Modele cyklu życia systemu cd Zasady programowania zwinnego Wykład 3

Zofia Kruczkiewicz

Literatura

1. Roger S. Pressman, Praktyczne podejście do oprogramowania, WNT, 2004
2. Stephen H. Kan, Metryki i modele w inżynierii jakości oprogramowania, Mikom, 2006
3. Jacobson, Booch, Rumbaung, The Unified Software Development Process,Addison Wesley, 1999
4. Shalloway A.,Trott James R.,Projektowanie zorientowane obiektowo. Wzorce projektowe. Gliwice, Helion, 2005
5. Robert C. Martin, Micah Martin, Agile Programowanie zwinne. Zasady, wzorce i praktyki zwinnego wytwarzania oprogramowania w C#, Helion 2008

Zasady ewolucyjnego zwinnego tworzenia oprogramowania [5]

(1) Zasady procesu zwinnego

- **Zasada pojedynczej odpowiedzialności (Single-Responsibility Principle - SRP)**

  Żadna klasa nie może być modyfikowana z więcej niż jednego powodu.

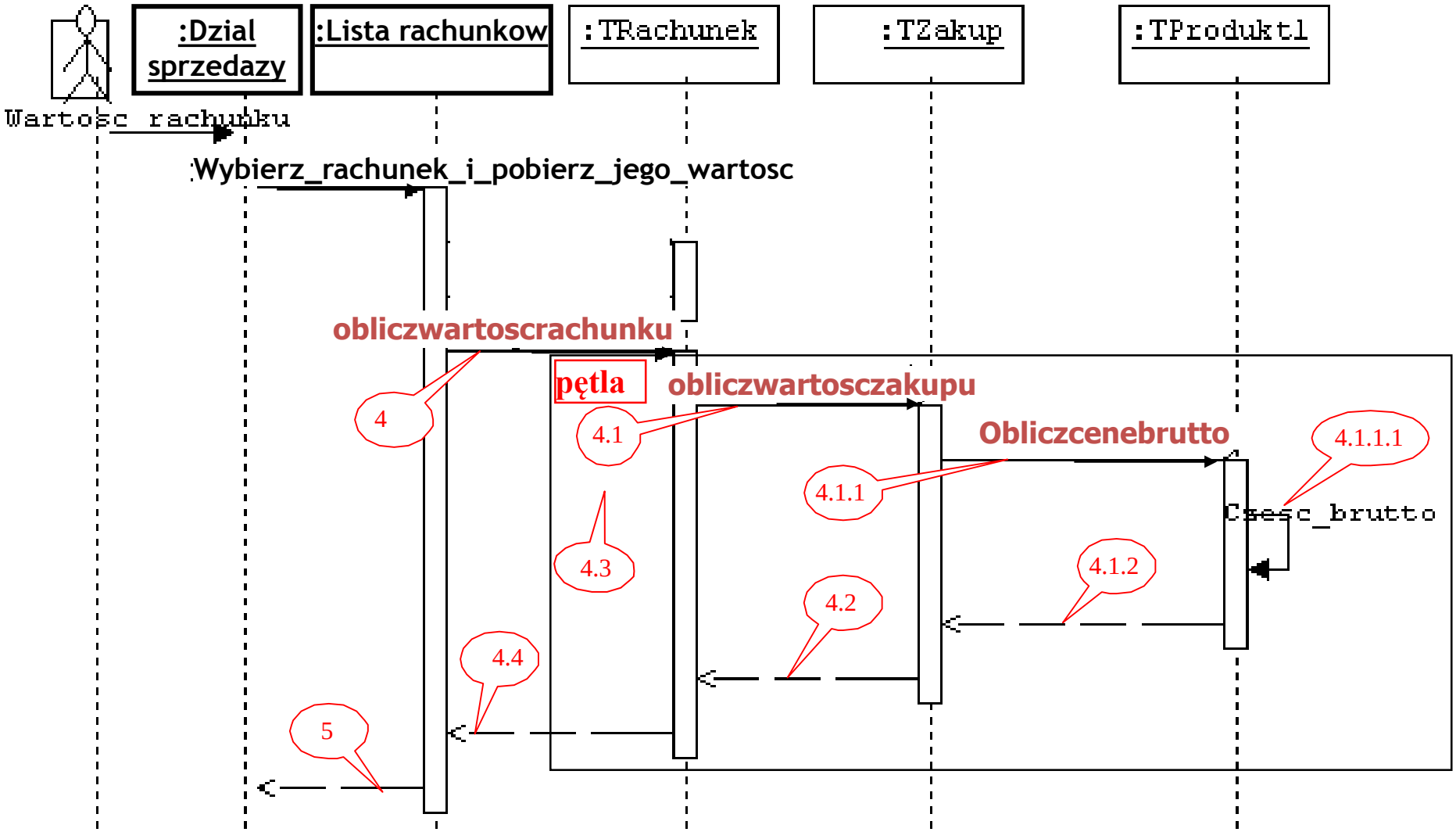
  1) Klasa obsługująca reguły biznesowe nie powinna zarządzać trwałością

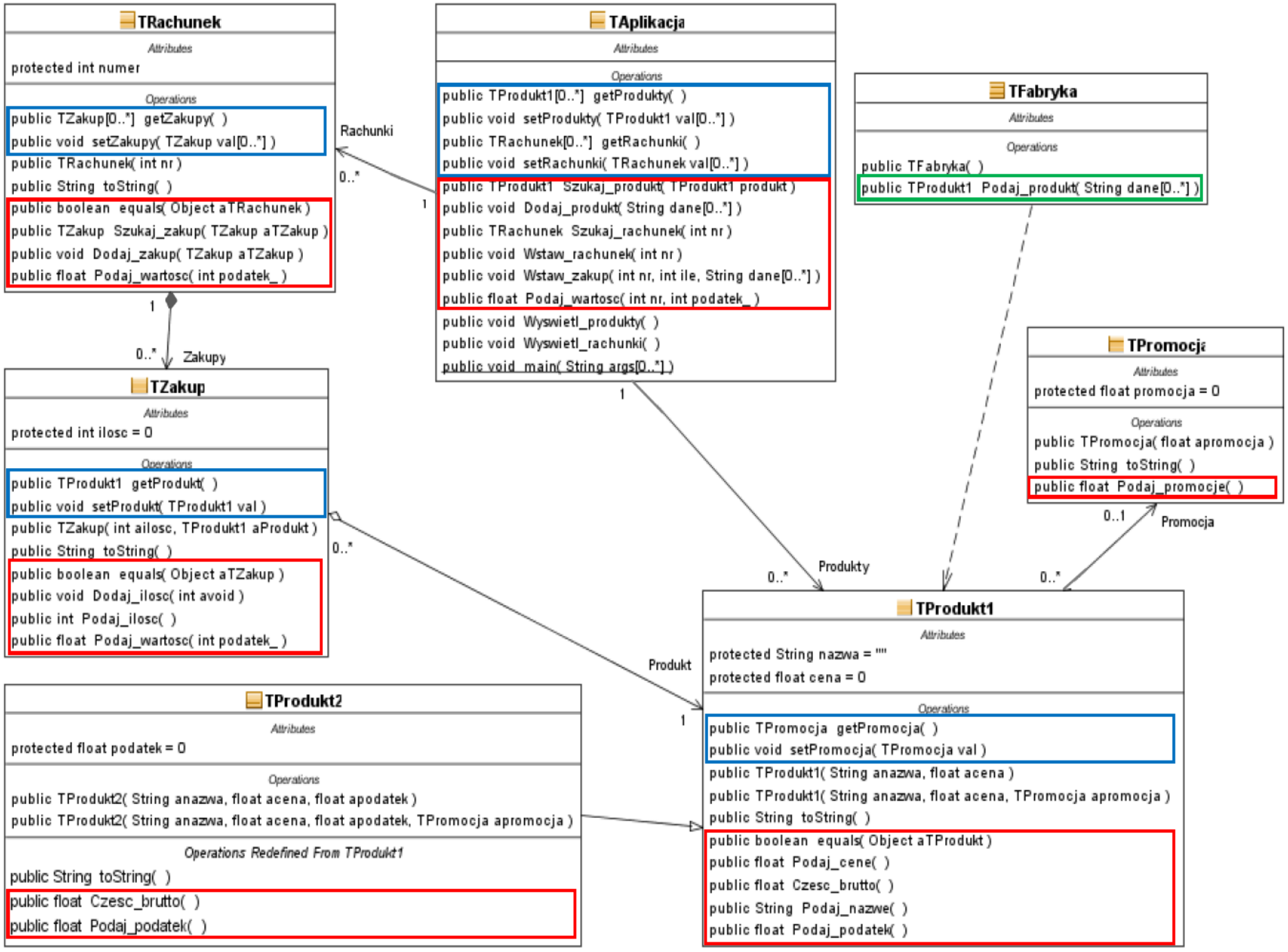
  2) Klasa tworząca obiekty nie powinna ich używać

