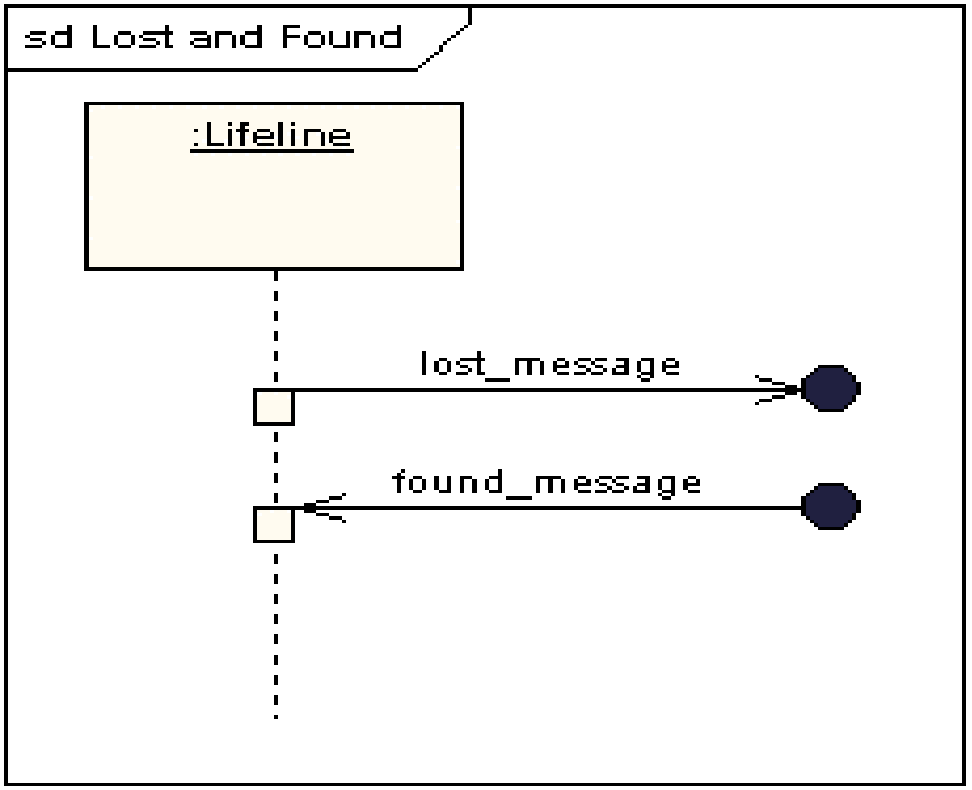
Własne wiadomości (Self Message)

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



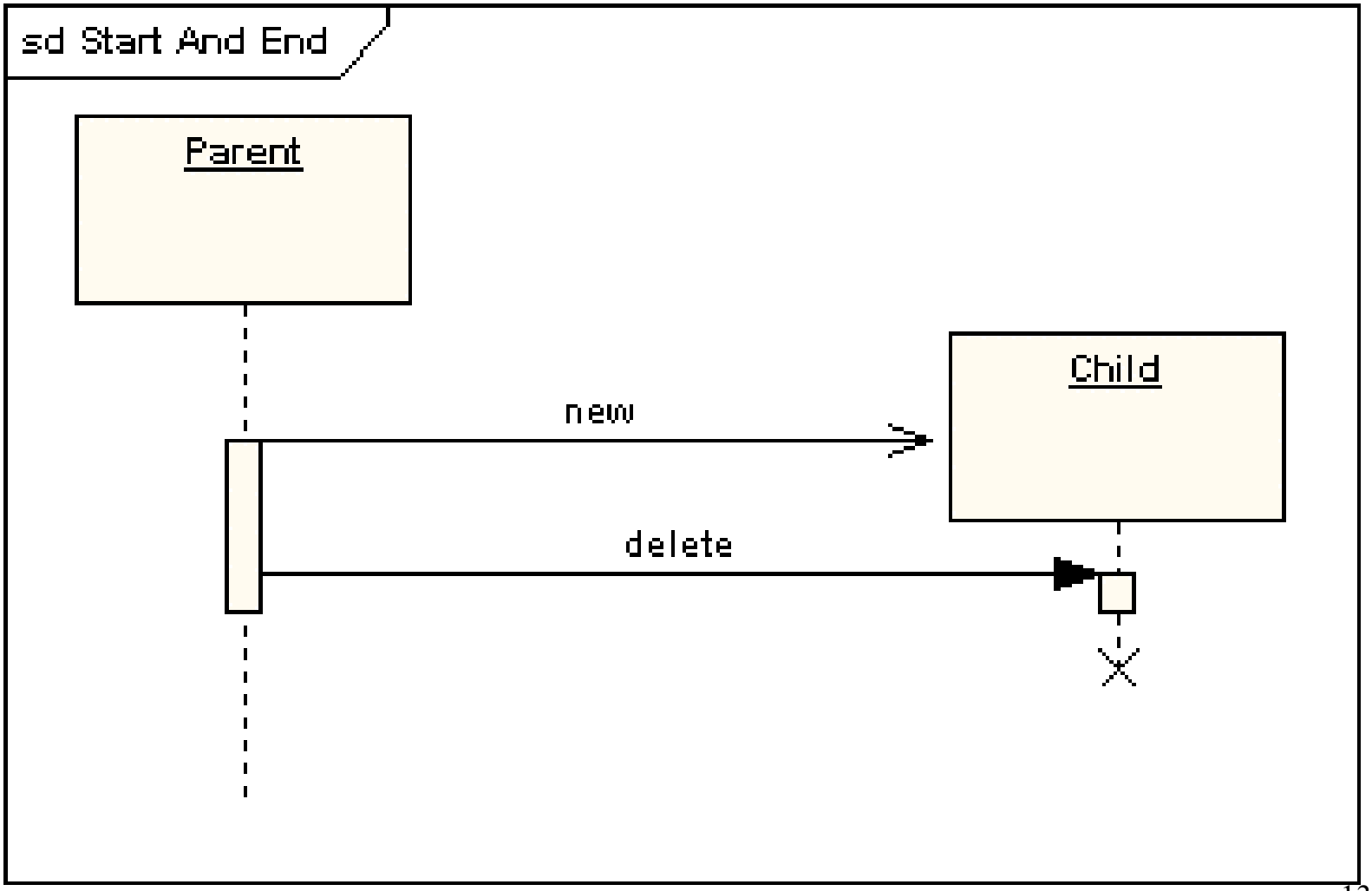
Zgubione i znalezione wiadomości (Lost and Found Messages)

- **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.



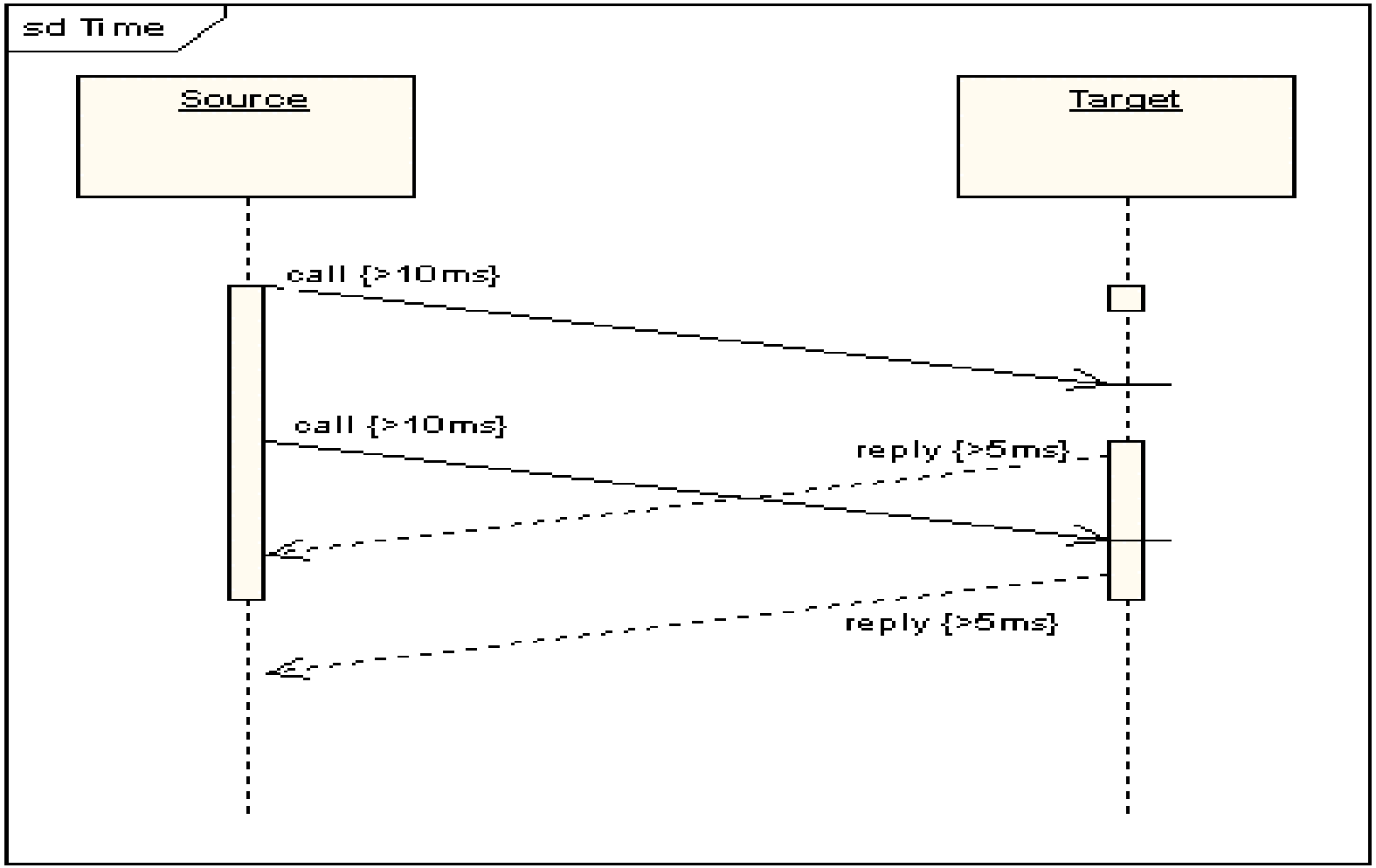
Start linii życia i jej koniec
(Lifeline Start and End)

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.

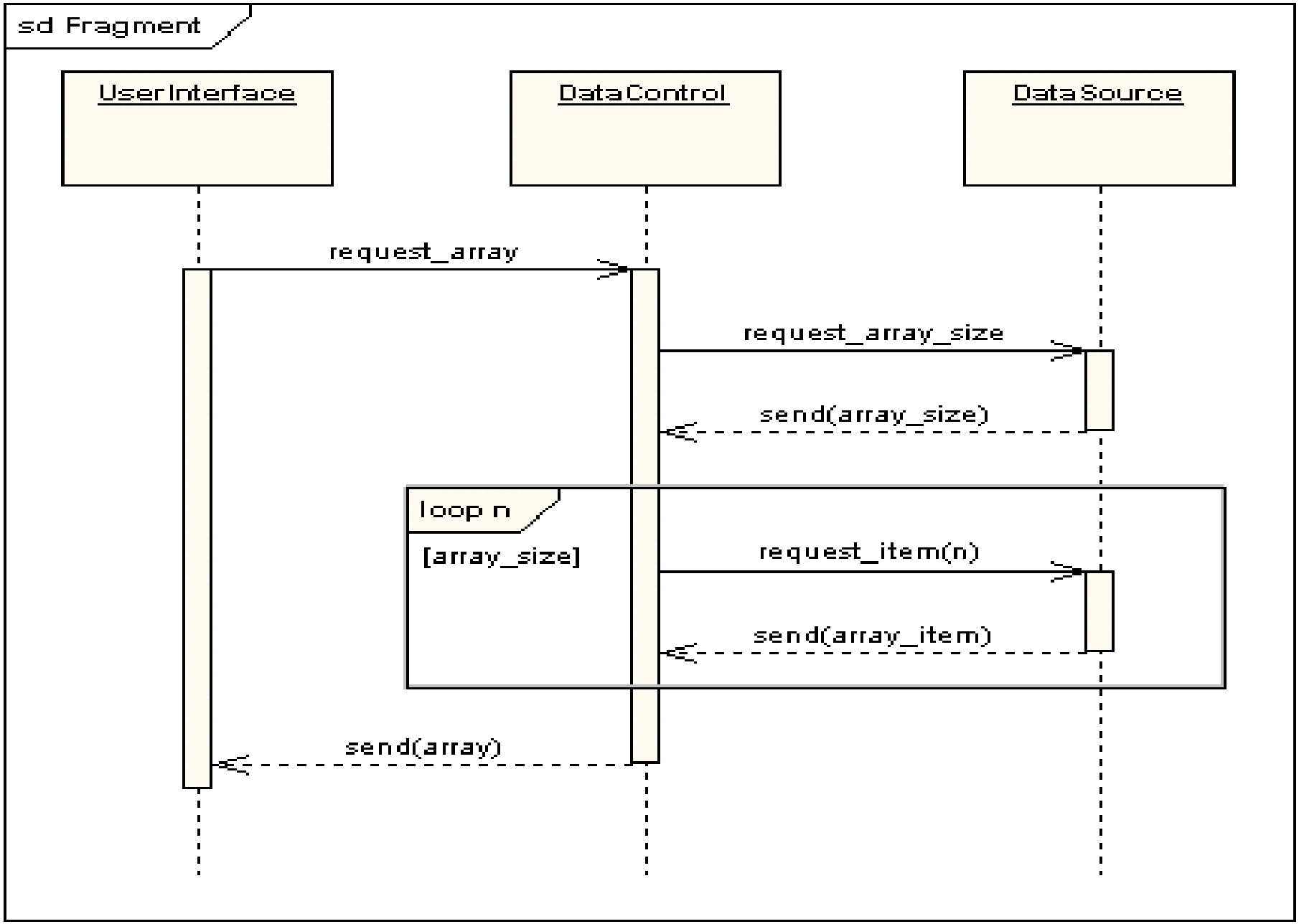


Złożone modelowanie sekwencji wiadomości

Fragmenty ujęte w ramki umożliwiają:

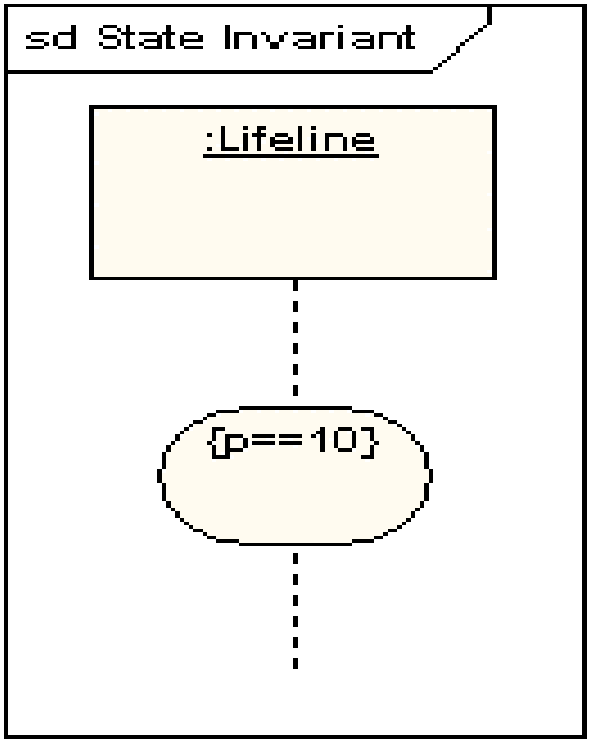
1. **fragmenty alternatywne** (oznaczone "**opt**") modelują konstrukcje **if…then…else**
2. **fragmenty opcjonalne** (oznaczone "**alt**") modelują konstrukcje **switch**.
3. **fragment Break** modeluje alternatywną sekwencję zdarzeń dla pozostałej części diagramu.
4. **fragment równoległy** (oznaczony "**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



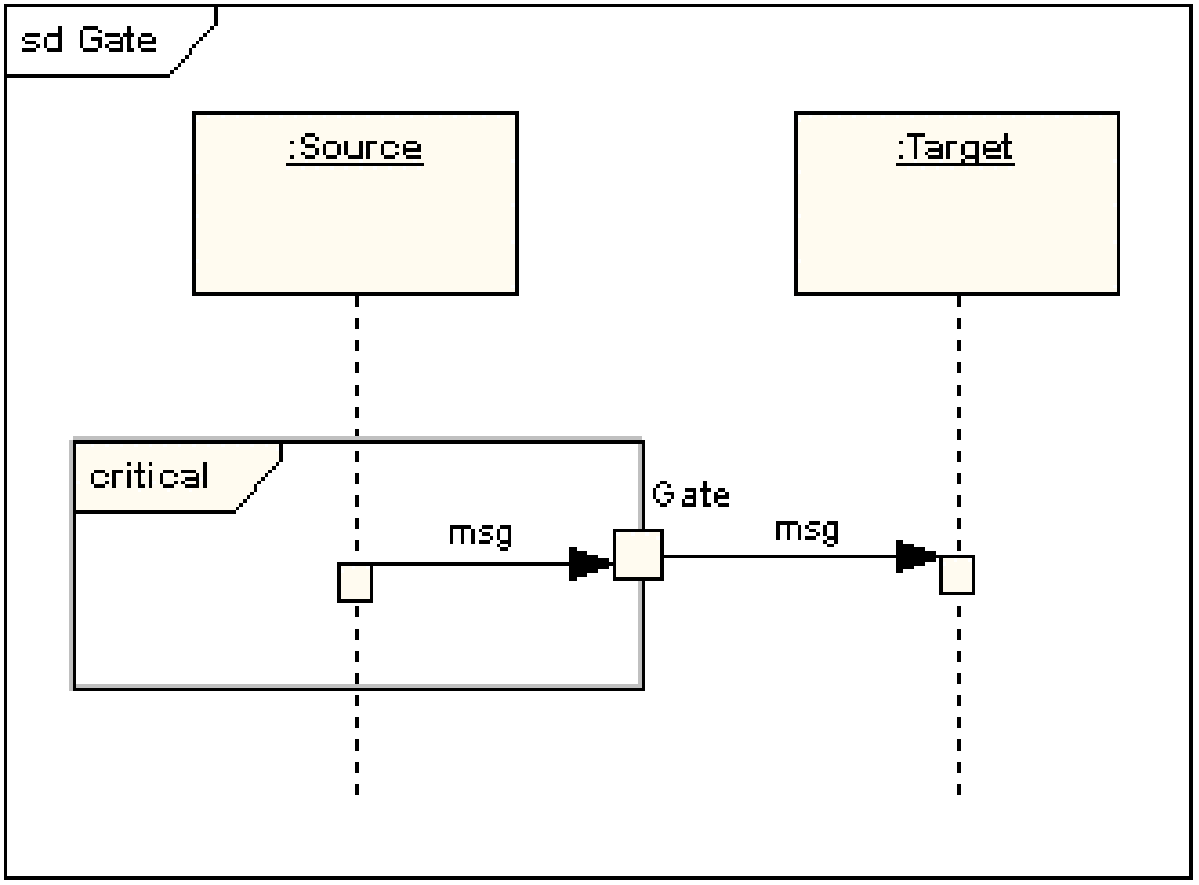
Stan niezmienny lub ciągły (State Invariant /Continuations)

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



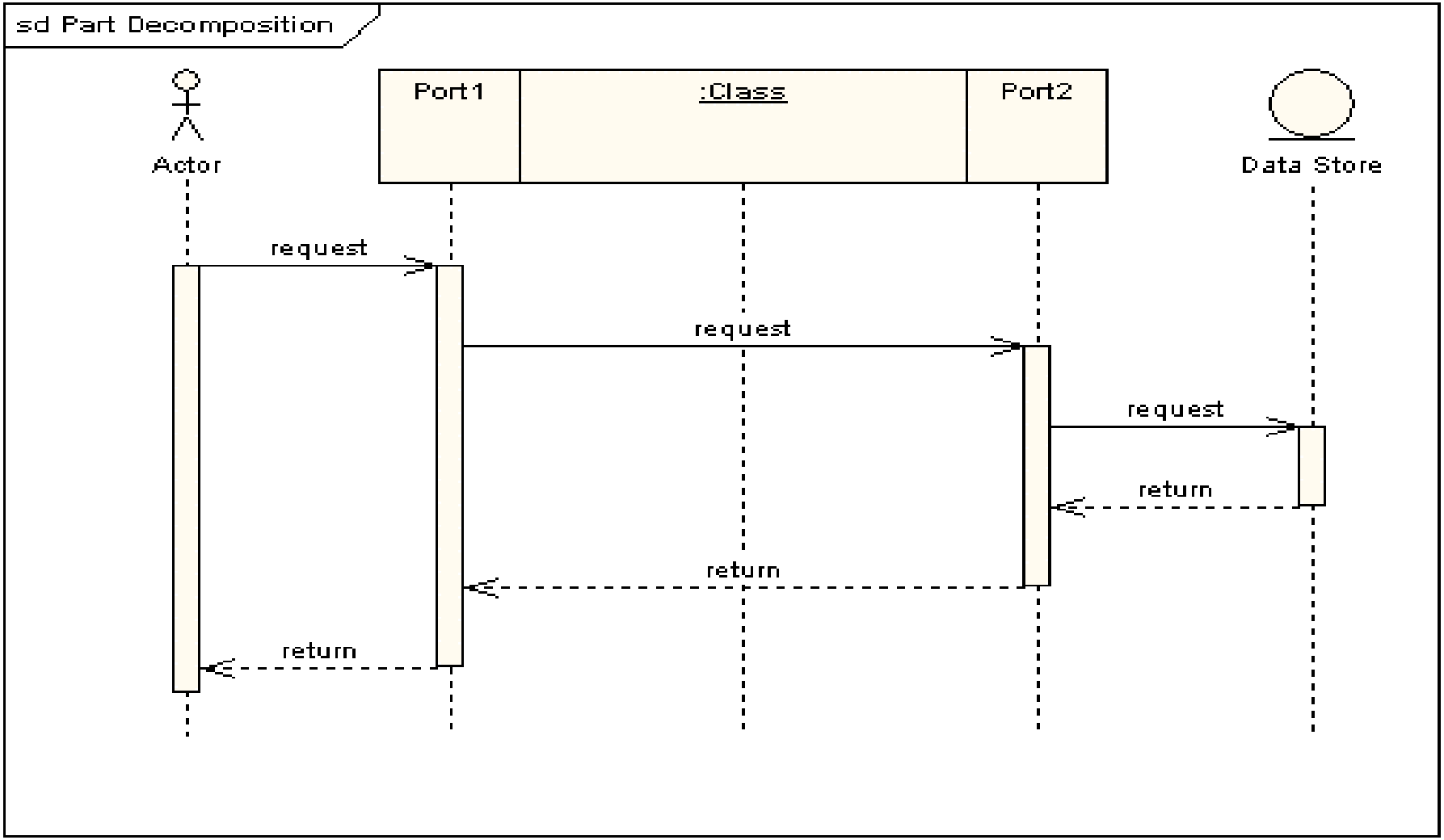
Brama (Gate)

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. **Diagramy sekwencji UML**

   https://sparxsystems.com/resources/tutorials/uml2/sequence-diagram.html

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

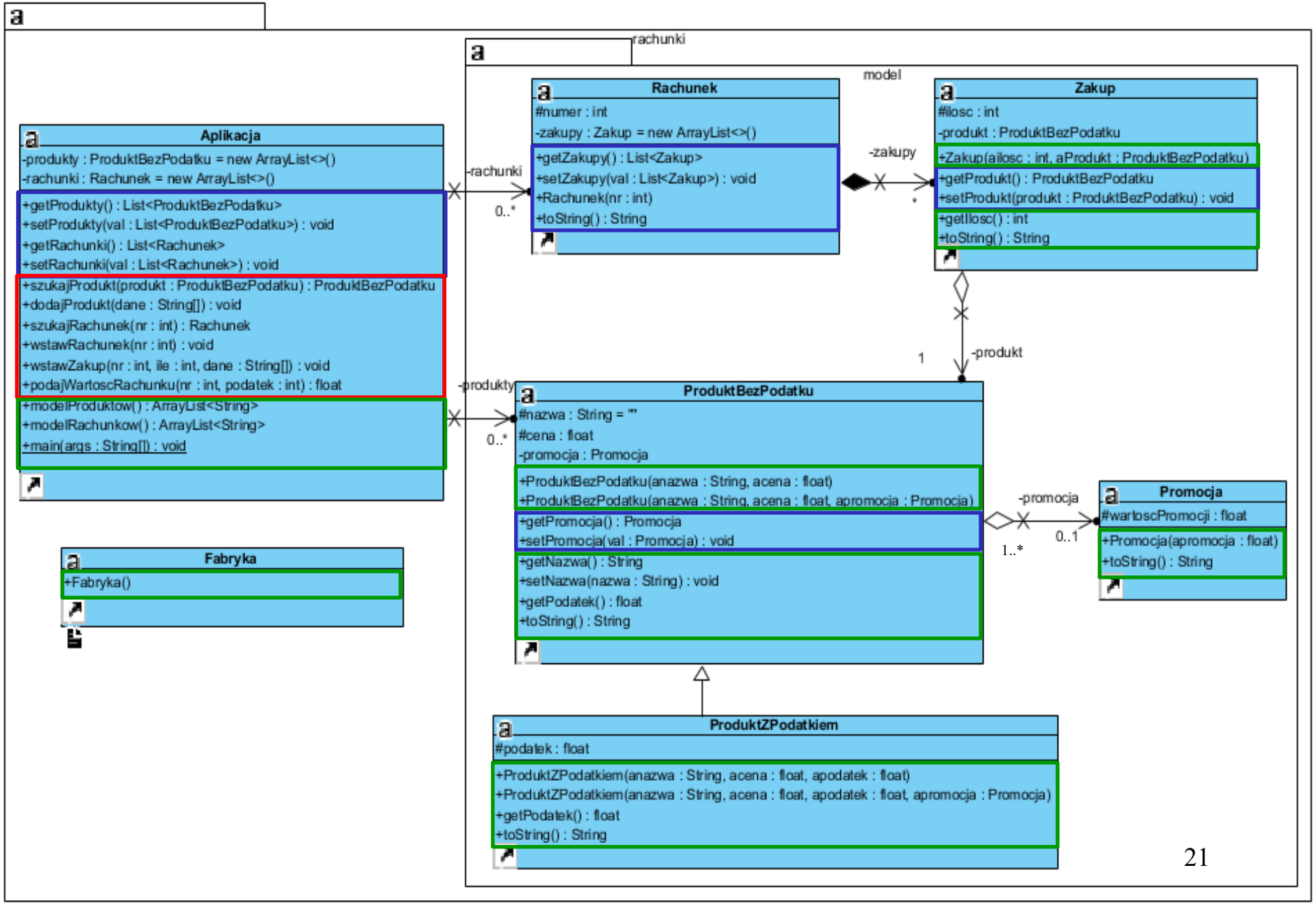
Iteracja 1

Projekt przypadku użycia

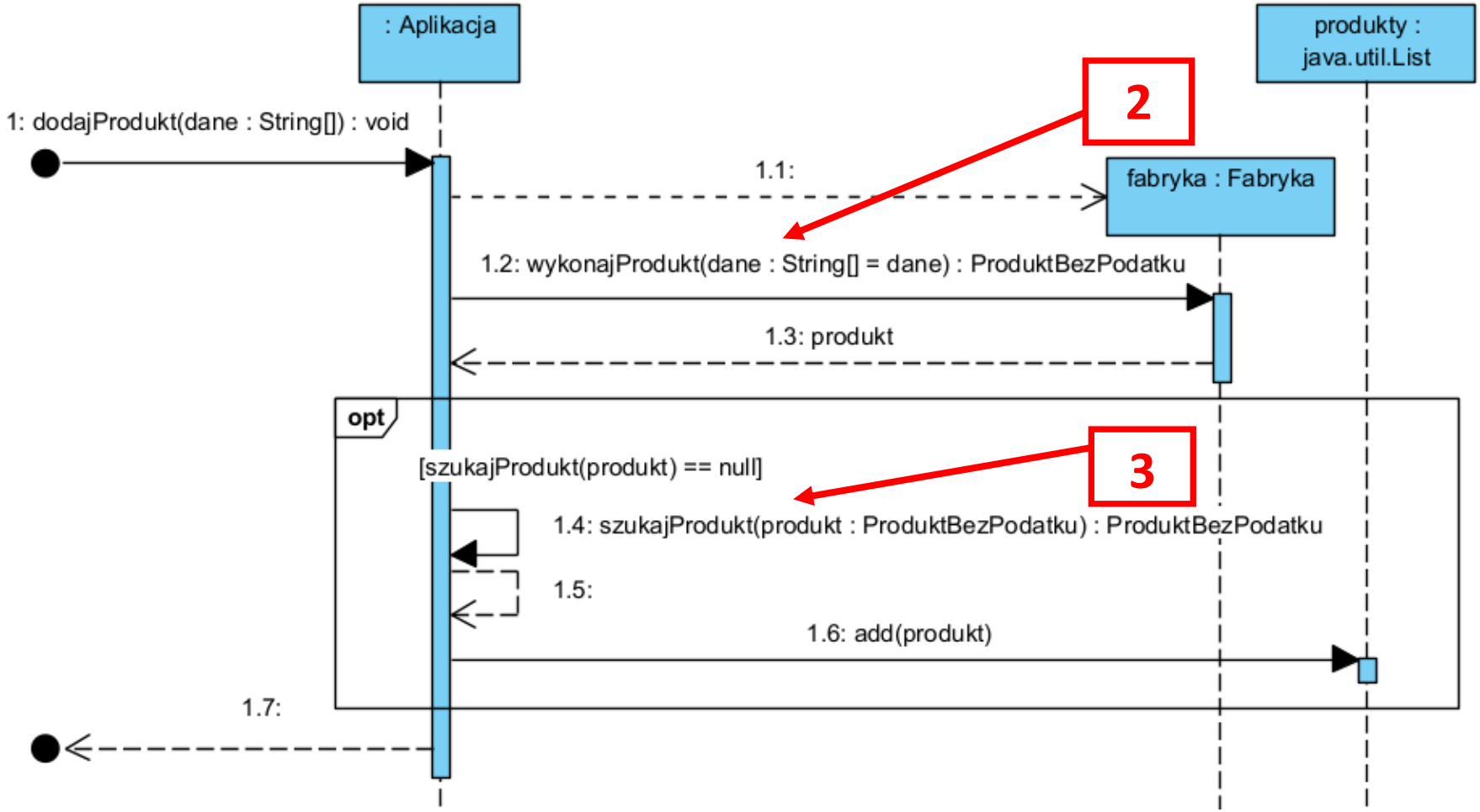
„ **Wstawianie nowego produktu**”

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

Projekt powiązań | Metody przypadków użycia | Decyzja projektowa

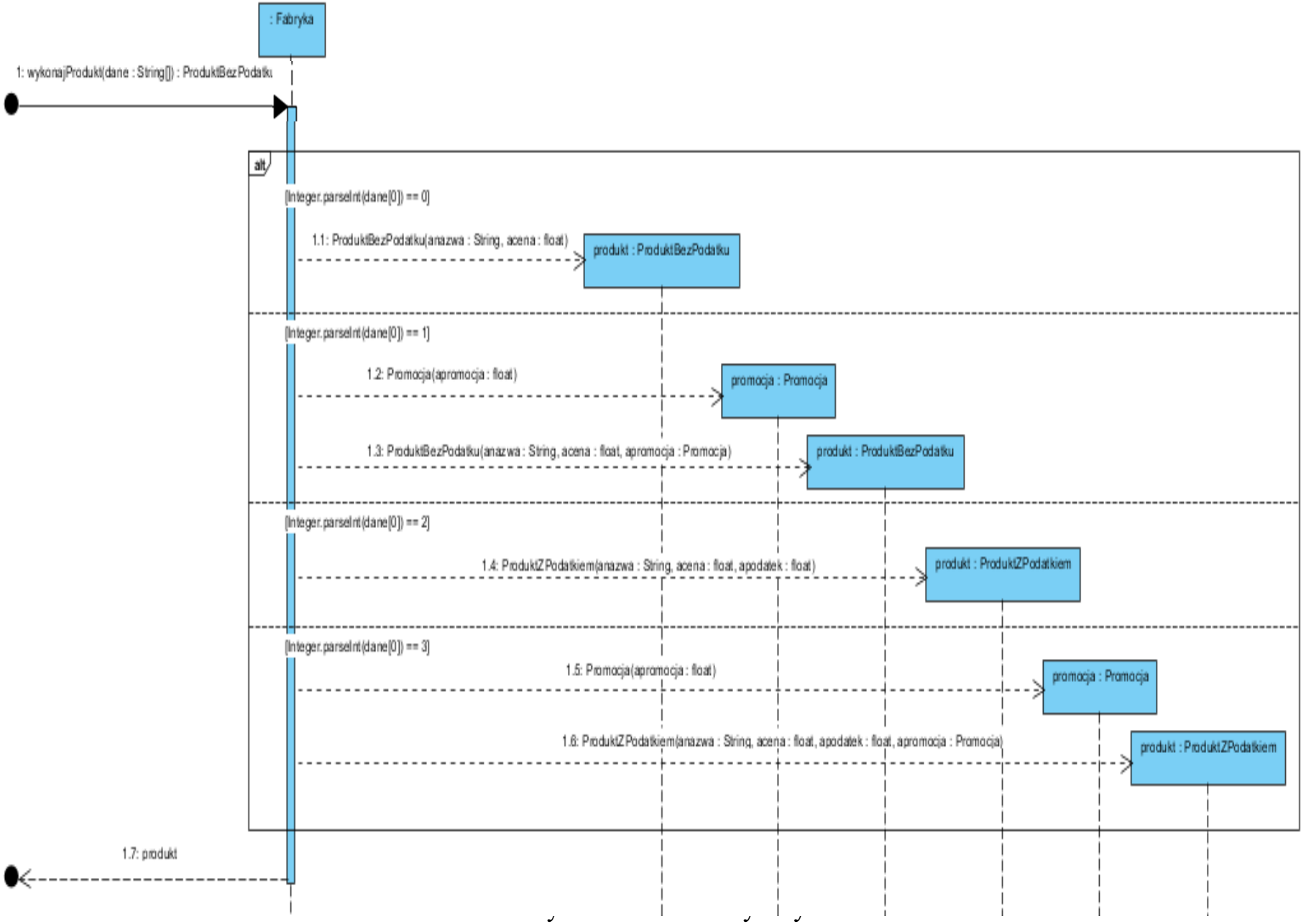


PU Wstawianie nowego produktu
(1) void dodajProdukt(String [] dane)



```
                                                    //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);
  }
```