  3) Oddzielanie wzajemnie powiązanych odpowiedzialności- np. obiektowa idea rachunku:

  – Zmiana promowania ze względu na producenta produktu - tylko modyfikacja kodu klas z rodziny TProdukt
  – Zmiana promowania ze względu na ilość zakupionego produktu - tylko modyfikacja kodu klasy TZakup
  – Sposób wyznaczania wartości rachunku np. ze względu na grupy podatkowe – zmiana kodu klast typu TRachunek

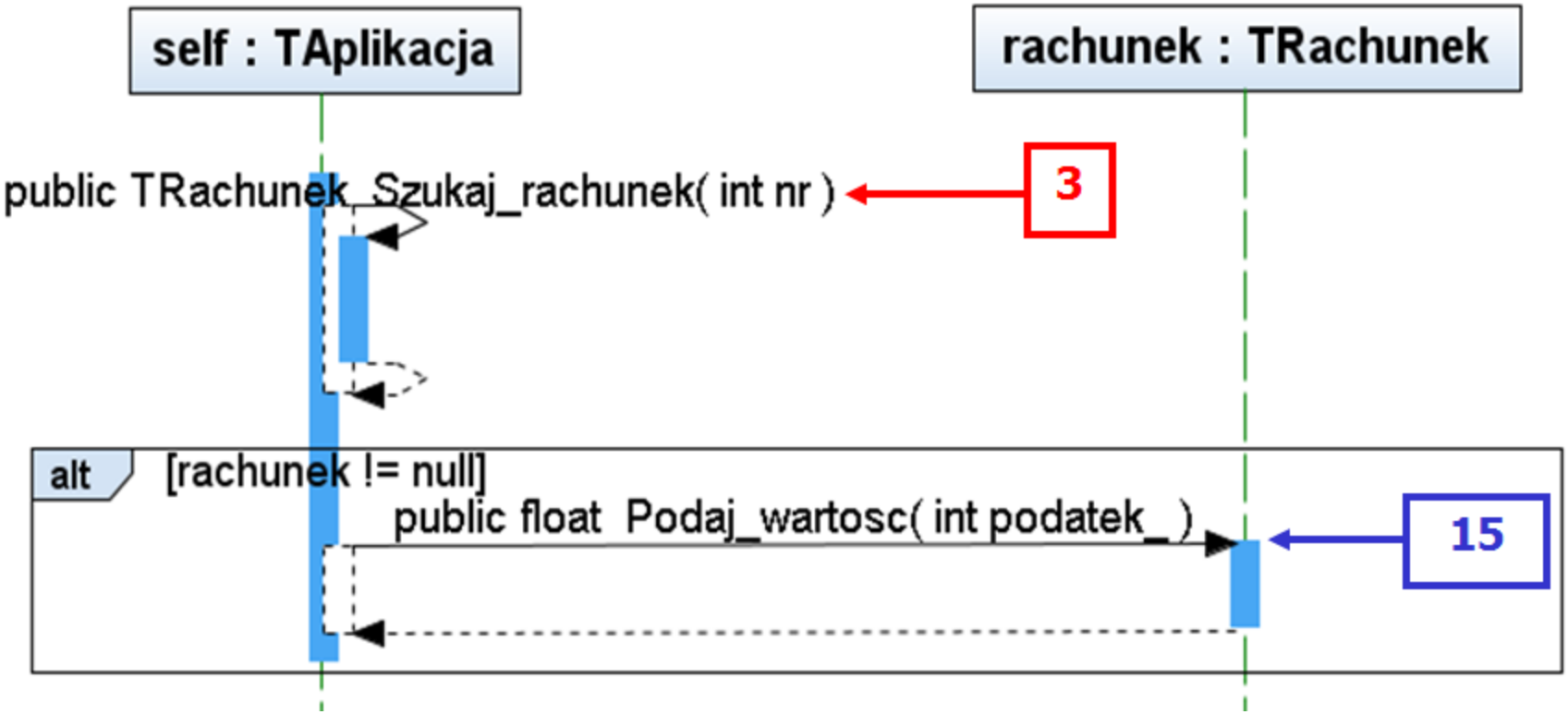
Diagram sekwencji UML– obiektowy sposób przedstawienia scenariusza obliczania rachunku




TRachunek
Attributes
protected int numer
Operations
public TZakup[0..*] getZakupy( )
public void setZakupy(TZakup val[0..*])
public TRachunek(int nr)
public String toString( )
public boolean equals(Object aTRachunek)
public TZakup Szukaj_zakup(TZakup aTZakup)
public void Dodaj_zakup(TZakup aTZakup)
public float Podaj_wartosc(int podatek_)
TAplikacja
Attributes
Operations
public TProdukt1[0..*] getProdukty( )
public void setProdukty(TProdukt1 val[0..*])
public TRachunek[0..*] getRachunki( )
public void setRachunki(TRachunek val[0..*])
public TProdukt1 Szukaj_produkt(TProdukt1 produkt)
public void Dodaj_produkt(String dane[0..*])
public TRachunek Szukaj_rachunek(int nr)
public void Wstaw_rachunek(int nr)
public void Wstaw_zakup(int nr, int ile, String dane[0..*])
public float Podaj_wartosc(int nr, int podatek_)
public void Wyswietl_produkty( )
public void Wyswietl_rachunki( )
public void main(String args[0..*])
TFabryka
Attributes
Operations
public TFabryka( )
public TProdukt1 Podaj_produkt(String dane[0..*])
TPromocja
Attributes
protected float promocja = 0
Operations
public TPromocja(float apromocja)
public String toString( )
public float Podaj_promocje( )
TZakup
Attributes
protected int ilosc = 0
Operations
public TProdukt1 getProdukt( )
public void setProdukt(TProdukt1 val)
public TZakup(int ailosc, TProdukt1 aProdukt)
public String toString( )
public boolean equals(Object aTZakup)
public void Dodaj_ilosc(int avoid)
public int Podaj_ilosc( )
public float Podaj_wartosc(int podatek_)
TProdukt1
Attributes
protected String nazwa = ""
protected float cena = 0
Operations
public TPromocja getPromocja( )
public void setPromocja(TPromocja val)
public TProdukt1(String anazwa, float acena)
public TProdukt1(String anazwa, float acena, TPromocja apromocja)
public String toString( )
public boolean equals(Object aTProdukt)
public float Podaj_cene( )
public float Czesc_brutto( )
public String Podaj_nazwe( )
public float Podaj_podatek( )
TProdukt2
Attributes
protected float podatek = 0
Operations
public TProdukt2(String anazwa, float acena, float apodatek)
public TProdukt2(String anazwa, float acena, float apodatek, TPromocja apromocja)
Operations Redefined From TProdukt1
public String toString( )
public float Czesc_brutto( )
public float Podaj_podatek( )
Rachunki
0..*
1
1
0..*
Zakupy
1
0..*
Produkty
0..*
0..*
Produkt
1
0..1
Promocja
0..*

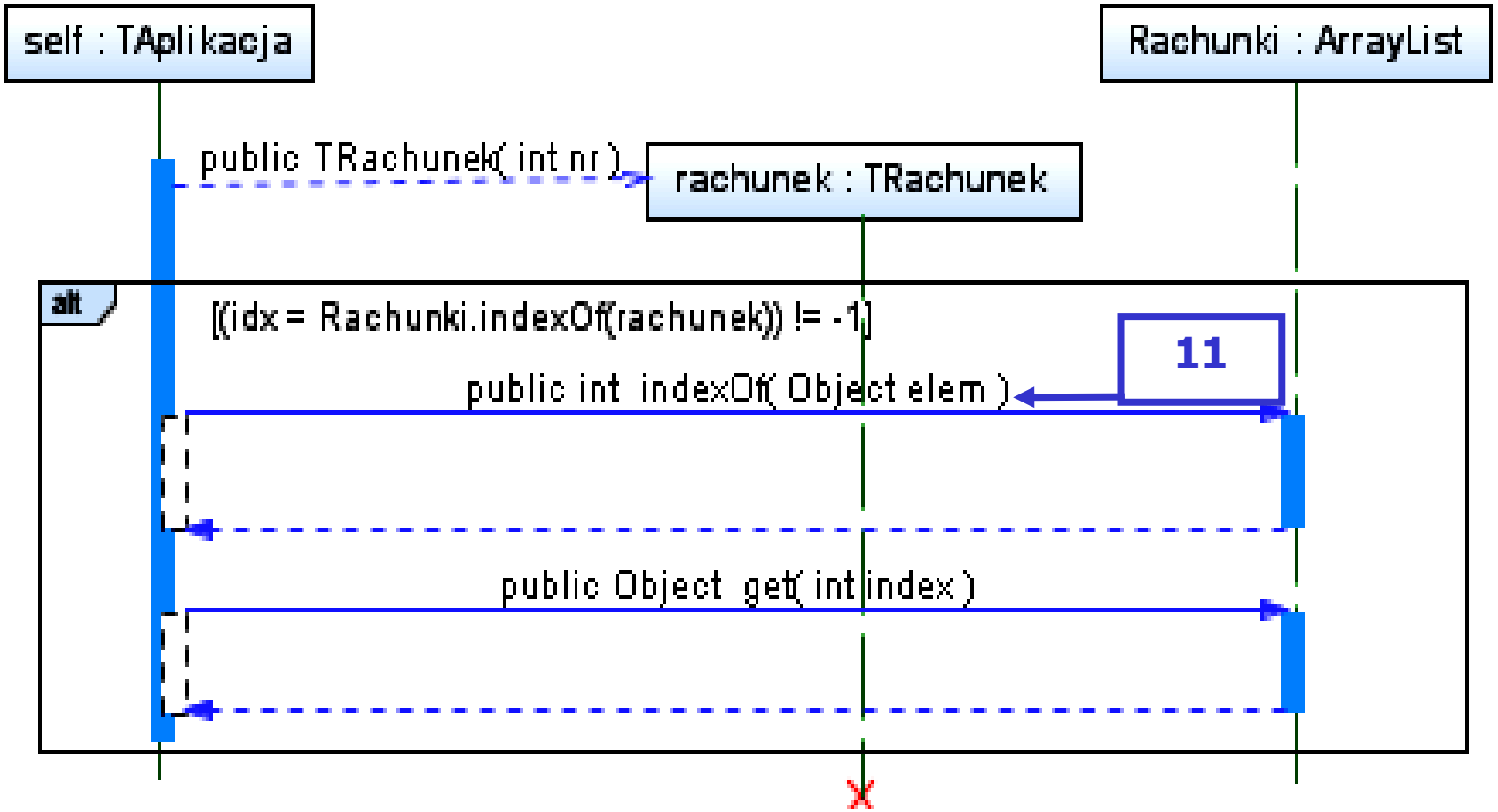
(6) Obliczanie wartosci rachunku

(float TAplikacja::Podaj_wartosc(int nr, int podatek_))