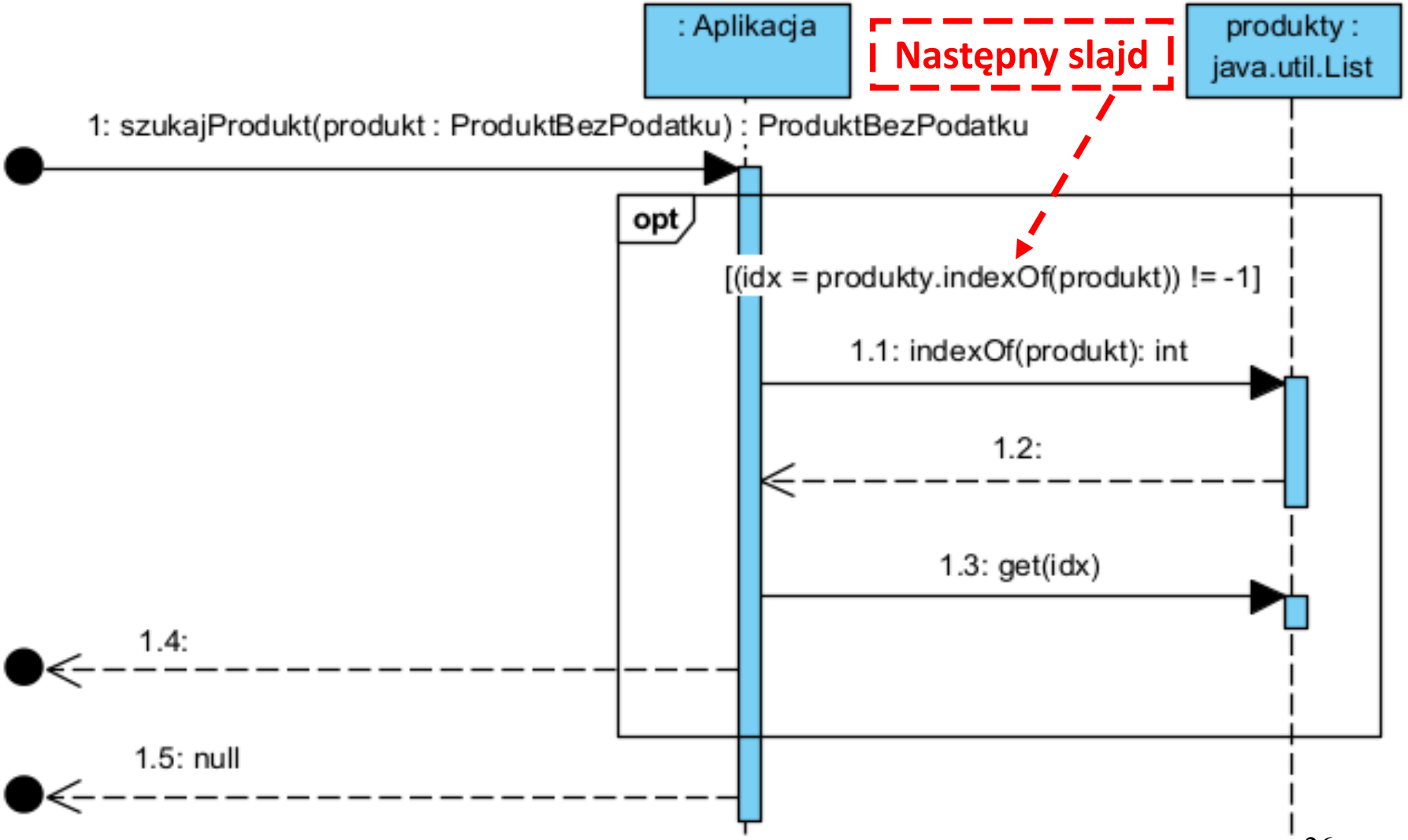
(2) ProduktBez Podatku wykonajProdukt(String dane[])



```
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

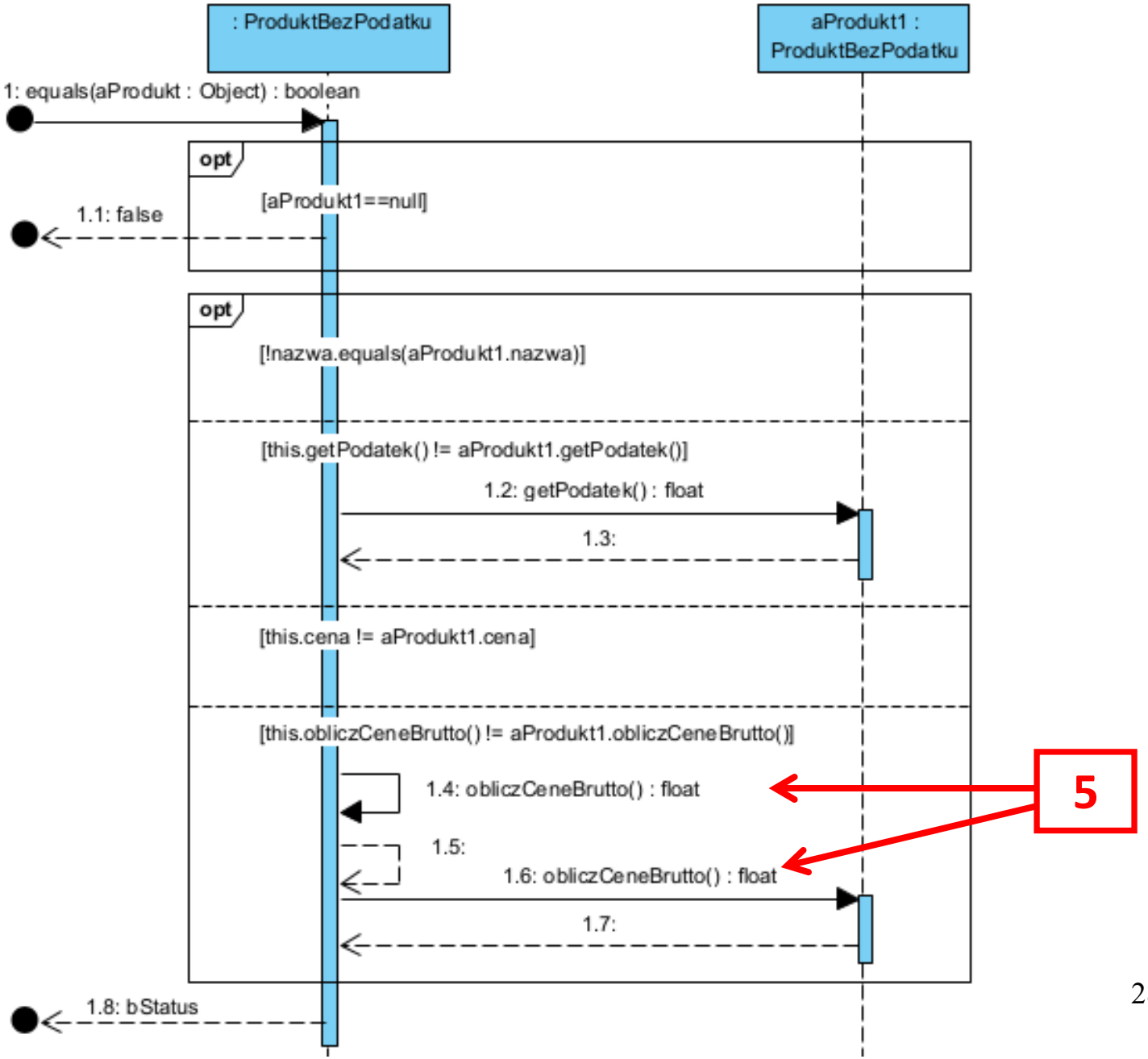
(3) ProduktBezPodatku szukajProdukt(ProduktBezPodatku produkt)



```
                                                    //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;
  }
```

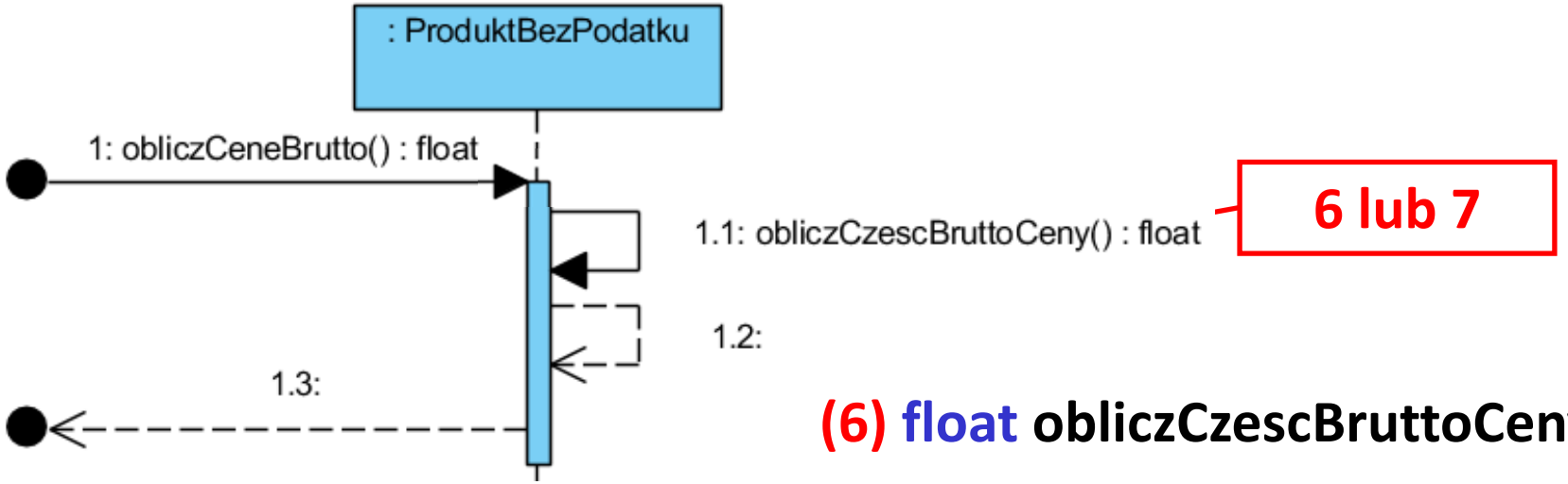
```
public int indexOf(Object o) {
      if (o == null) {
        for (int i = 0; i < size; i++)
          if (elementData[i]==null)
            return i;
      } else {
        for (int i = 0; i < size; i++)
          if (o.equals(elementData[i]))
            return i;  }
      return -1;  }
```

(4) boolean equals(Object aProdukt)

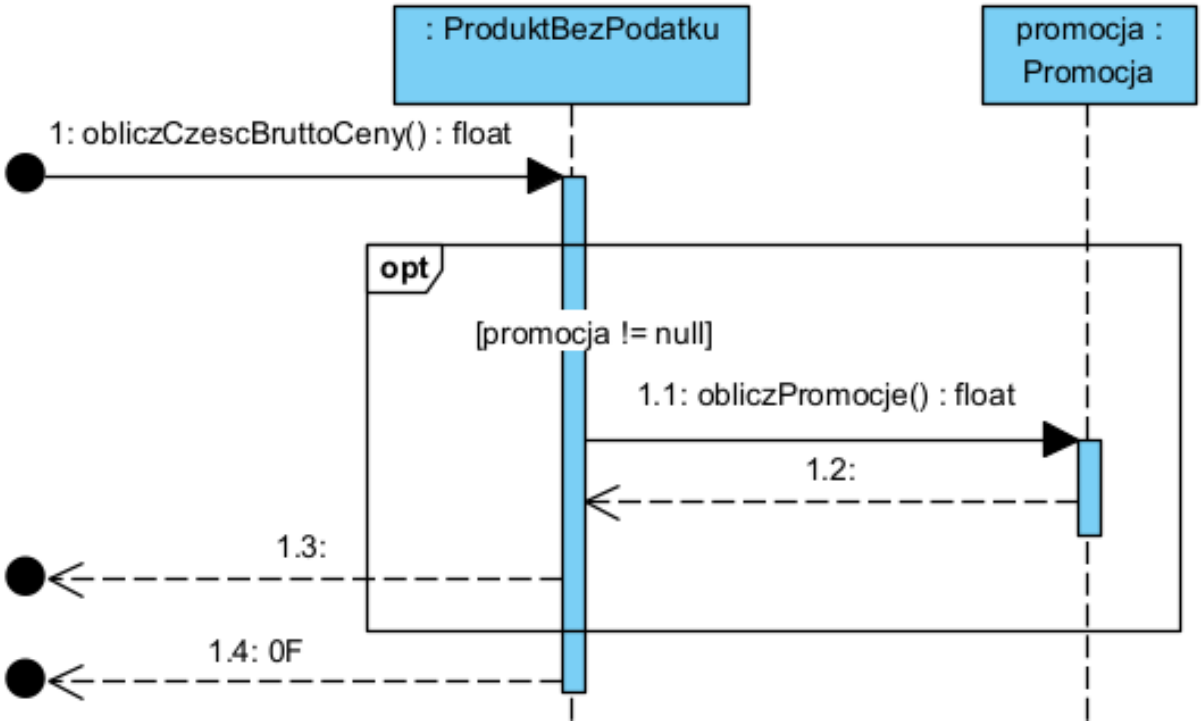


```
//class ProduktBezPodatku
public boolean equals (Object aProdukt)
 {
    ProduktBezPodatku aProdukt1=(ProduktBezPodatku)aProdukt;
    if ( aProdukt1 == null ) return false;
    boolean bStatus = true;
   if ( !nazwa.equals(aProdukt1.nazwa)) bStatus = false;
   else
    if (this.getPodatek()!=aProdukt1.getPodatek())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

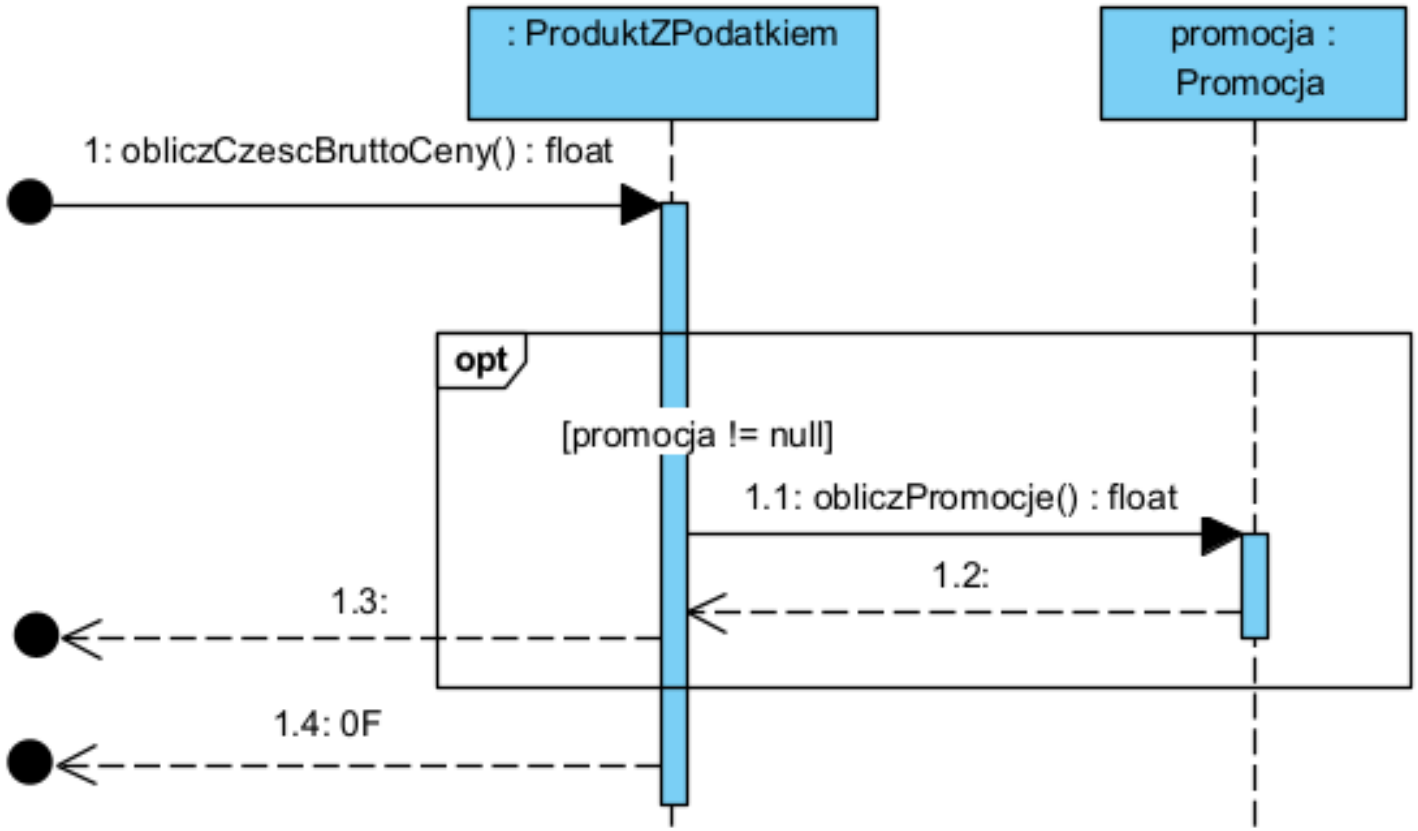
(5) float obliczCeneBrutto()