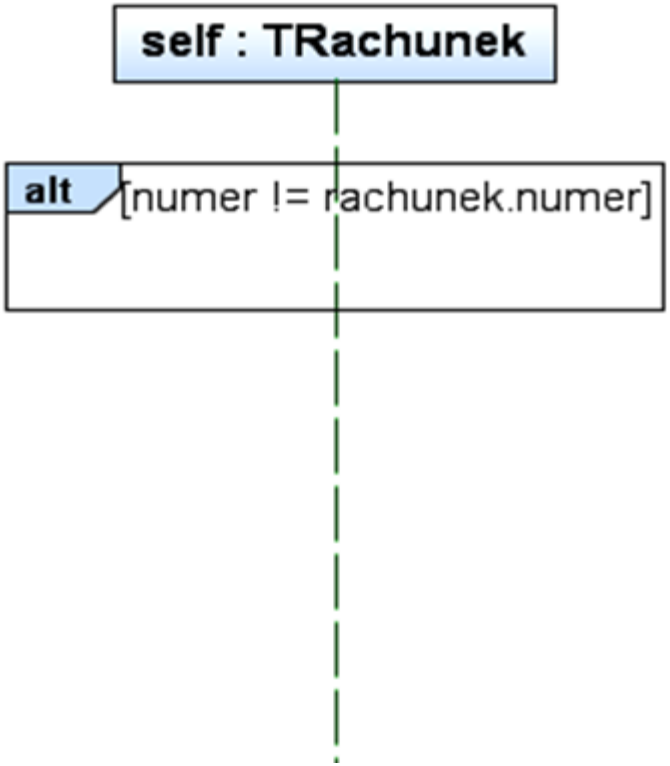


(3) Szukanie rachunku

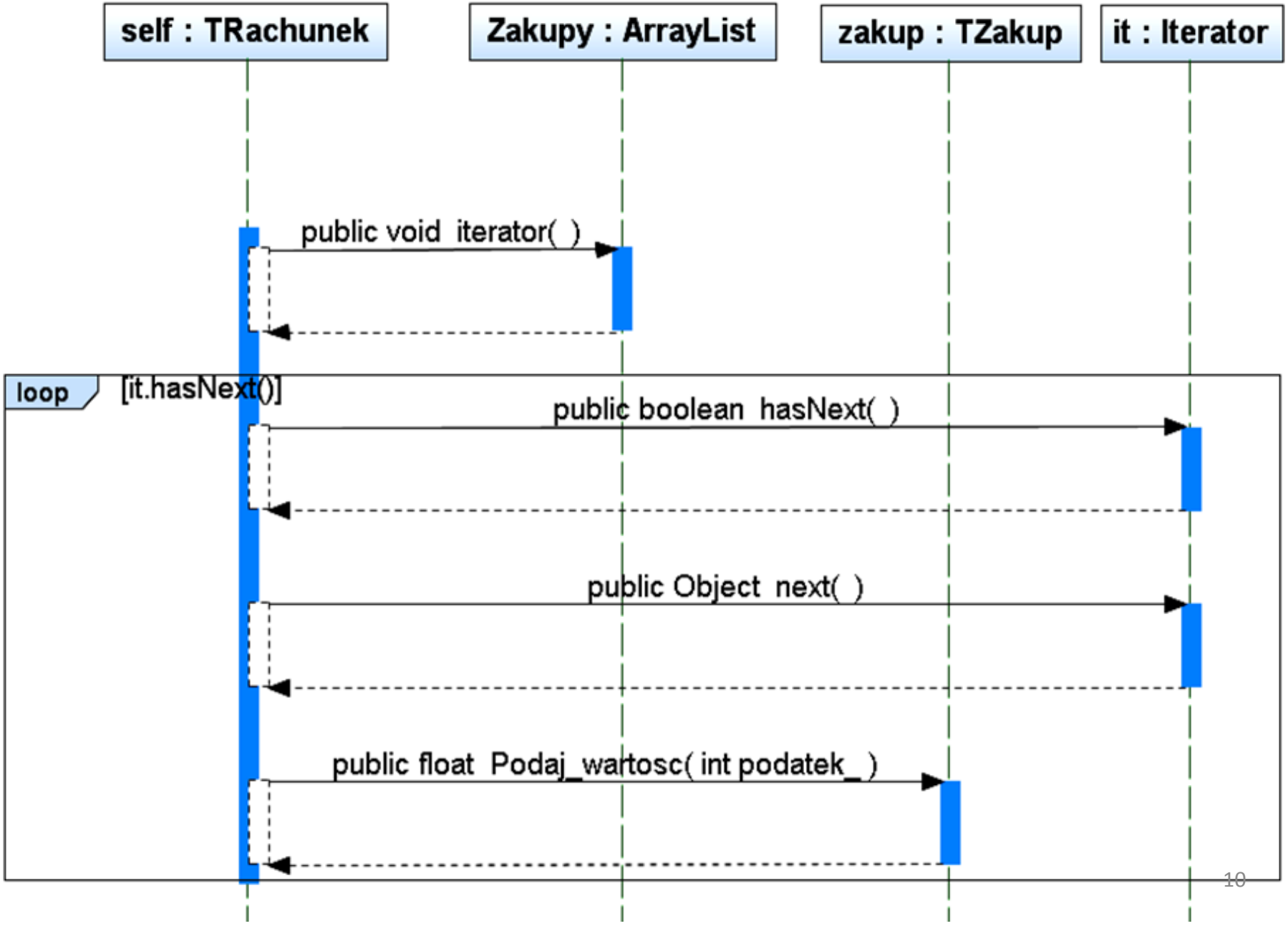
(TRachunek TAplikacja::Szukaj_rachunek(int nr))



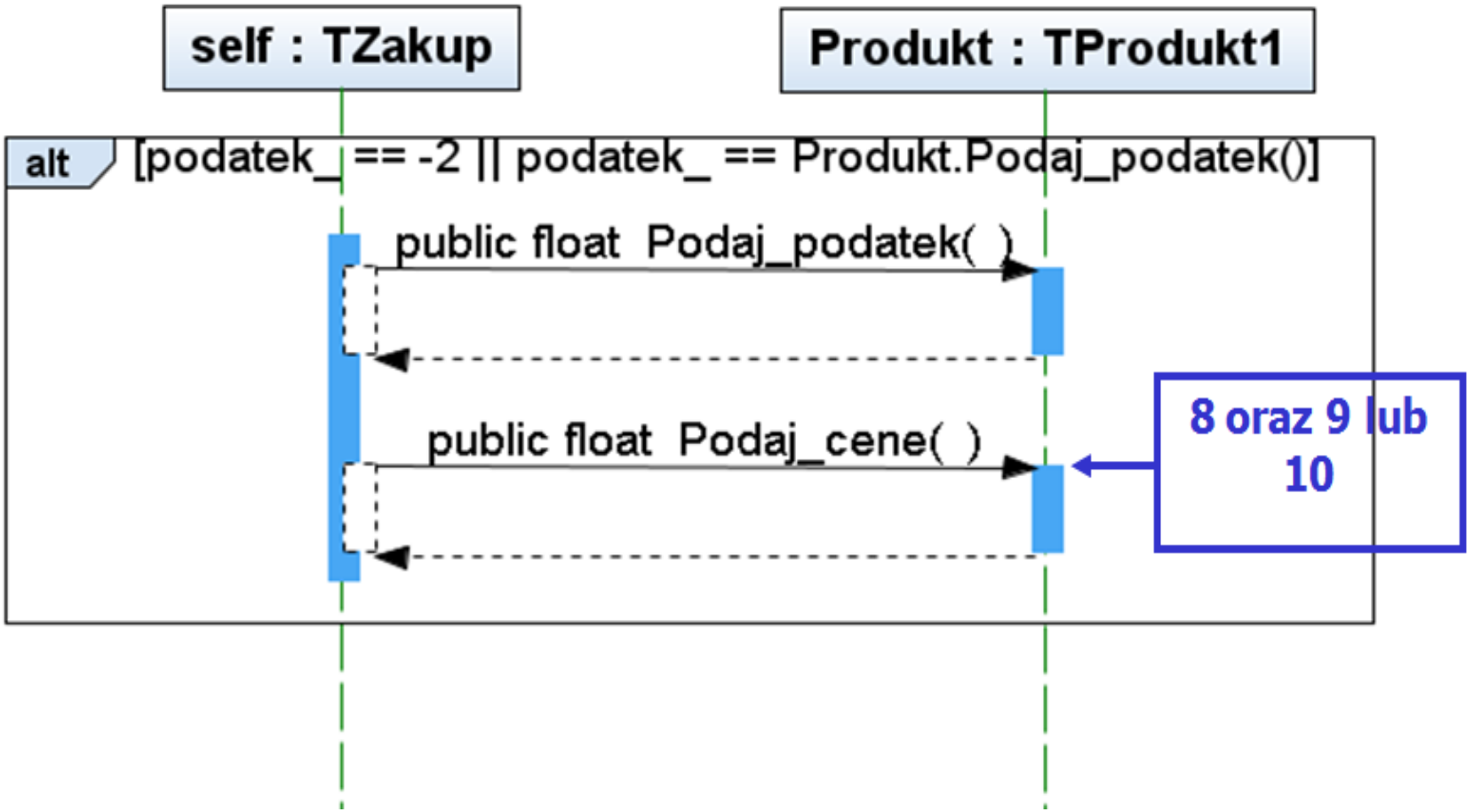
(11) boolean TRachunek::equals(Object rachunek)



(15) float TRachunek::Podaj_wartosc(int podatek_)

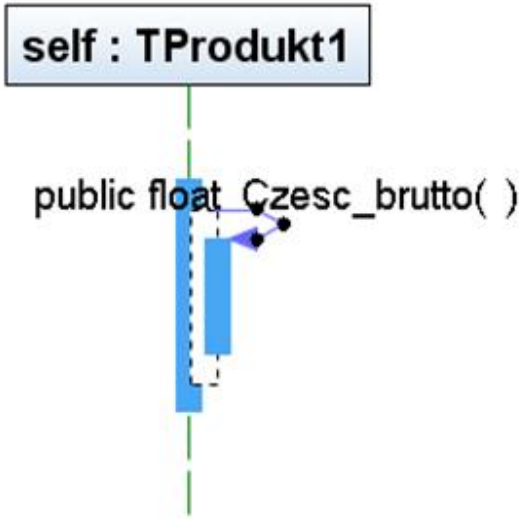


(16) float TZakup::Podaj_wartosc(int podatek_)



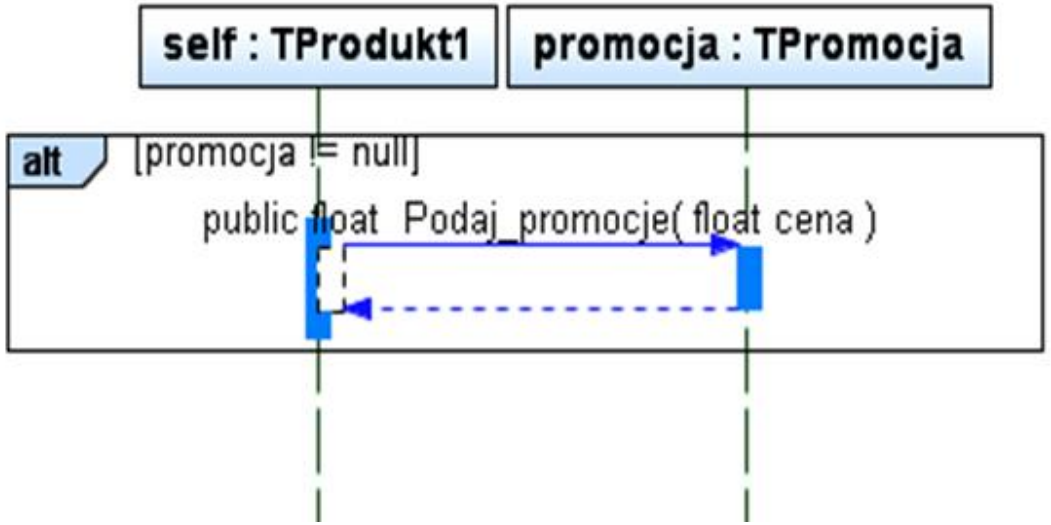
(8)

float TProdukt1::Podaj_cene()