(6) float obliczCzescBruttoCeny()



(7) float obliczCzescBruttoCeny()

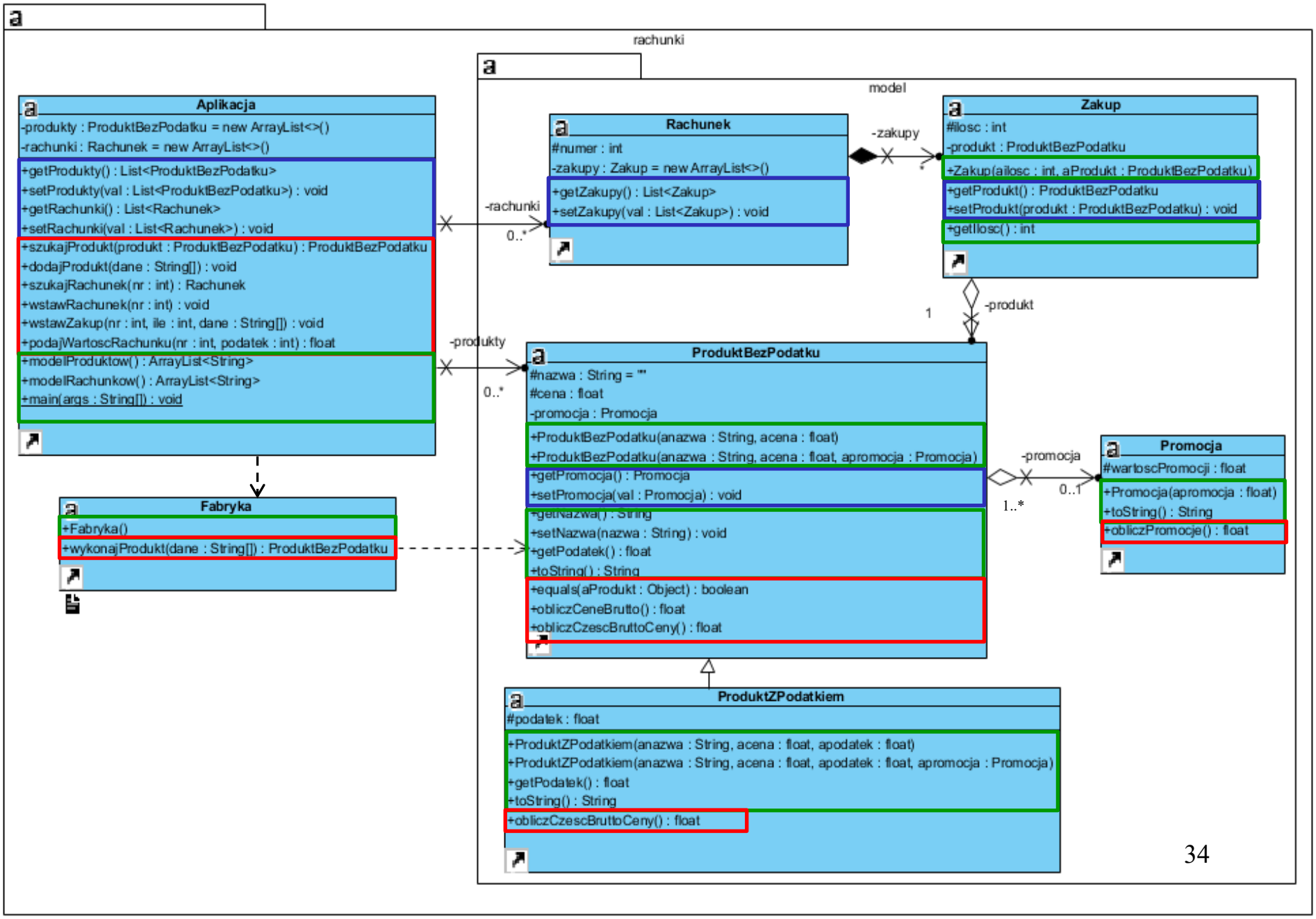


```
//class ProduktBezPodatku
public float obliczCeneBrutto ()
  {
      return cena + obliczCzescBruttoCeny();
  }

 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



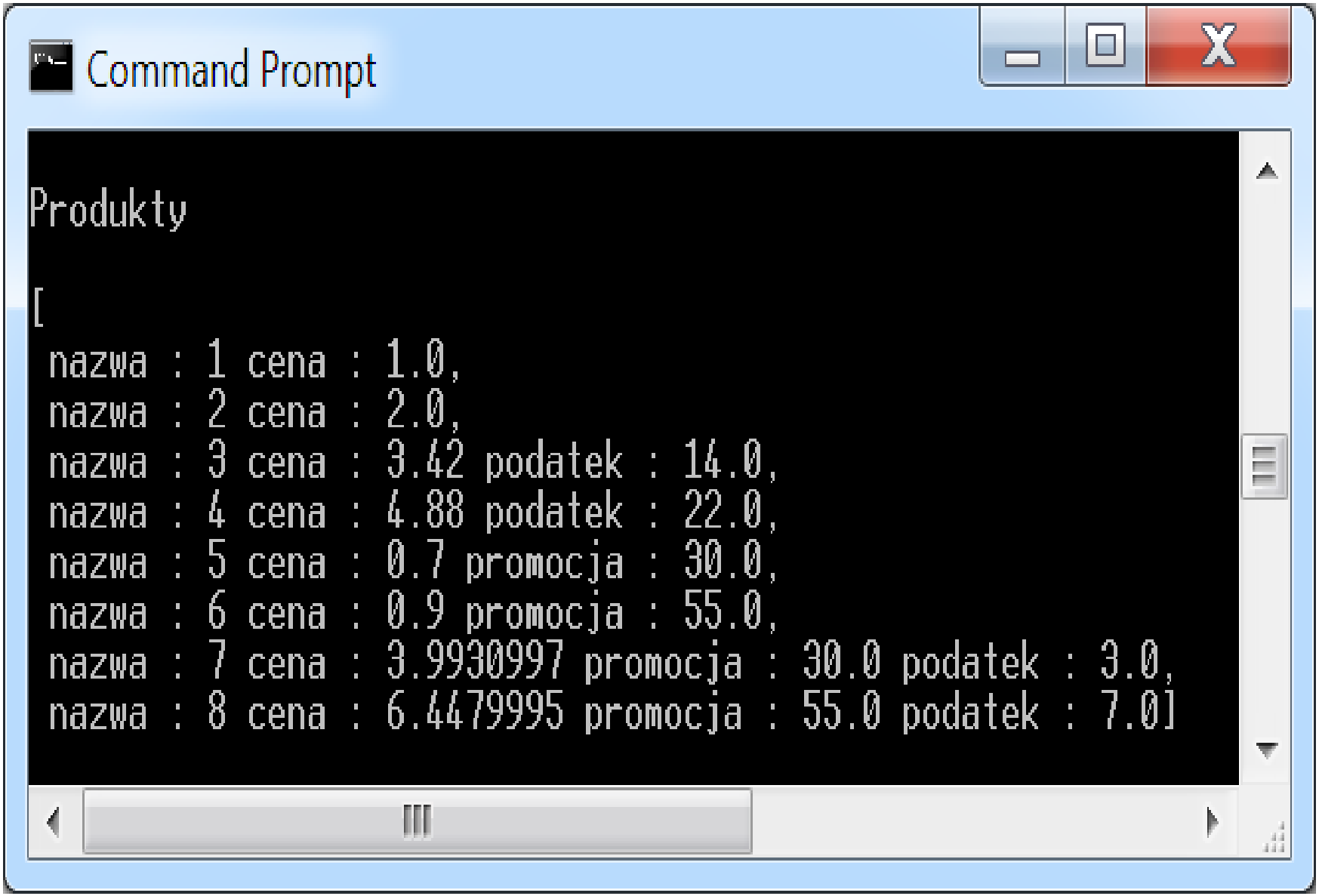
```
public ArrayList<String> modelProduktow()          //Aplikacja
 {     ArrayList<String> modelProduktow = new ArrayList();
       for (ProduktBezPodatku produkt : produkty)
          modelProduktow.add("\n" + produkt.toString());
       return modelProduktow;
   }
```

```
 @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();
 }
```

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

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

```
public static void main(String args[])   //metoda main – testowanie ręczne metod
{ Aplikacja app=new Aplikacja();  //klasy Aplikacja hermetyzującej logikę biznesową
  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]

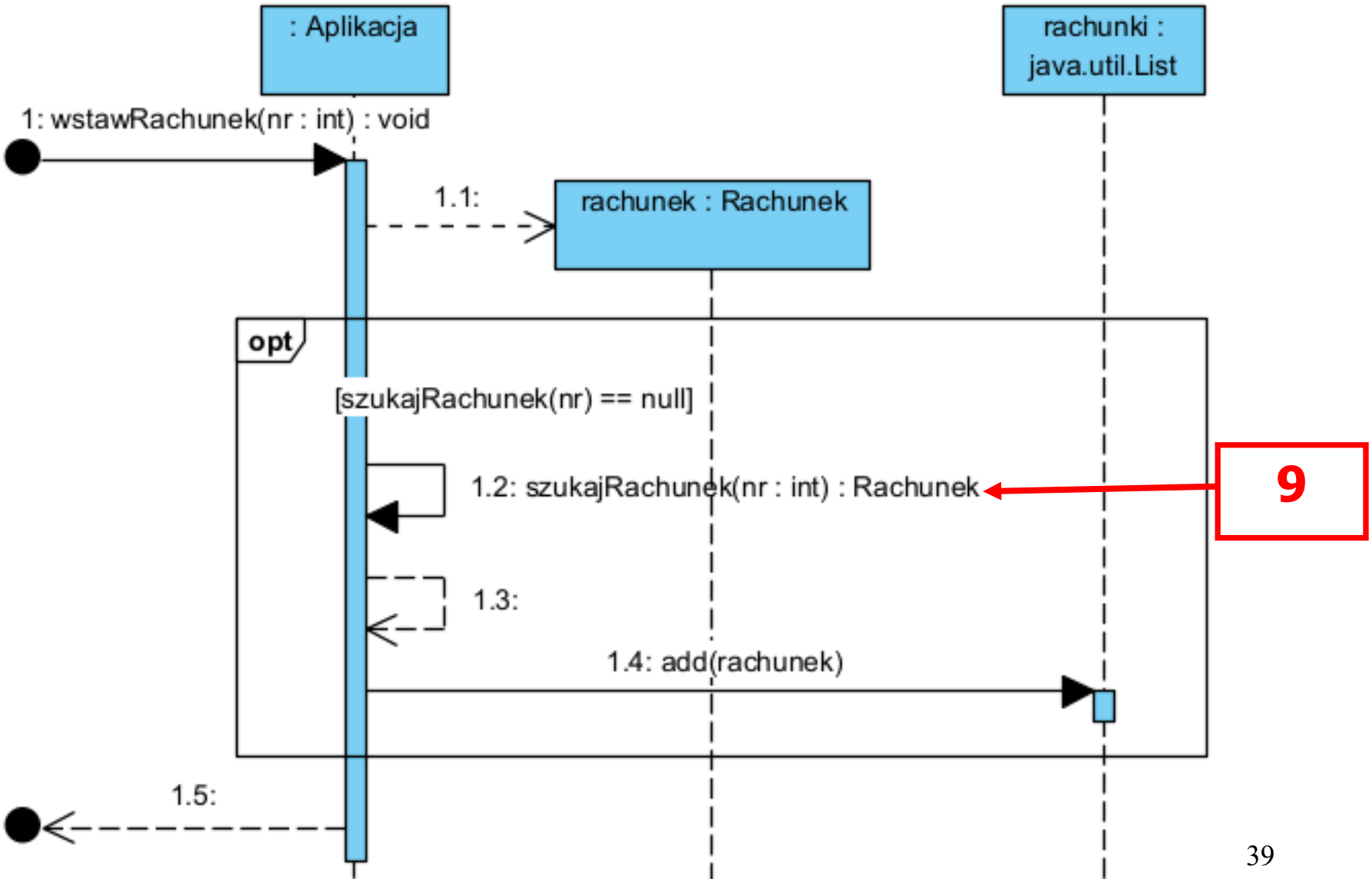
Iteracja 2

Projekt przypadku użycia

„ **Dodawanie rachunku**”

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

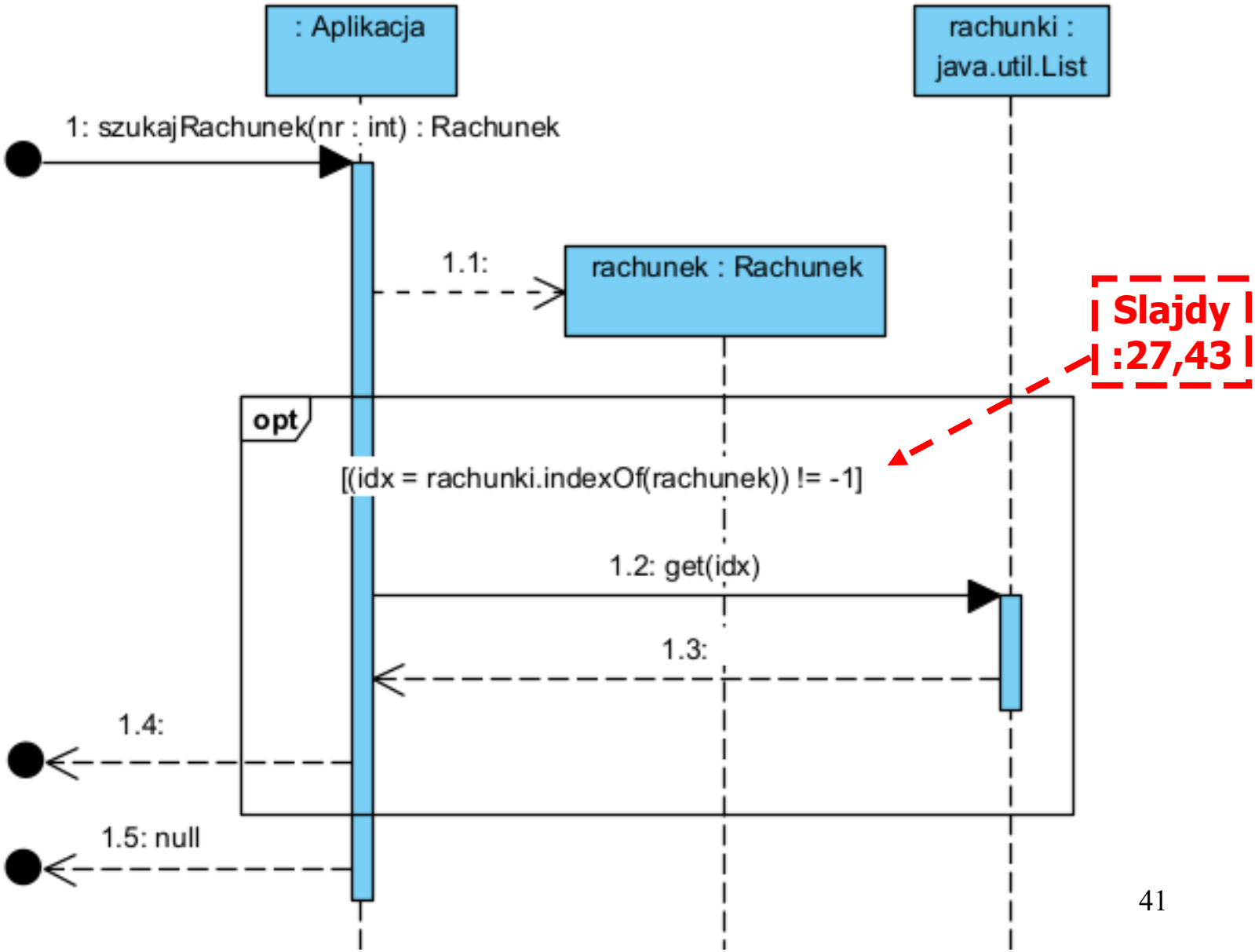
PU Dodawanie rachunku
(8) void wstawRachunek(int nr)



```
                                                    //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

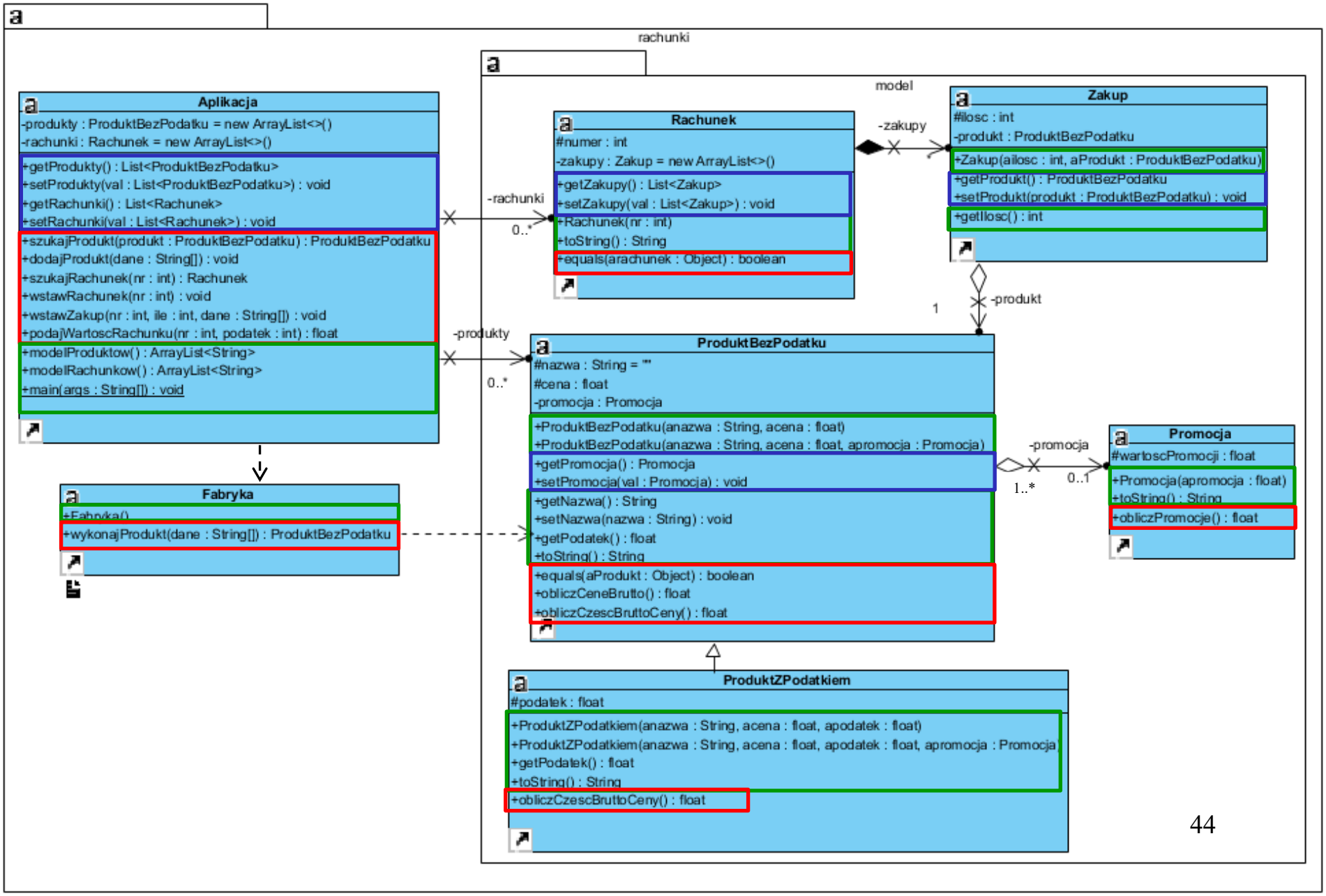
(9) Rachunek szukajRachunek(int nr)



```
                                                    //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

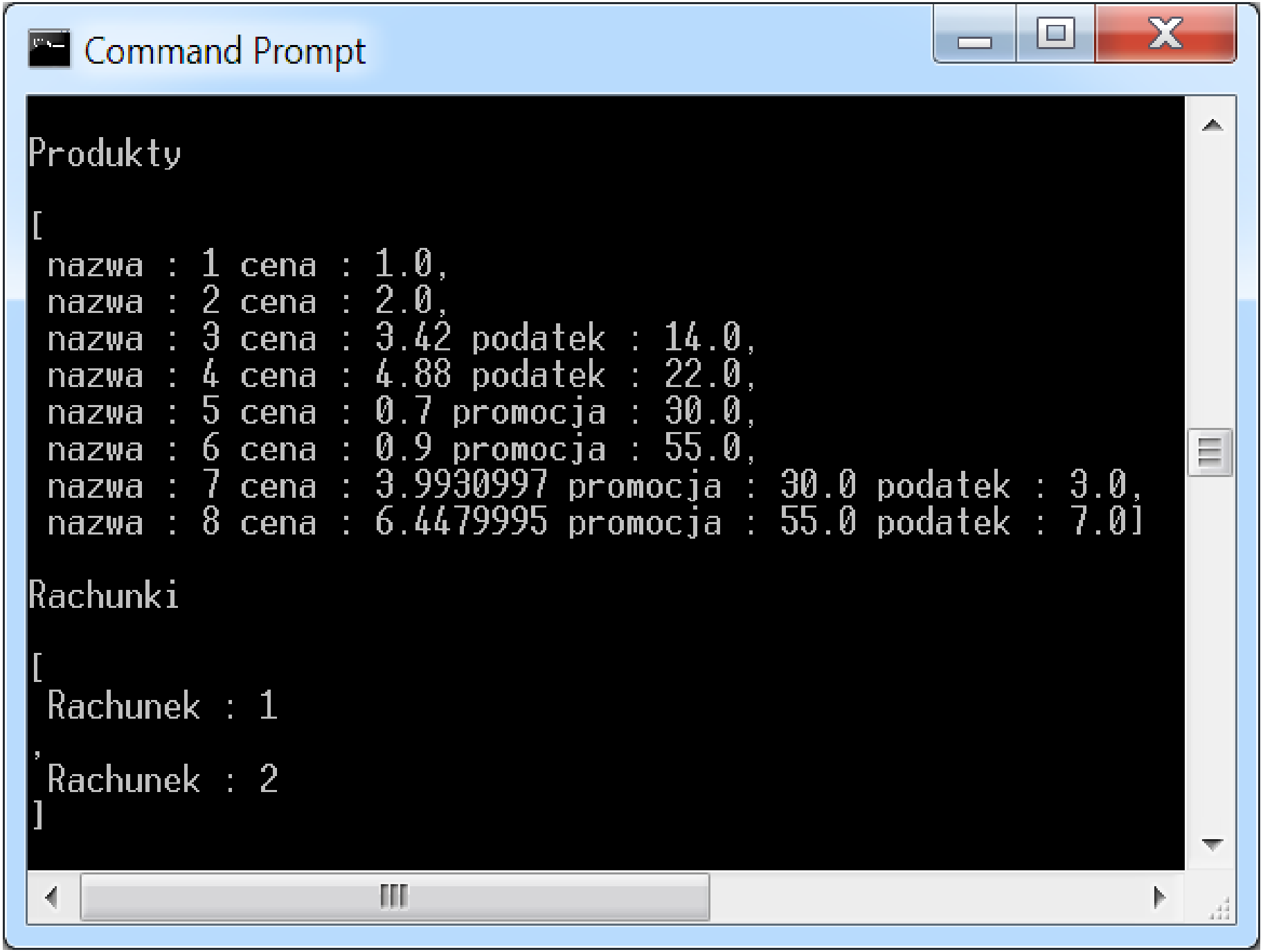


```
//Decyzje na poziomie tworzenia kodu
//Aplikacja
public ArrayList<String> modelRachunkow() {
      ArrayList<String> modelRachunkow = new ArrayList();
      for (Rachunek rachunek : rachunki) {
         modelRachunkow.add("\n" + rachunek.toString());
      }
      return modelRachunkow;
   }
```

```
//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(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

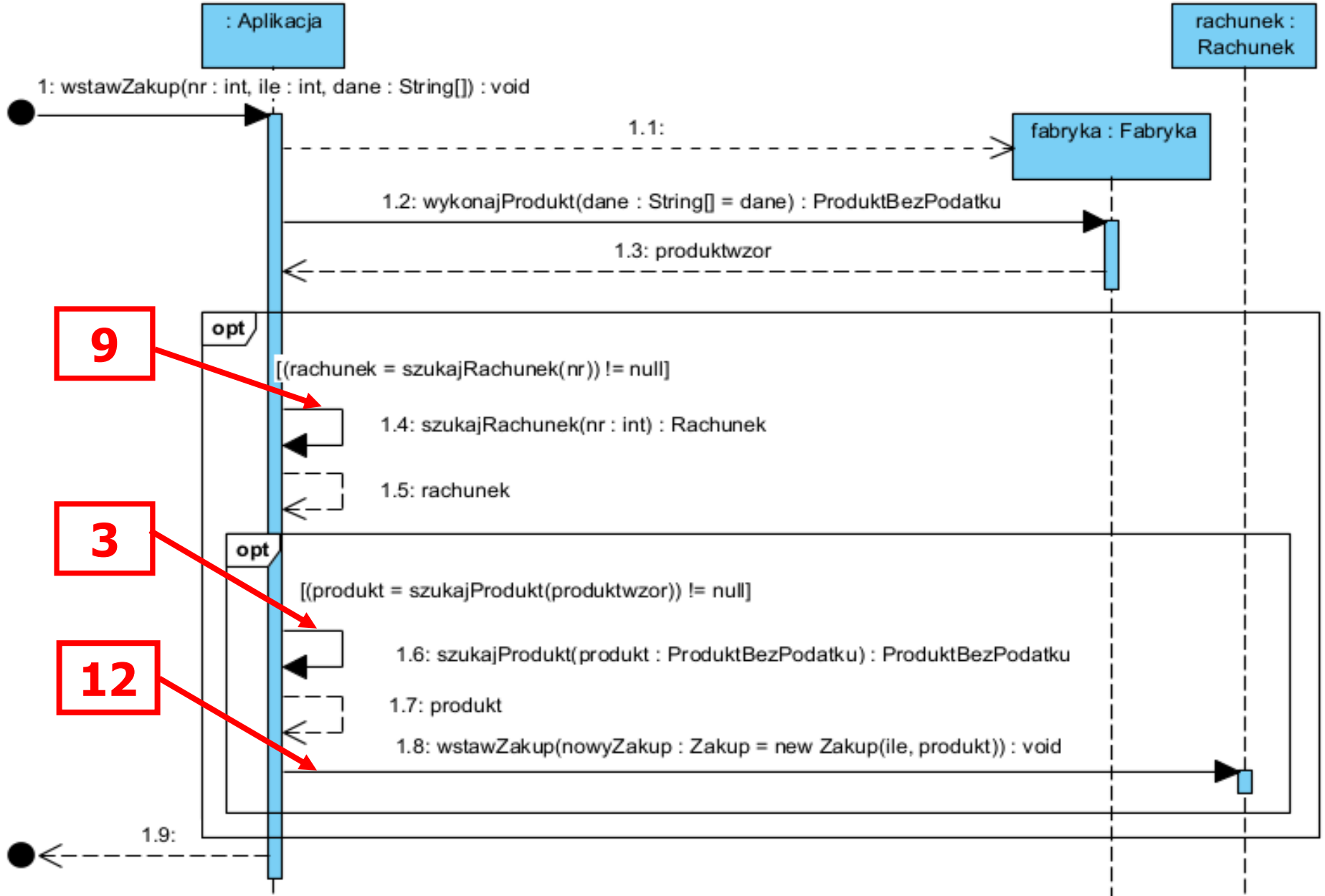
Projekt przypadku użycia

„**Dodawanie zakupu**”

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

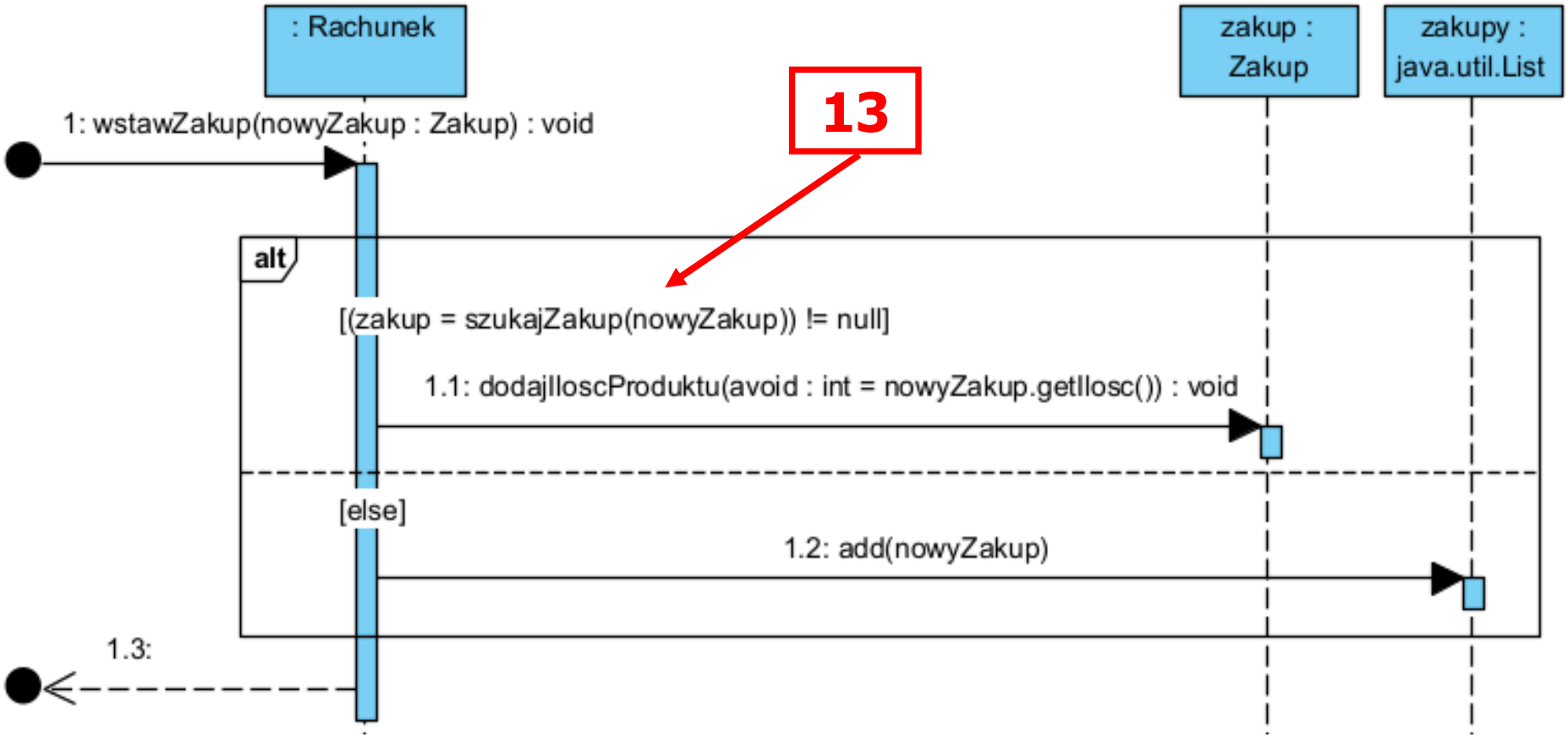
PU Dodawanie zakupu

(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));
 }
```