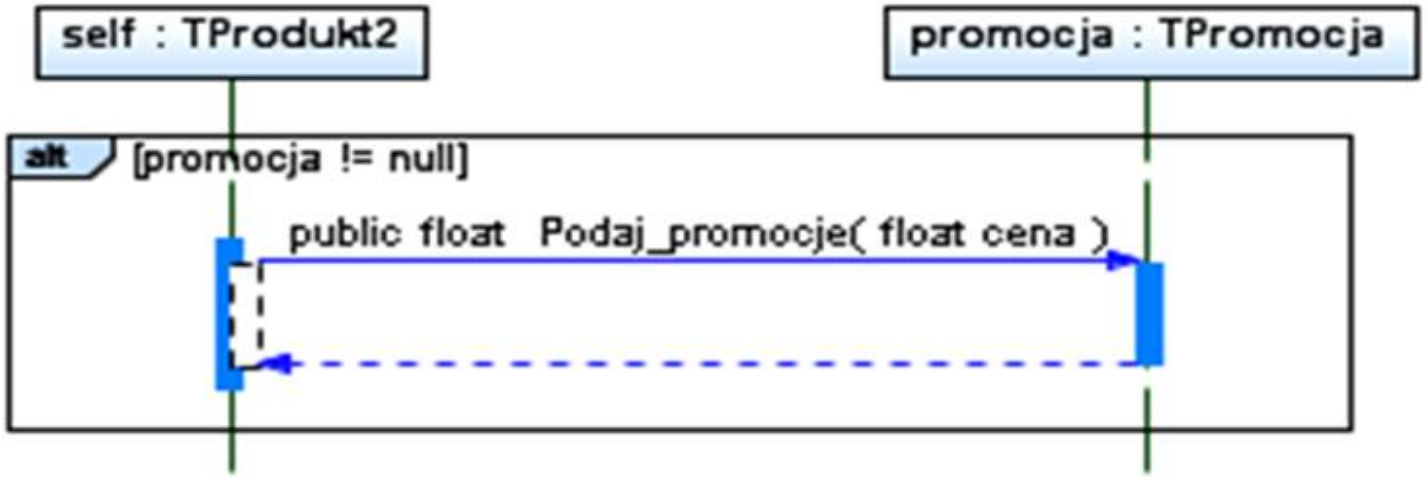


(9)

float TProdukt1::Czesc_brutto()



(10) Przedefiniowanie metody Czesc_brutto()
float TProdukt2::Czesc_brutto()



(2) Zasady procesu zwinnego

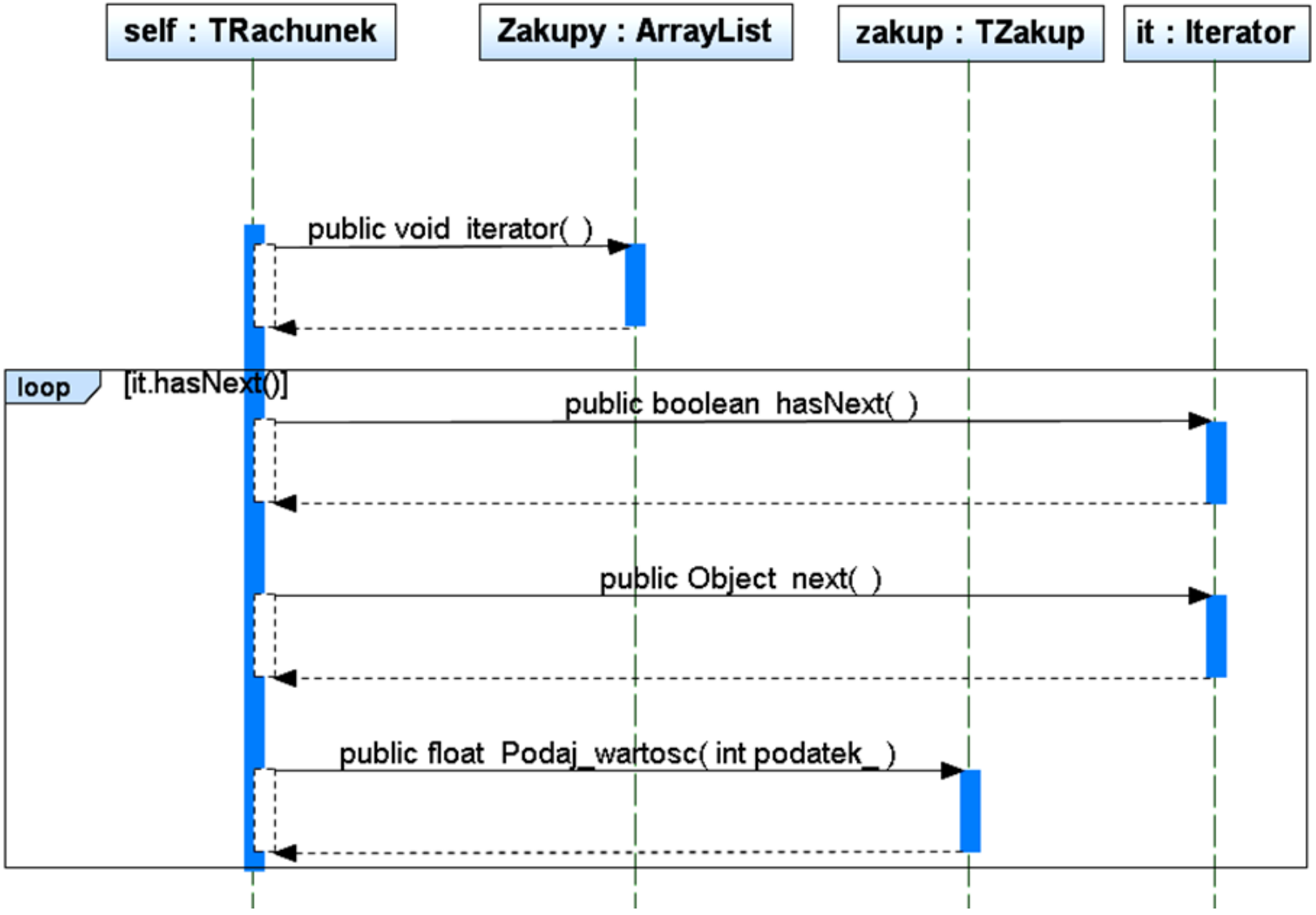
- **Zasada otwarte-zamknięte (Open/Closed Principle - OCP)**

  Składniki oprogramowania (klasy, moduły, funkcje itp.) powinny być otwarte na rozbudowę, ale zamknięte dla modyfikacji .