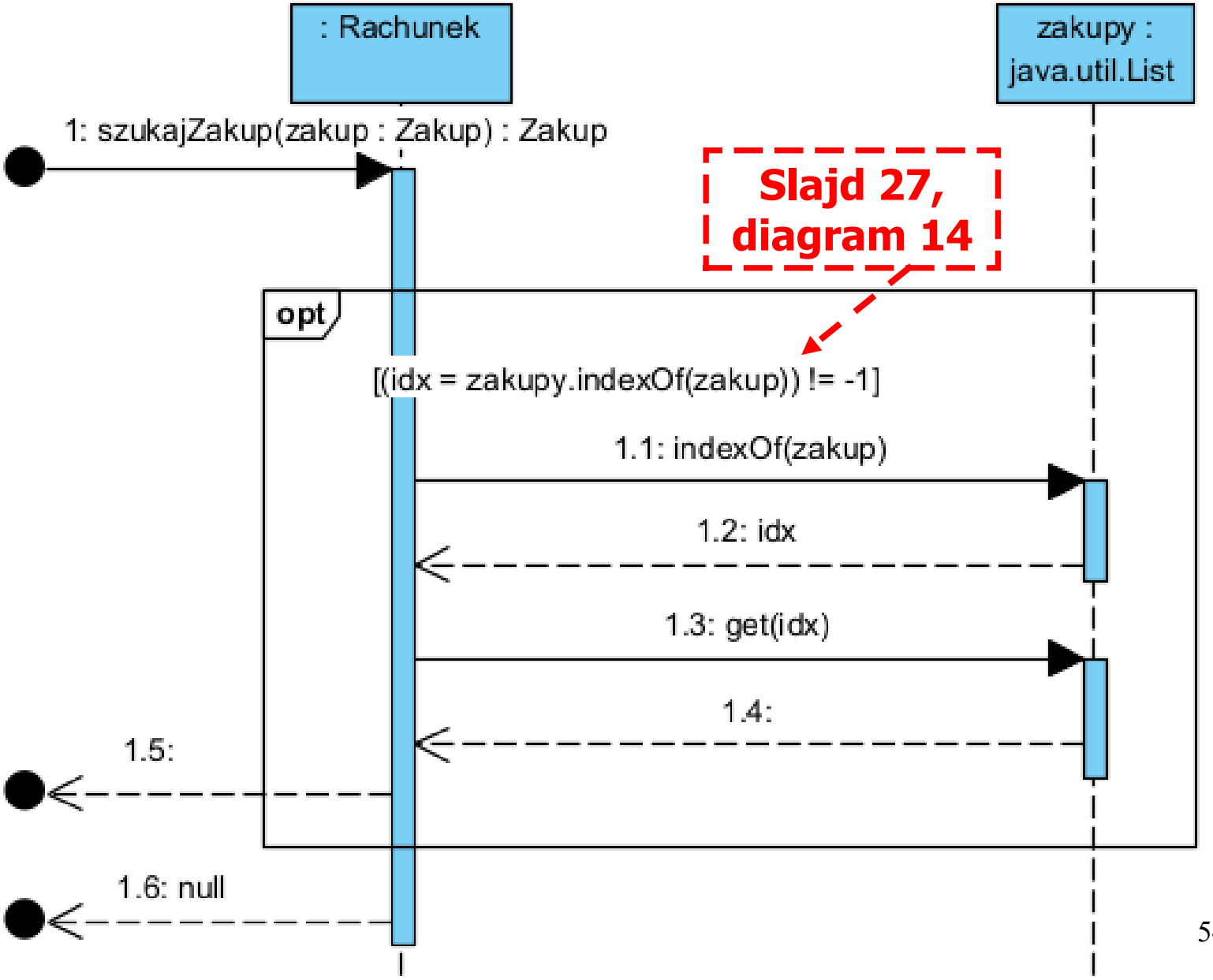
(12) void wstawZakup(Zakup azakup)



```
                                                    //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);
  }
```

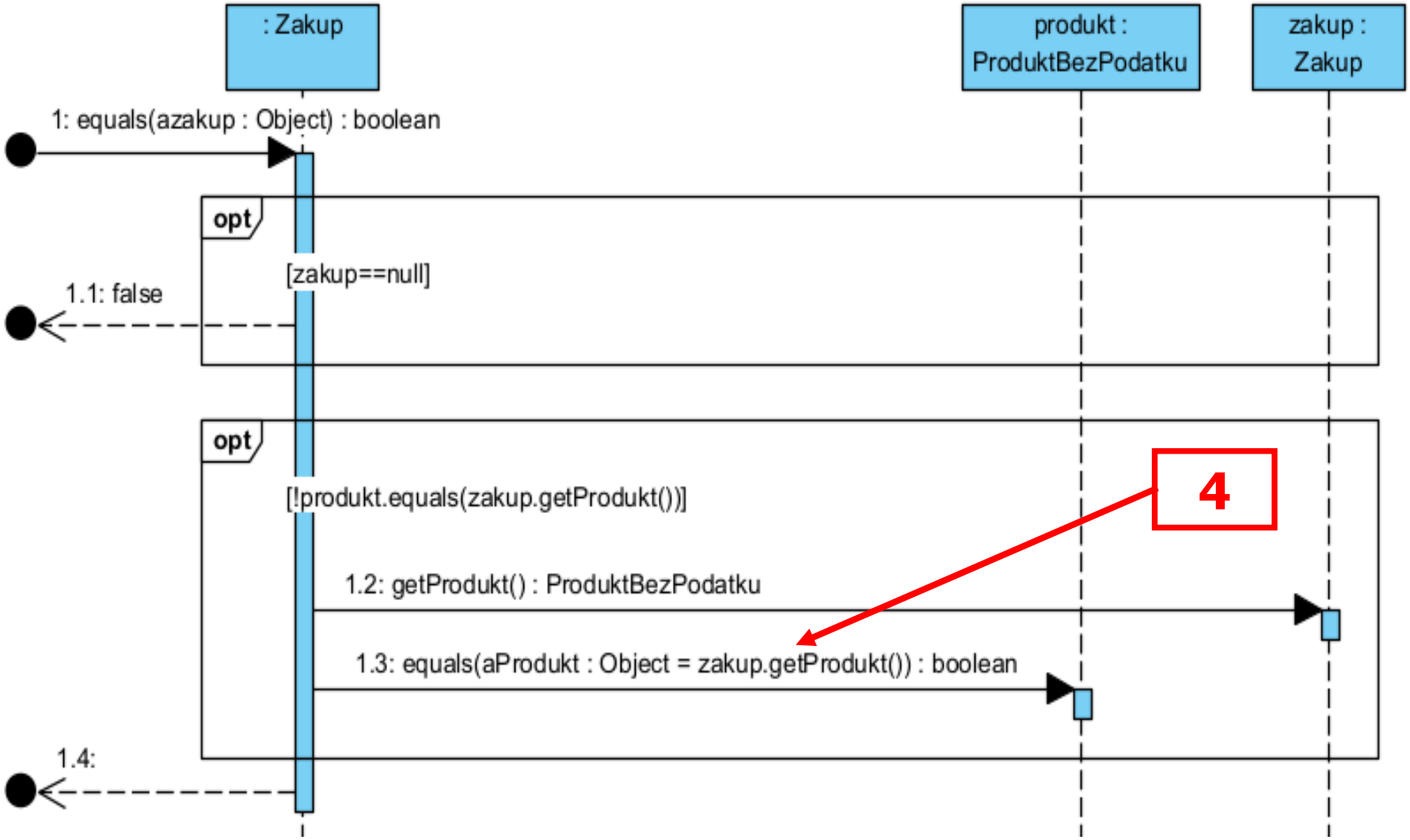
```
                                                    //Zakup

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)

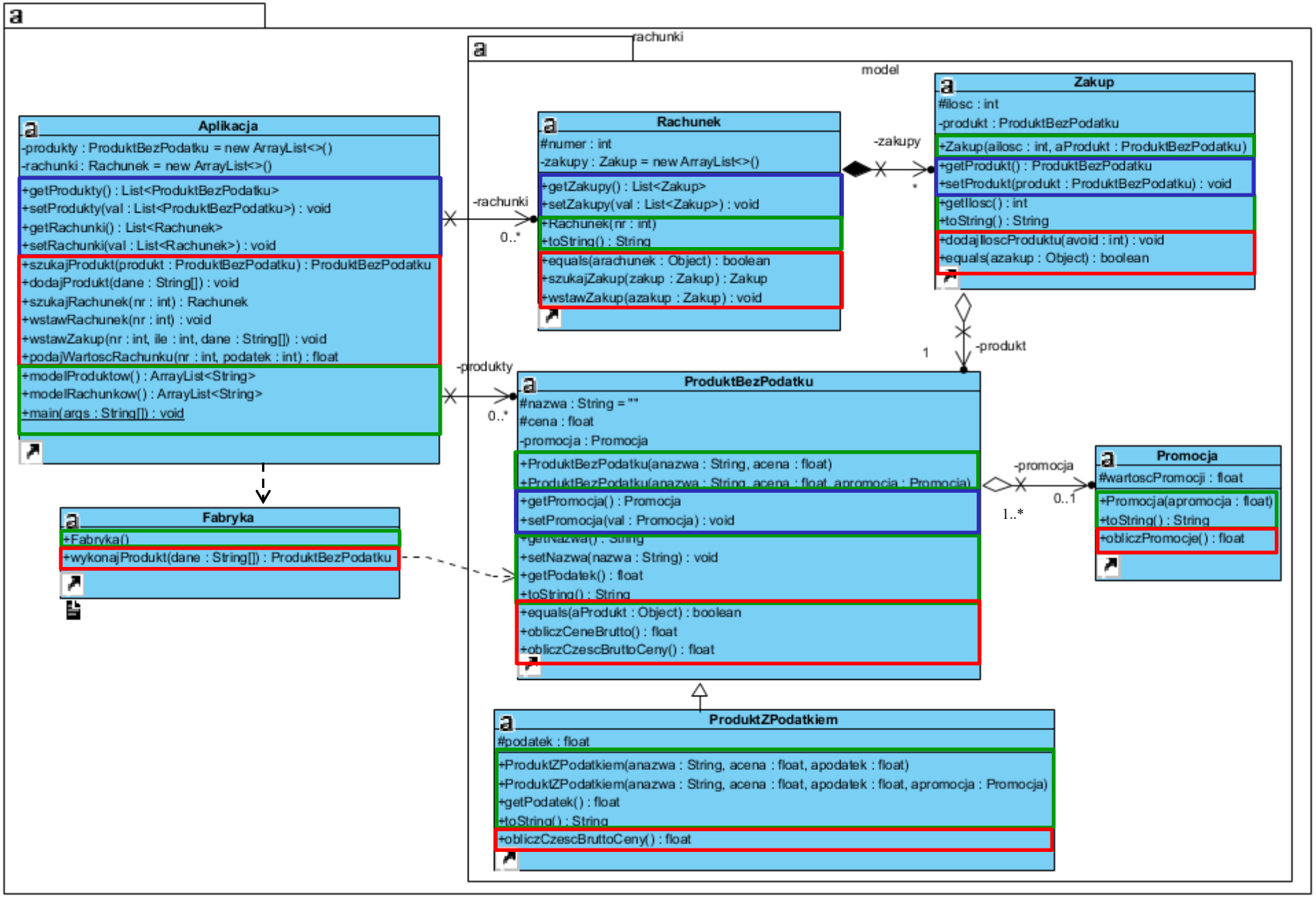


```
                                                        //Zakup

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
  }
```

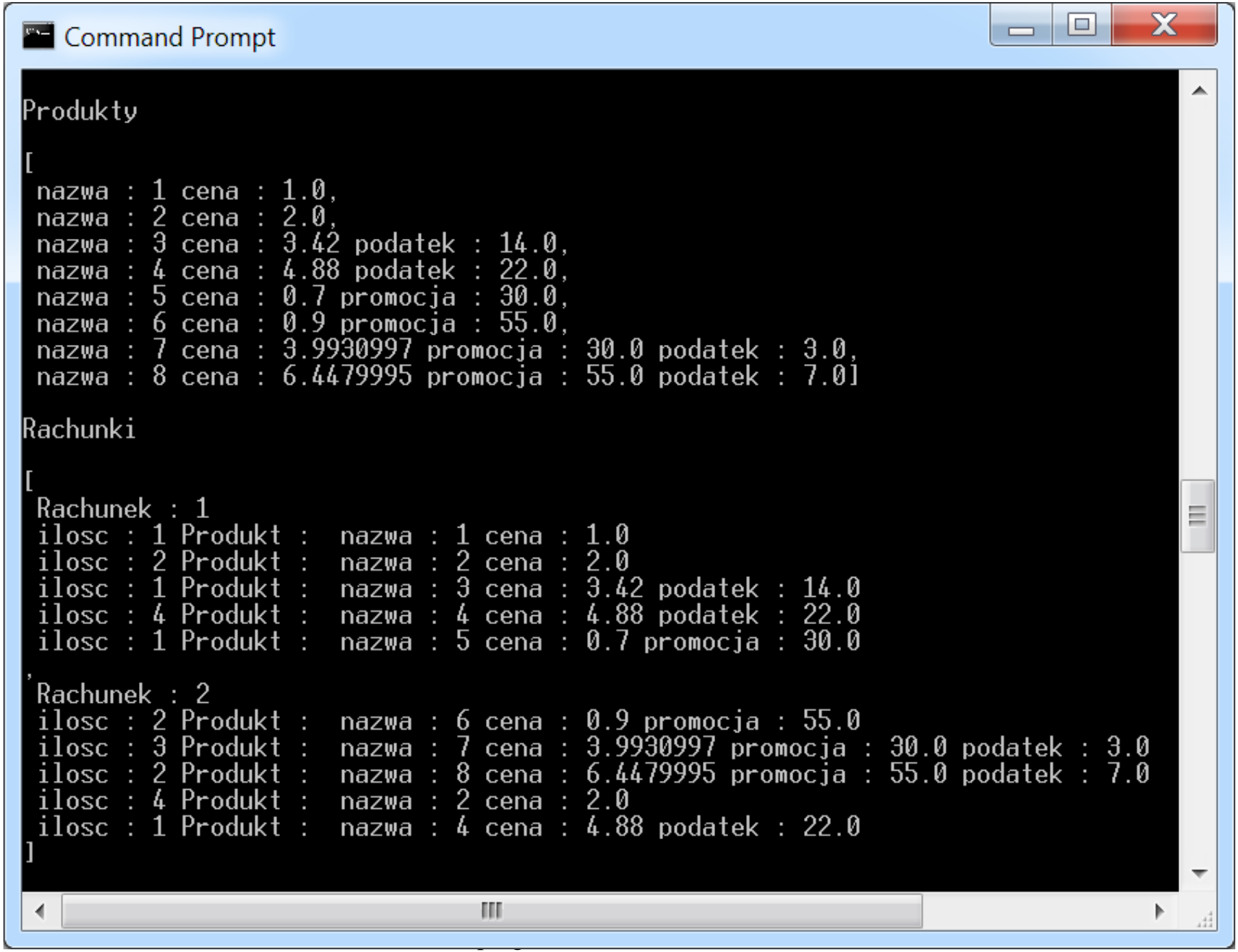
Projekt powiązań

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

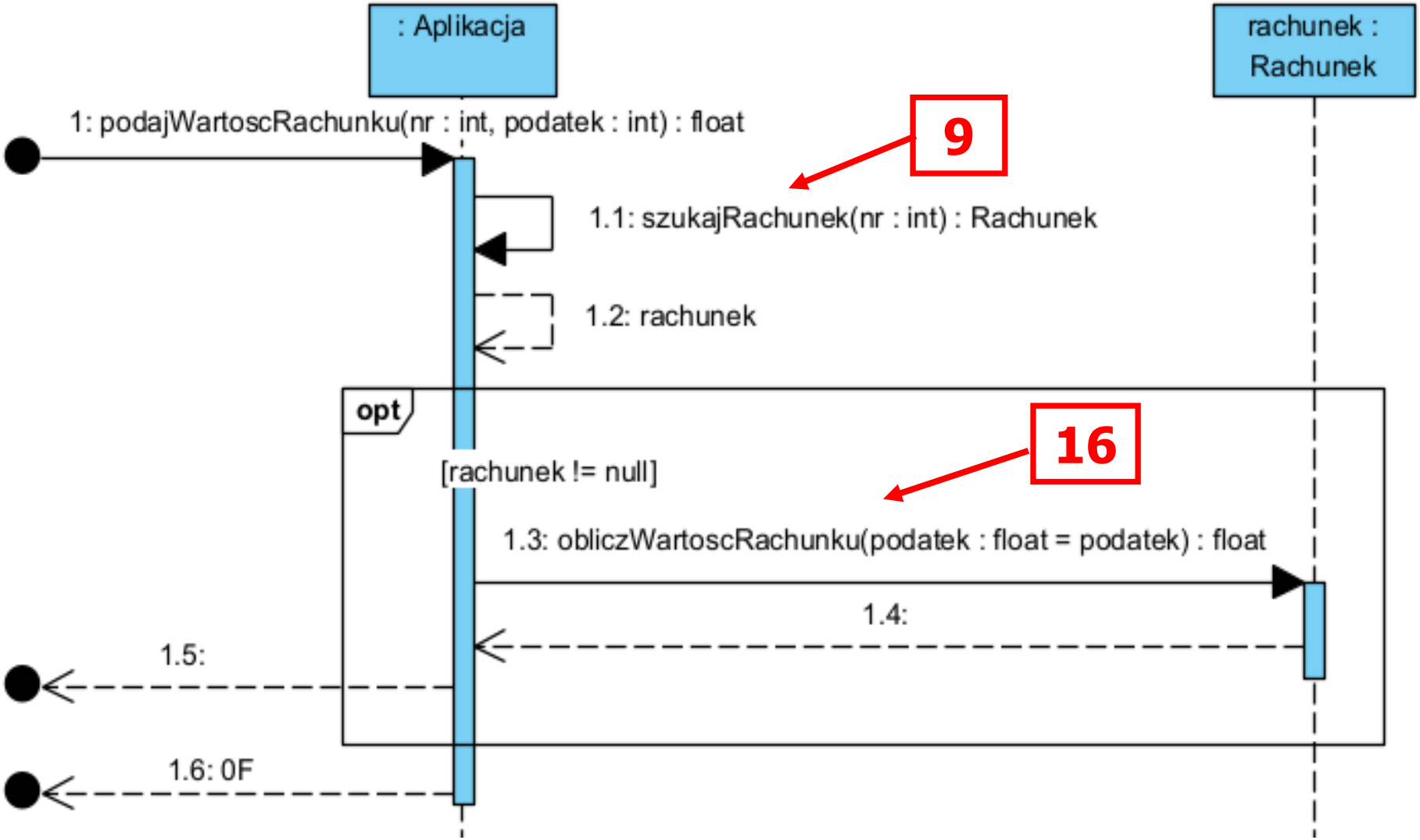
Projekt przypadku użycia

„**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.

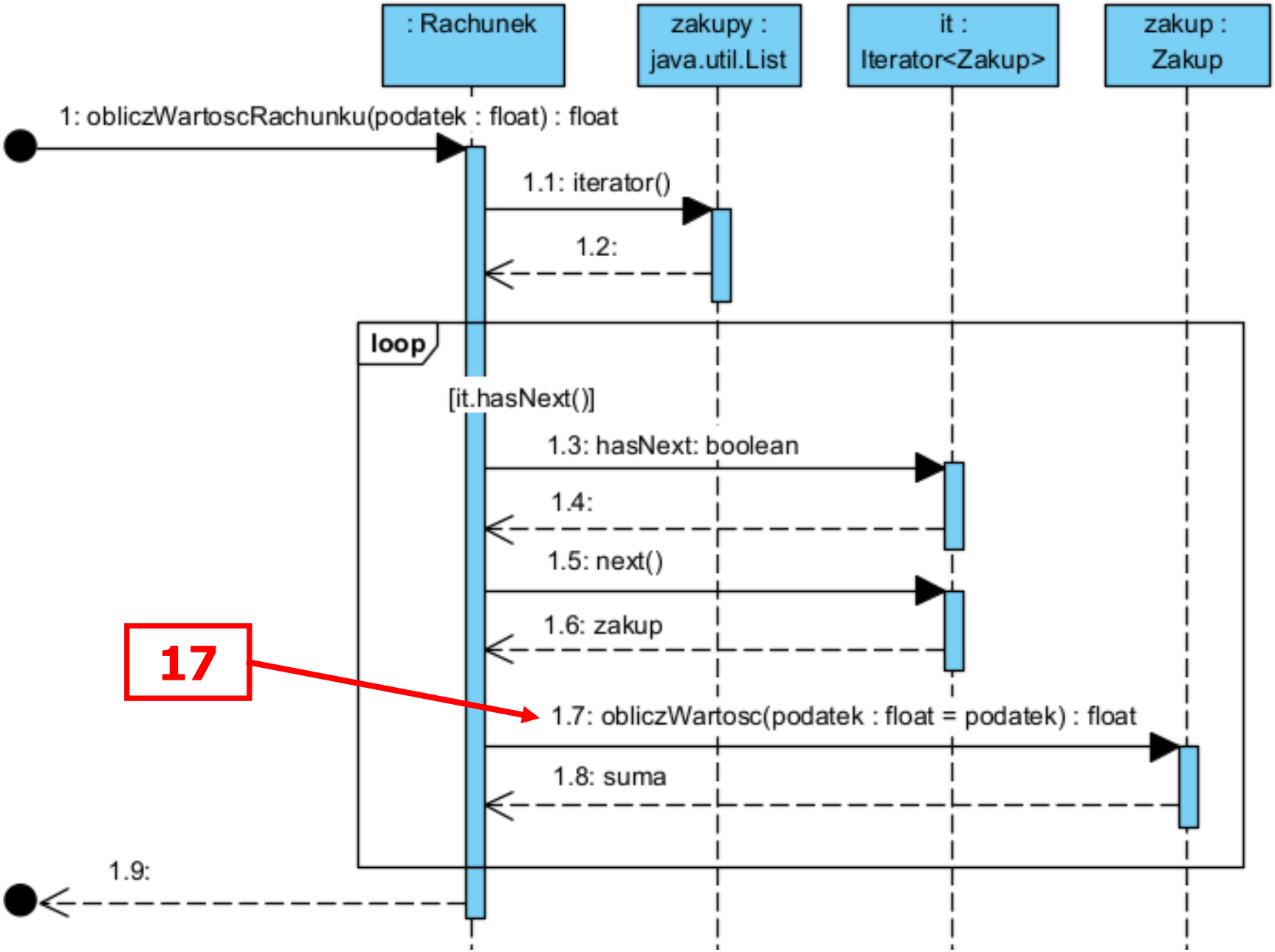
PU Obliczanie wartosci rachunku

(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;
 }
```