1) Stosowanie **dziedziczenia, polimorfizmu, implementacji interfejsów** tylko w takich przypadkach, gdy istnieje możliwość zmian.
2) Należy wyeliminować rozpoznawanie klas zarządzanych lub używanych np. instrukcją **switch** przez klasy, które **używają** lub **zarządzają** tymi klasami
3) Przykłady: obiektowa idea rachunku:
   - zmiana promowania ze względu na producenta produktu - tylko modyfikacja kodu produktów przez polimorfizm i dziedziczenie (TProdukt1, TProdukt2)
   - zmiana promowania ze względu na ilość zakupionego produktu - tylko modyfikacja kodu zakupów przez dziedziczenie i polimorfizm (TZakup1, TZakup2, itp.)
   - Sposób wyznaczania wartości rachunku np. ze względu na grupy podatkowe – zmiana kodu rachunku przy sumowania wartości zakupów przez dziedziczenie i polimorfizm (TRachunek1, TRachunek1…)

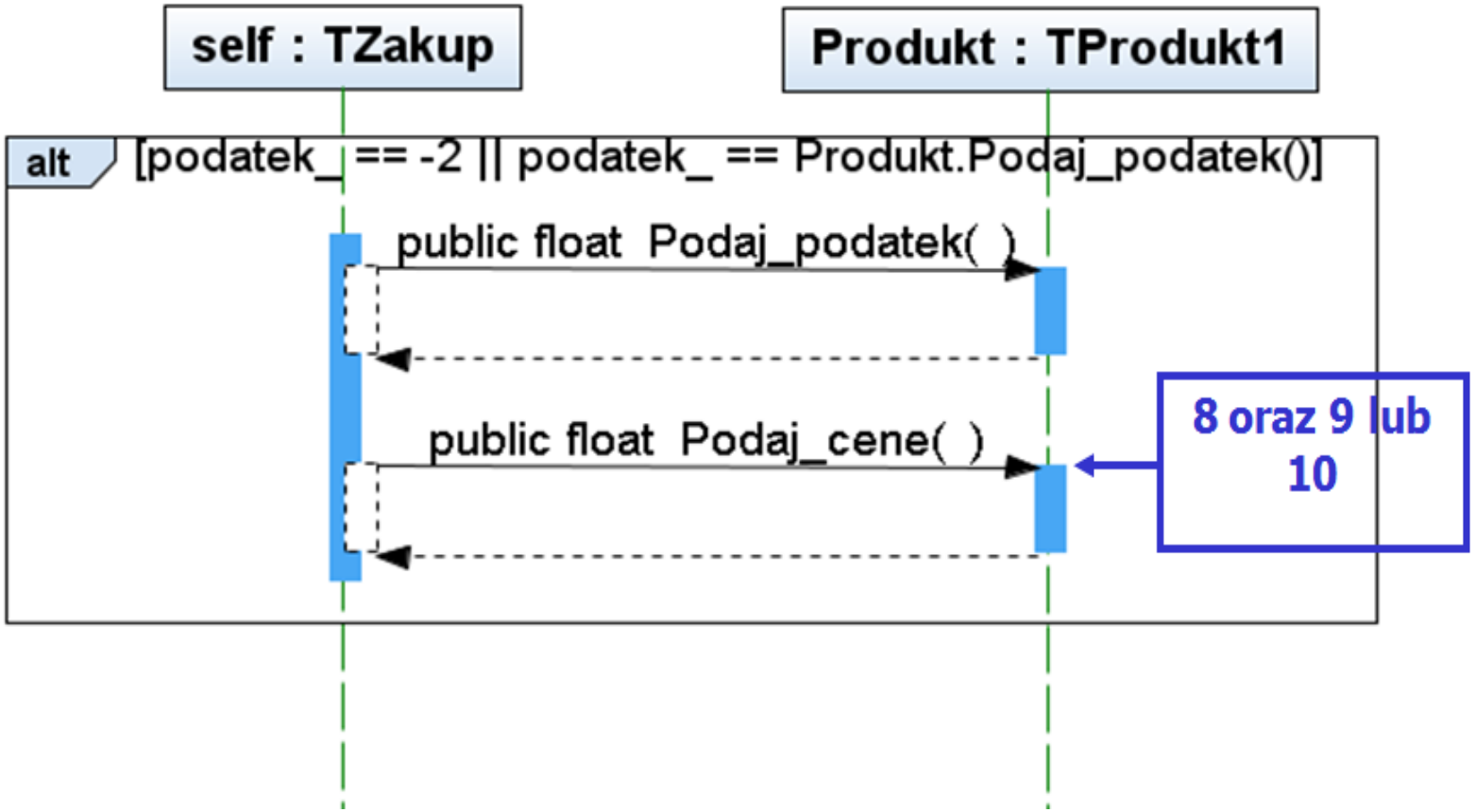
(15) float TRachunek::Podaj_wartosc(int podatek_)

Scenariusz metody Podaj_wartosc jest niezalezny od scenariusza wywołanej metody Podaj_wartosc od obiektu klasy TZakup



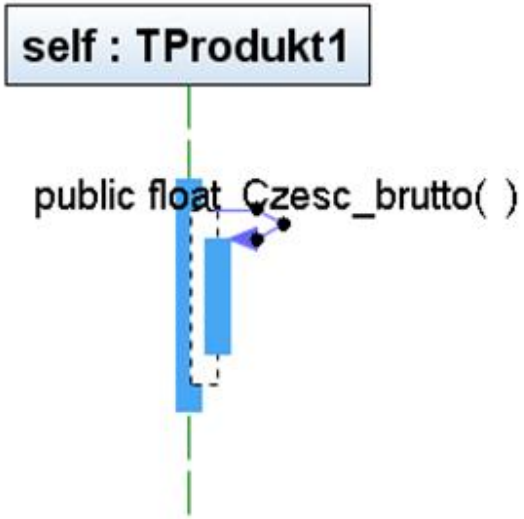
(16) float TZakup::Podaj_wartosc(int podatek_)

Scenariusz metody Podaj_wartosc niezależny od zmiennego scenariusza metody Podaj_cene



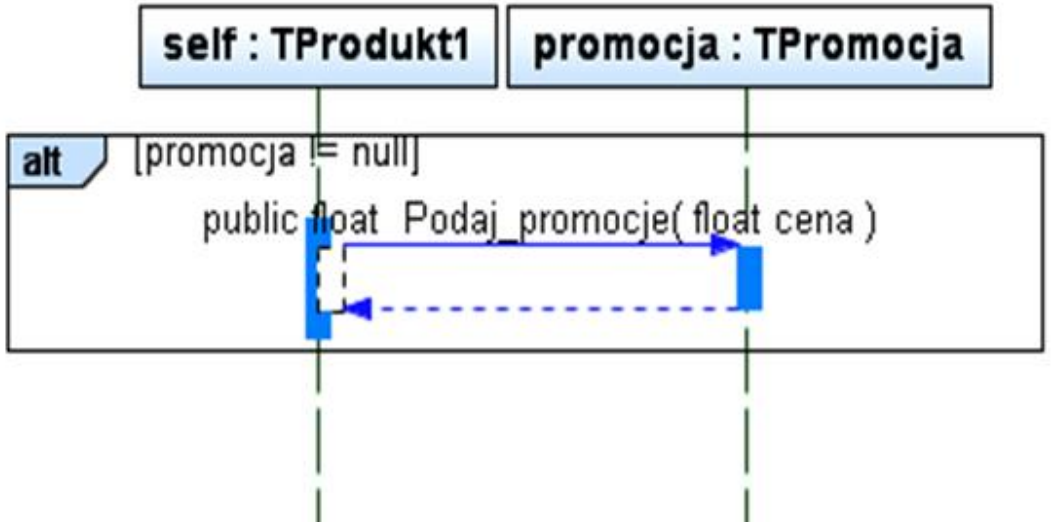
(8)

float TProdukt1::Podaj_cene()