(16.1) float obliczWartoscRachunku(int podatek) – pętla while

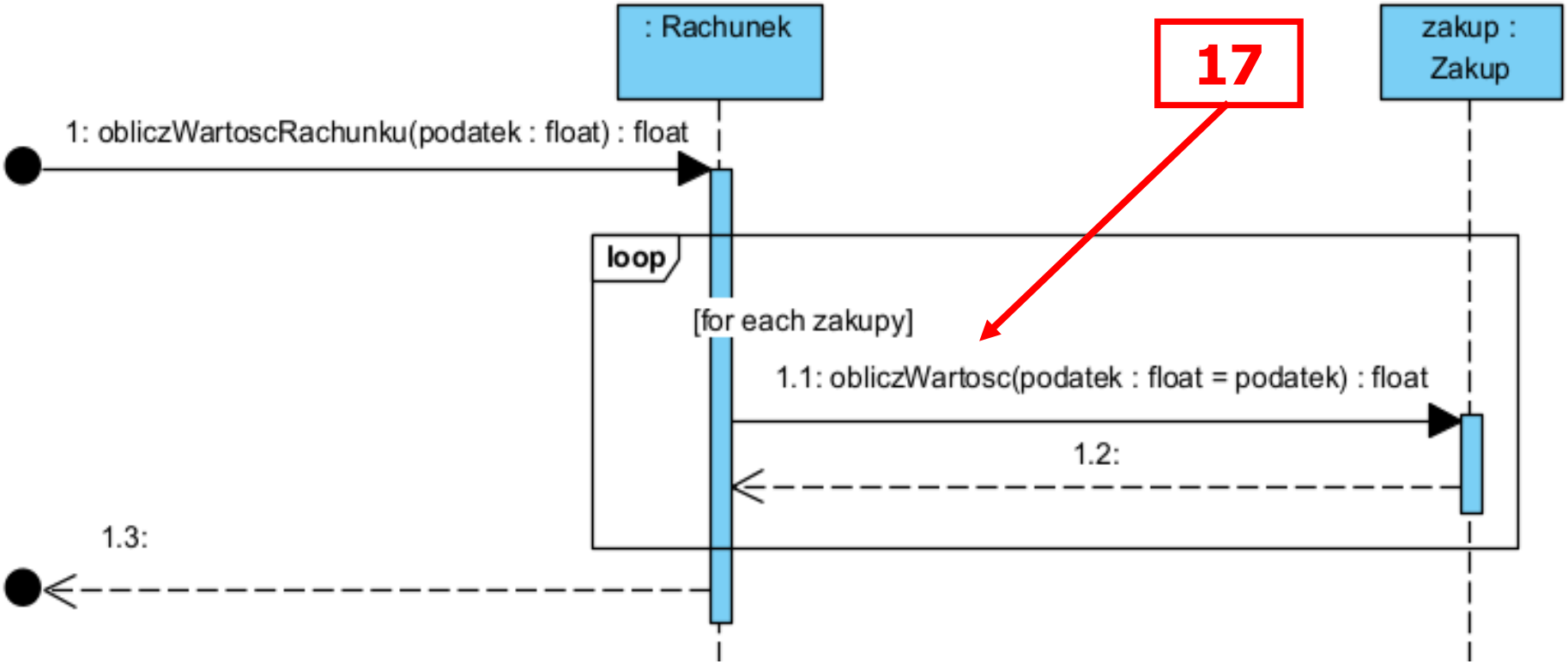


//Rachunek

1-a wersja z pętlą while

```
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;
  }
```

(16.2) float obliczWartoscRachunku(int podatek) - pętla for

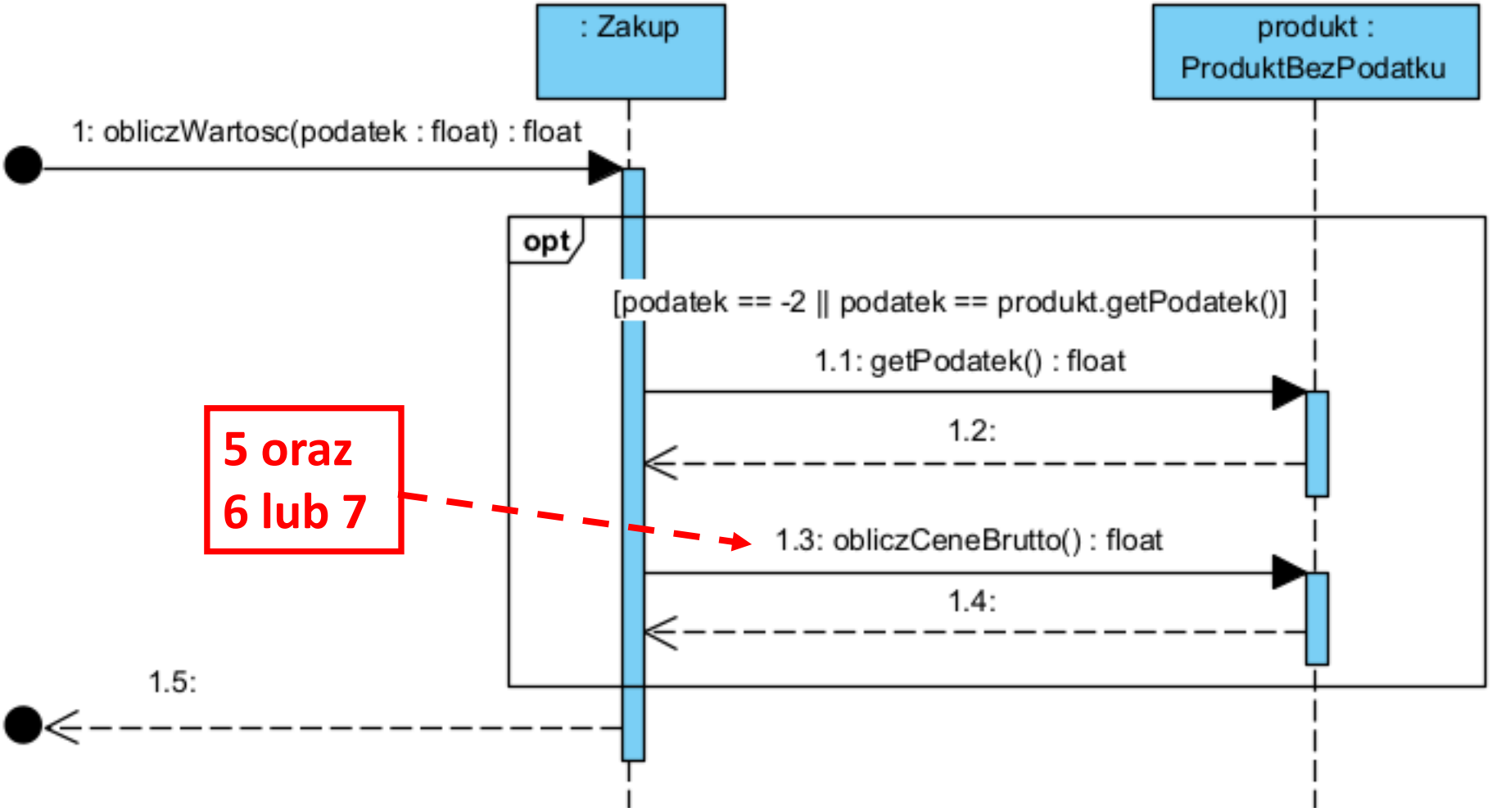


//Rachunek

2-a wersja z pętlą for

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

public float obliczWartoscRachunku (int podatek)
 {
   float suma=0;
   for (Zakup zakup:zakupy) {
      suma += zakup.obliczWartosc(podatek);
    }
   return suma;
  }
```

(17) float obliczWartosc(int podatek)

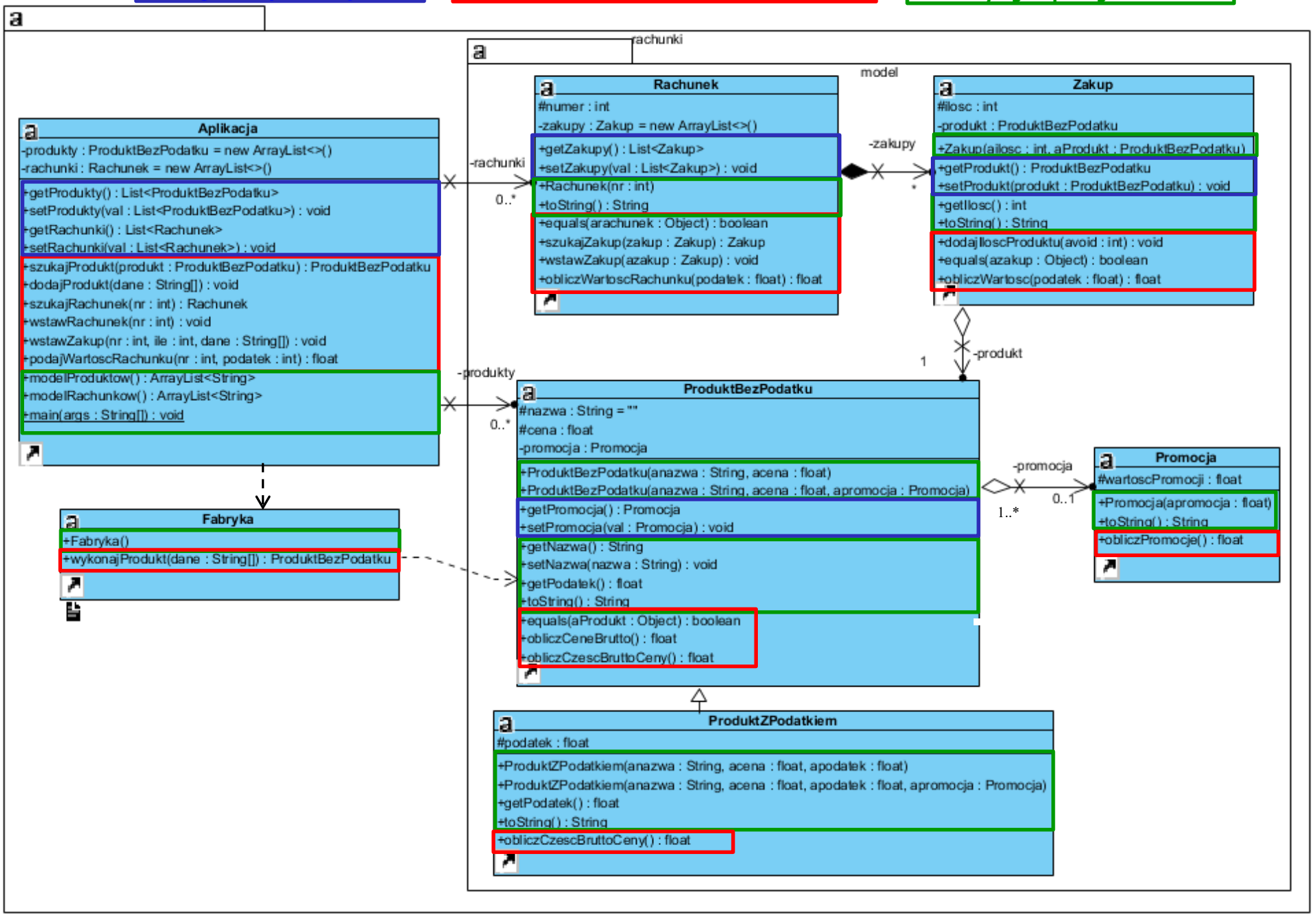