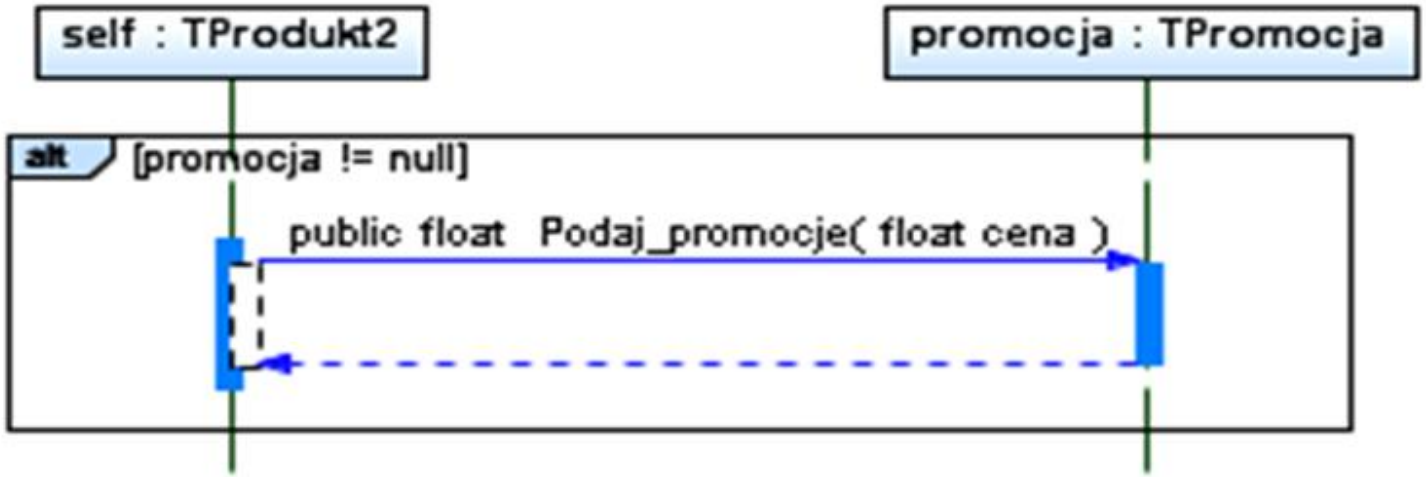


(9)

float TProdukt1::Czesc_brutto()



(10) Przedefiniowanie metody Czesc_brutto()
float TProdukt2::Czesc_brutto()



(3) Zasady procesu zwinnego

- **Zasada podstawiania Liskov**

  Musi istnieć możliwość zastępowania typów bazowych ich podtypami.

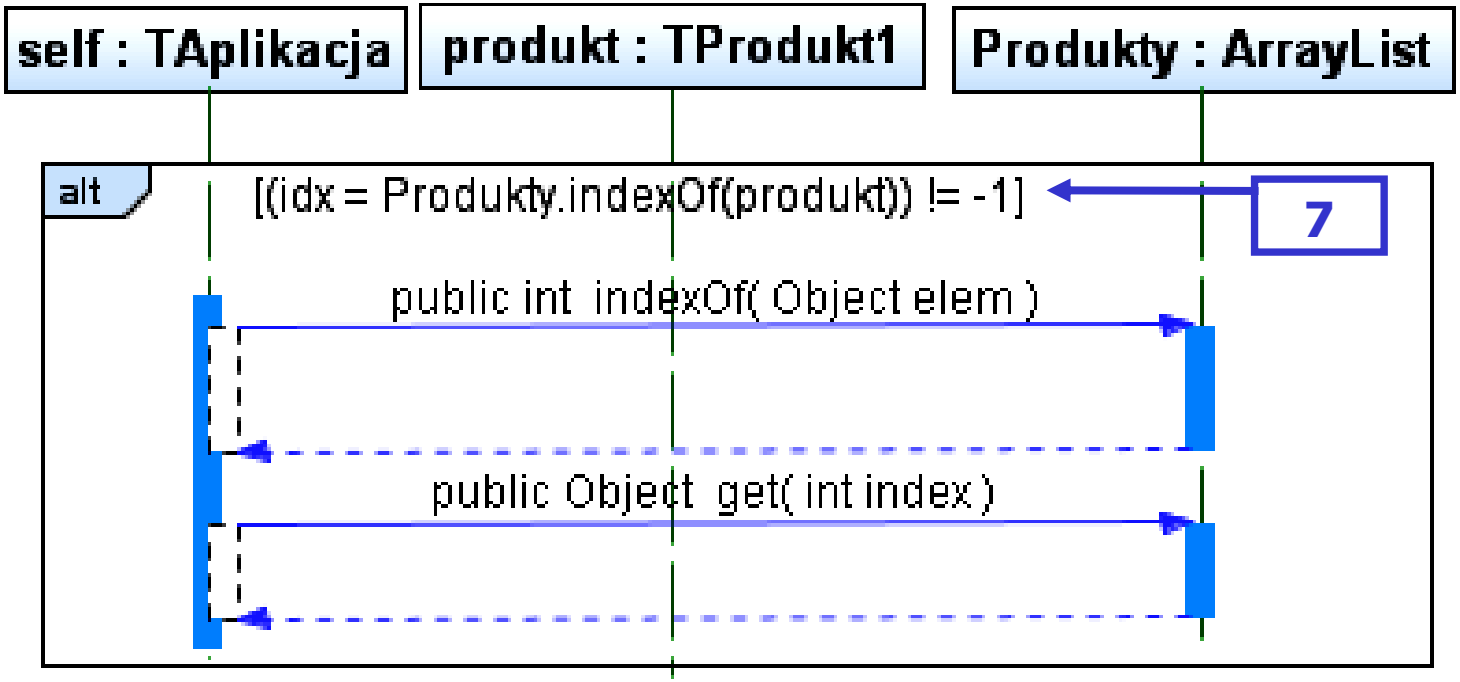
  Jest to warunek zasady otwarte-zamknięte (OCP).

  1) Klasa potomna nie może mieć mniejszej funkcjonalności niż jej klasa bazowa. Podstawianie klasy potomnej powinno następować automatycznie, bez potrzeby rozpoznawania typu obiektu np. instrukcją **switch**
  2) Przykład: obiektowa idea rachunku:
     – Zakup obliczając cenę nie musi znać typu produktu – zawsze otrzymuje cenę brutto (wynikającą z podatku lub/i promocji). Naruszeniem zasady byłoby stosowanie instrukcji if w celu rozpoznania typu produktu, aby pobrać różnie nazwane metody do obliczenia ceny brutto.