```
                                                          //Zakup


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

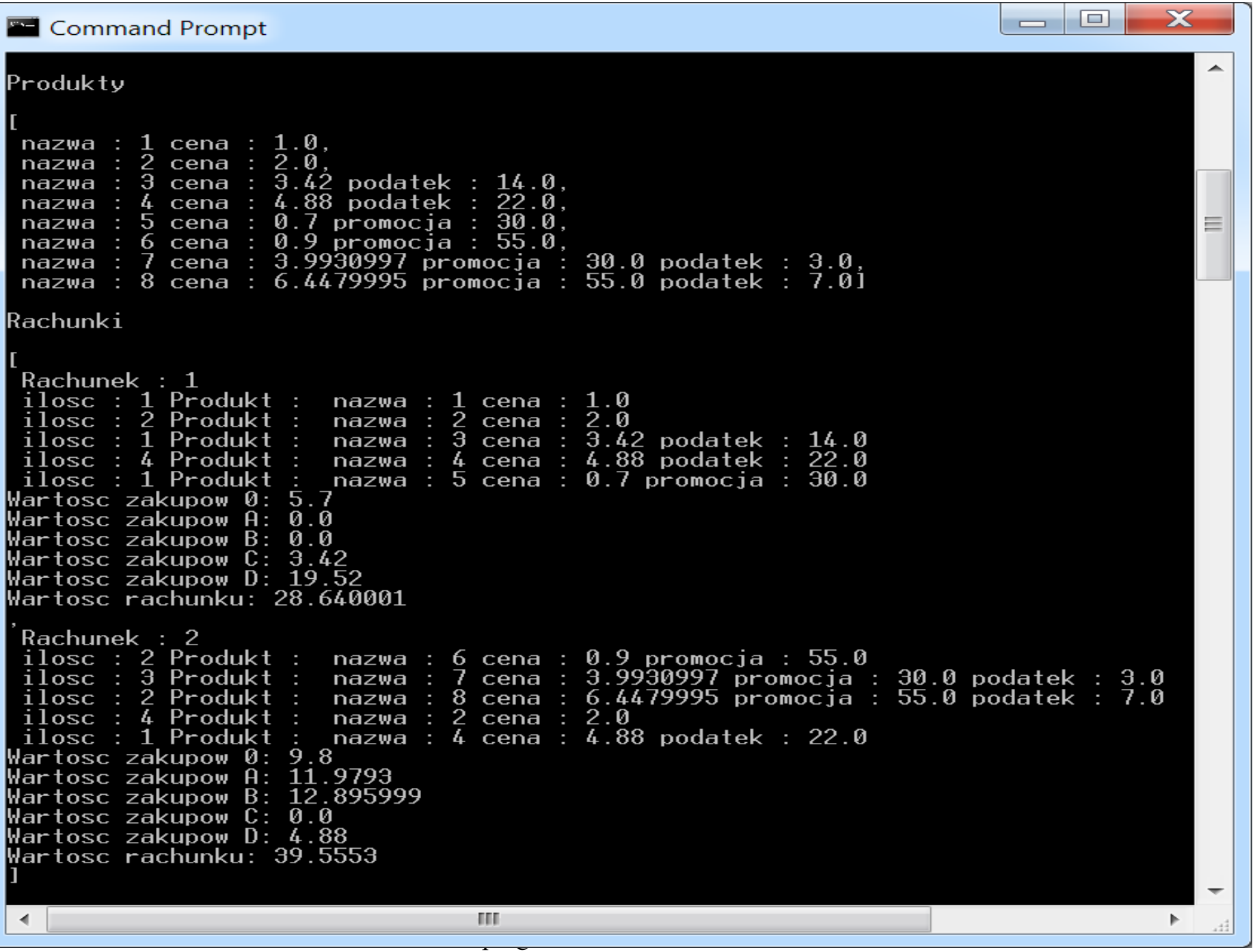


// Rachunek – zmiana kodu metody toString(),
// drukująca wartości rachunku w rożnych kategoriach podatku

```
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());
```

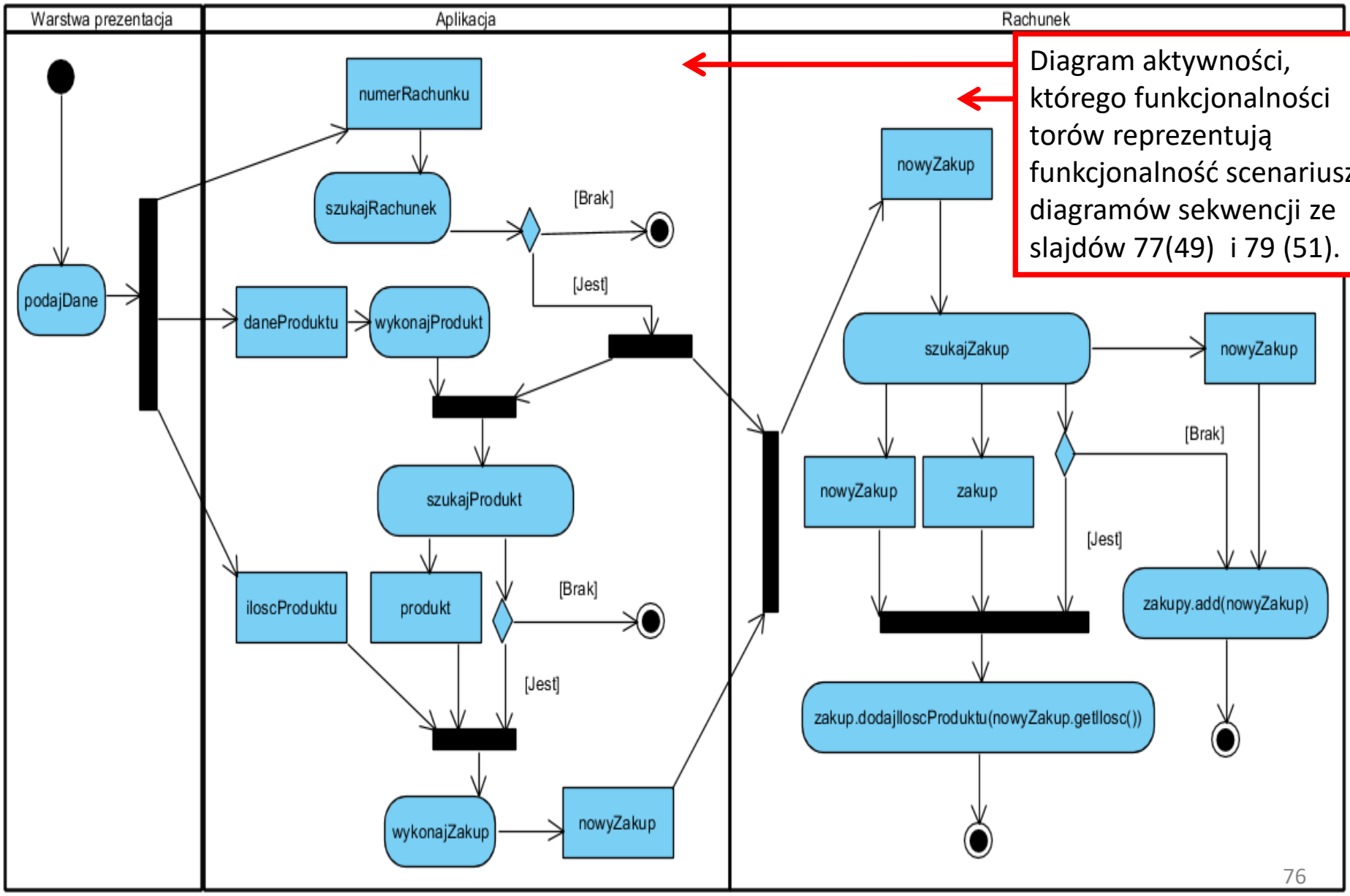
```
//c.d. kodu metody main po implementacji przypadków użycia:
// Szukanie rachunku i Wstawianie nowego rachunku
        app.wstawRachunek(1);
        app.wstawRachunek(1);
        app.wstawRachunek(2);
// Wstawianie nowego zakupu
        app.wstawZakup(1, 1, dane1);
        app.wstawZakup(1, 2, dane2);
        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
]

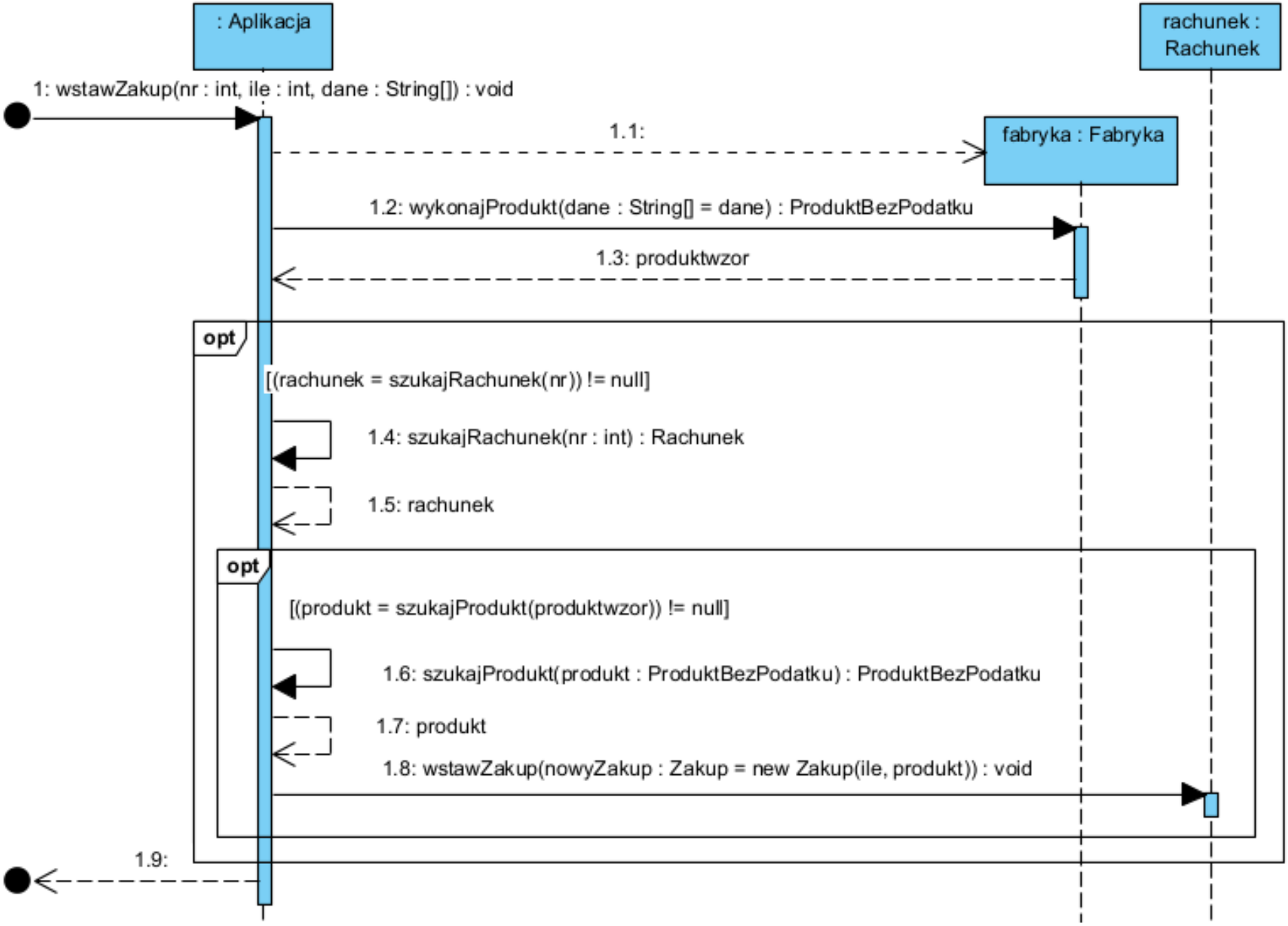
Diagramy klas, diagramy sekwencji

1. **Diagramy sekwencji UML**
   https://sparxsystems.com/resources/tutorials/uml2/sequence-diagram.html
2. **Przykłady diagramów sekwencji i klas – kontynuacja przykładu 2 z wykładu 2 i wykładu 3**
3. **Modelowanie zachowania obiektów za pomocą diagramów sekwencji i aktywności - porównanie**

Diagram czynności przypadku użycia ***Wstawianie nowego zakupu*** (**model przypadku użycia w warstwie biznesowej**)



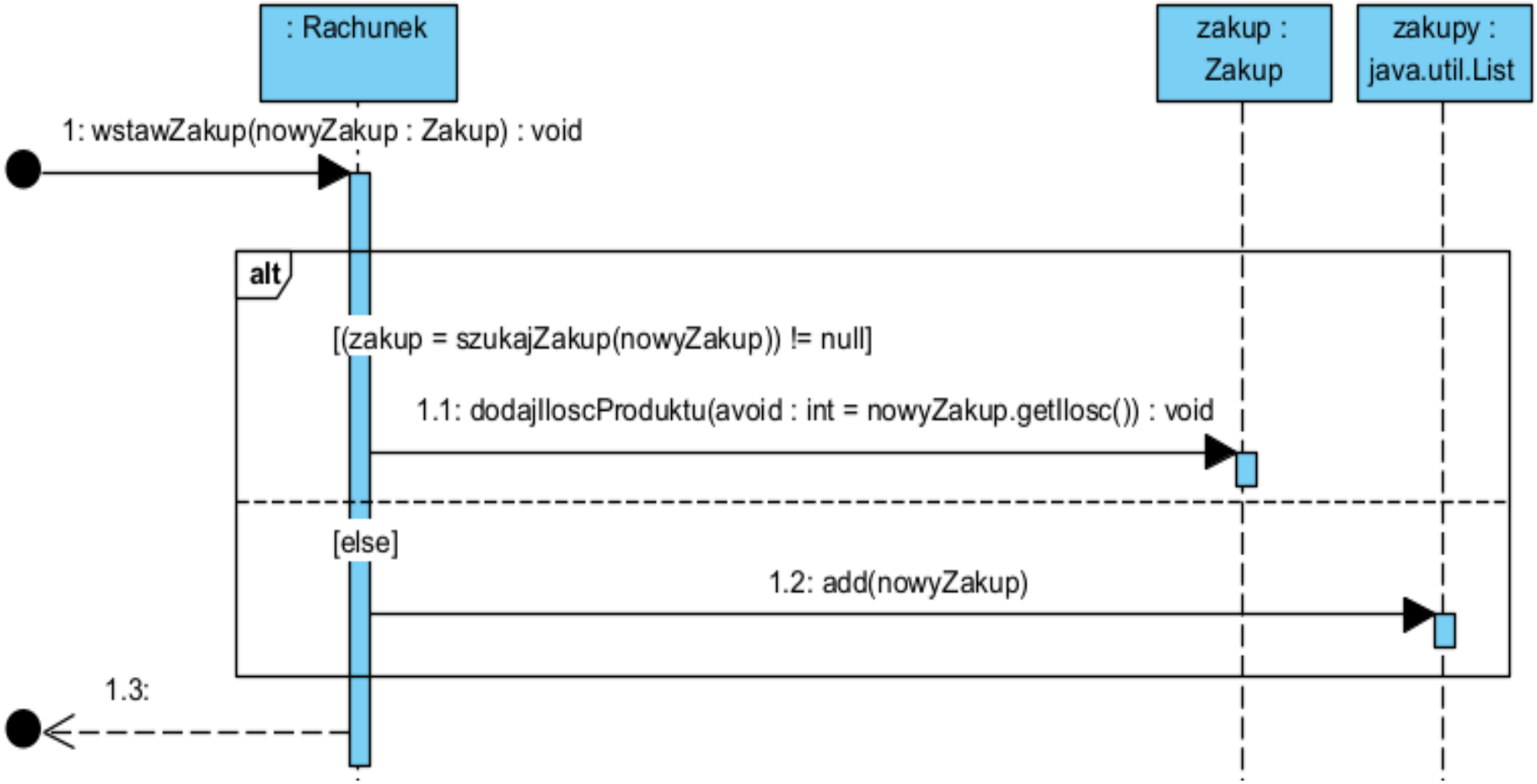
void wstawZakup (int nr, int ailosc, String dane[]) – scenariusz z toru Aplikacja



void wstawZakup (int nr, int ailosc, String dane[]) – kod z toru Aplikacja

```
//Aplikacja

  public   void wstawZakup (int nr, int ile, String dane[])
  {
    Rachunek rachunek;
     Fabryka fabryka = new Fabryka();
     ProduktBezPodatku produkt1 = fabryka.wykonajProdukt(dane);
     if ((rachunek=szukajRachunek(nr)) != null)
       if ((produkt1=szukajProdukt(produkt1)) != null)
         rachunek.wstawZakup(new Zakup(ile, produkt1));

}
```

void wstawZakup(Zakup azakup) – scenariusz z toru Rachunek



void wstawZakup(Zakup azakup) – kod z toru Rachunek

```
//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);
  }
```

Dziękuję za uwagę