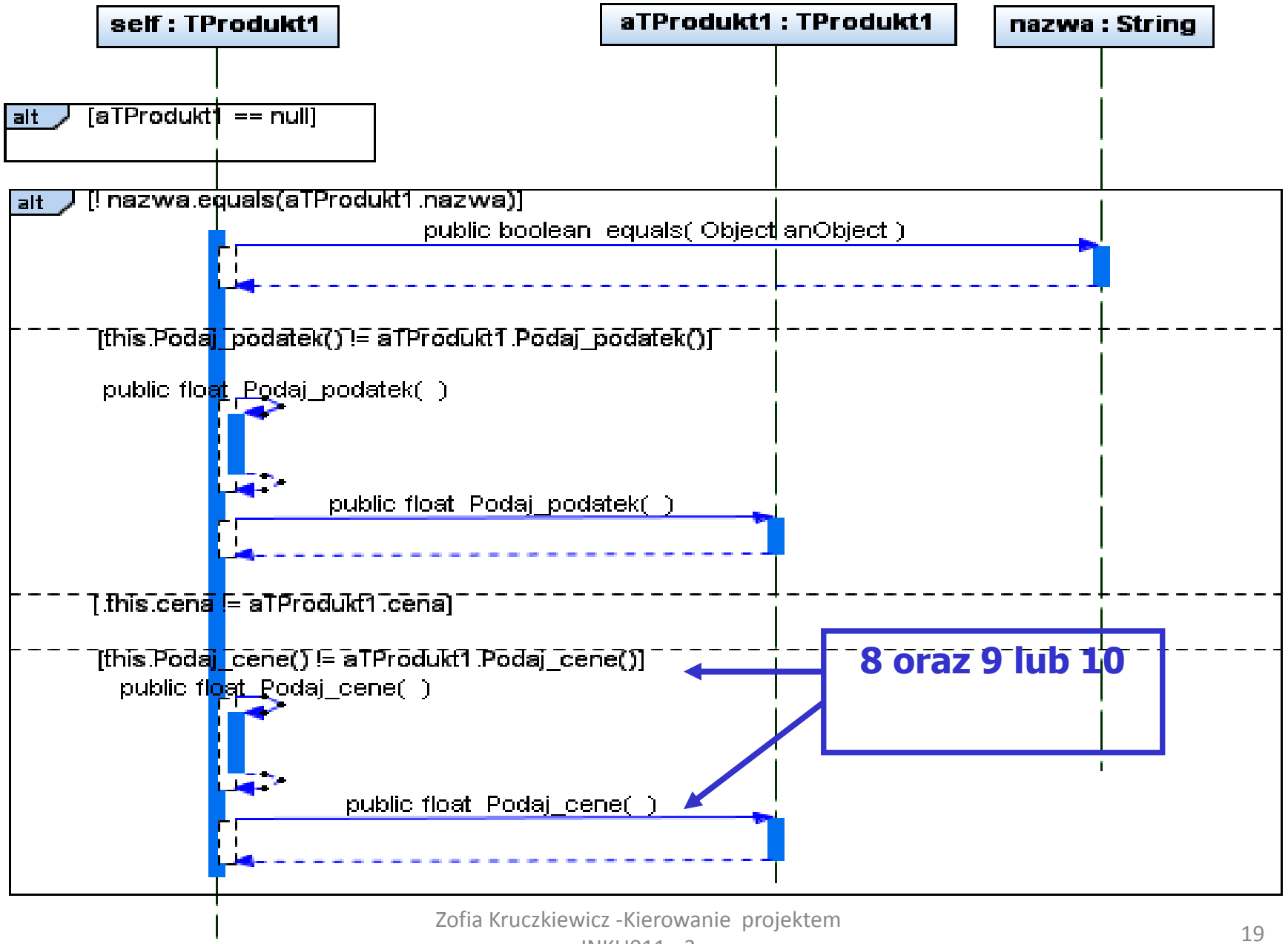
(1) Szukanie produktu

(TProdukt1 TAplikacja::Szukaj_produkt(TProdukt1 produkt))

Klasa **Object** - metody equals, hashCode
Klasy dziedziczące – mogą przedefiniować te metody equals, hashCode
Kolekcje z pakietu java.util używają te metody np w metodzie indexOf.



(7) boolean TProdukt1::equals(Object aTProdukt)



(4) Zasady procesu zwinnego

- **Zasada odwracania zależności (Dependency Inversion Principle -DIP)**

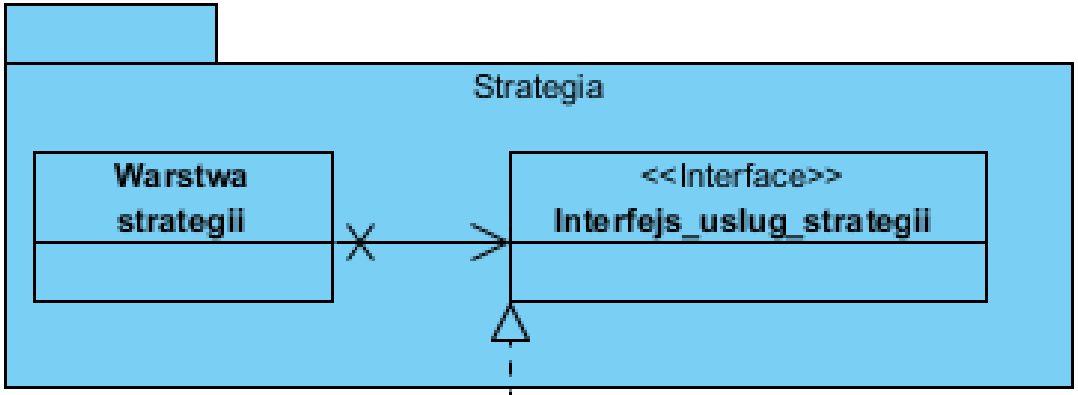
  A. Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Obie grupy modułów powinny zależeć od abstrakcji

  B. Abstrakcje nie powinny zależeć od szczegółowych rozwiązań. To szczegółowe rozwiązania powinny zależeć od abstrakcji.

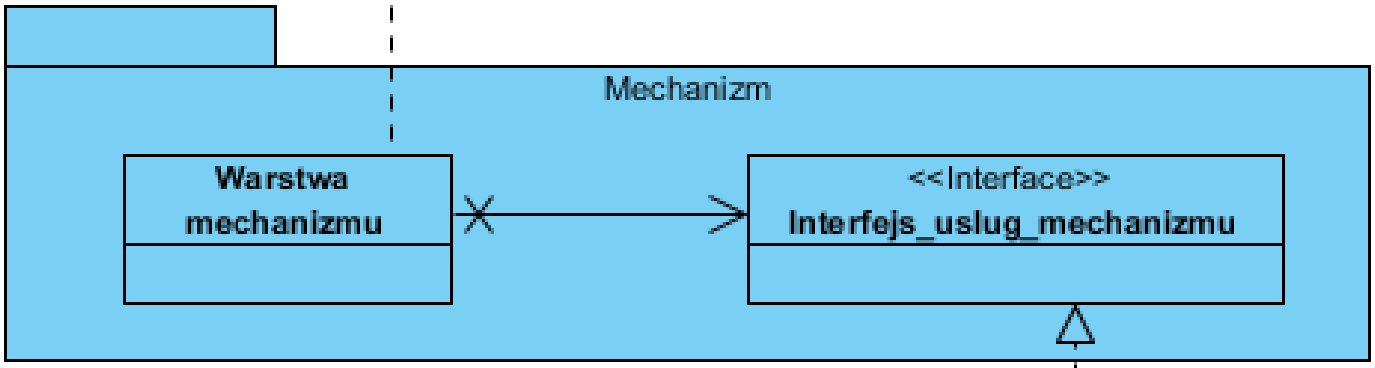
1) Strategia programu nie powinna zależeć od szczegółowych rozwiązań w zakresie implementacji. Przykłady:
   - Implementacja klasy JApplet
   - Implementacja interfejsów jako słuchaczy zdarzeń w pakiecie Swing
   - Implementacja obiektów typu wątki

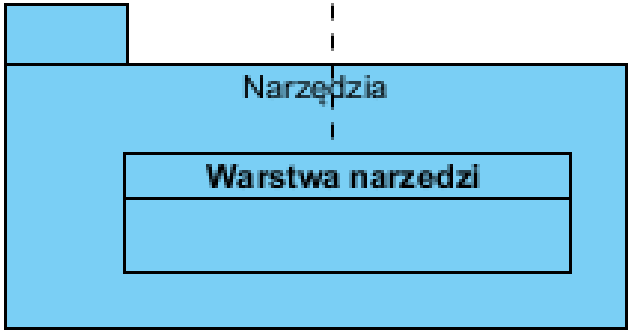
(4) Zasady procesu zwinnego

2) Podział na warstwy



- Warstwy niższe zależą od wyższych
- Eliminacja zależności przechodniej: warstwa strategii – warstwa narzędzi
- Eliminacja zależności bezpośredniej: warstwa strategii – warstwa mechanizmu





(4) Zasady procesu zwinnego

3) Odwracanie relacji własności („Nie dzwoń do nas. To my zadzwonimy do ciebie.”)

- Interfejsy są związane ze swoimi właścicielami, a nie implementującymi je klasami
- Warstwa strategii może być wielokrotnie używana w dowolnym kontekście pod warunkiem, że moduły niższego poziomu implementują interfejs usług strategii

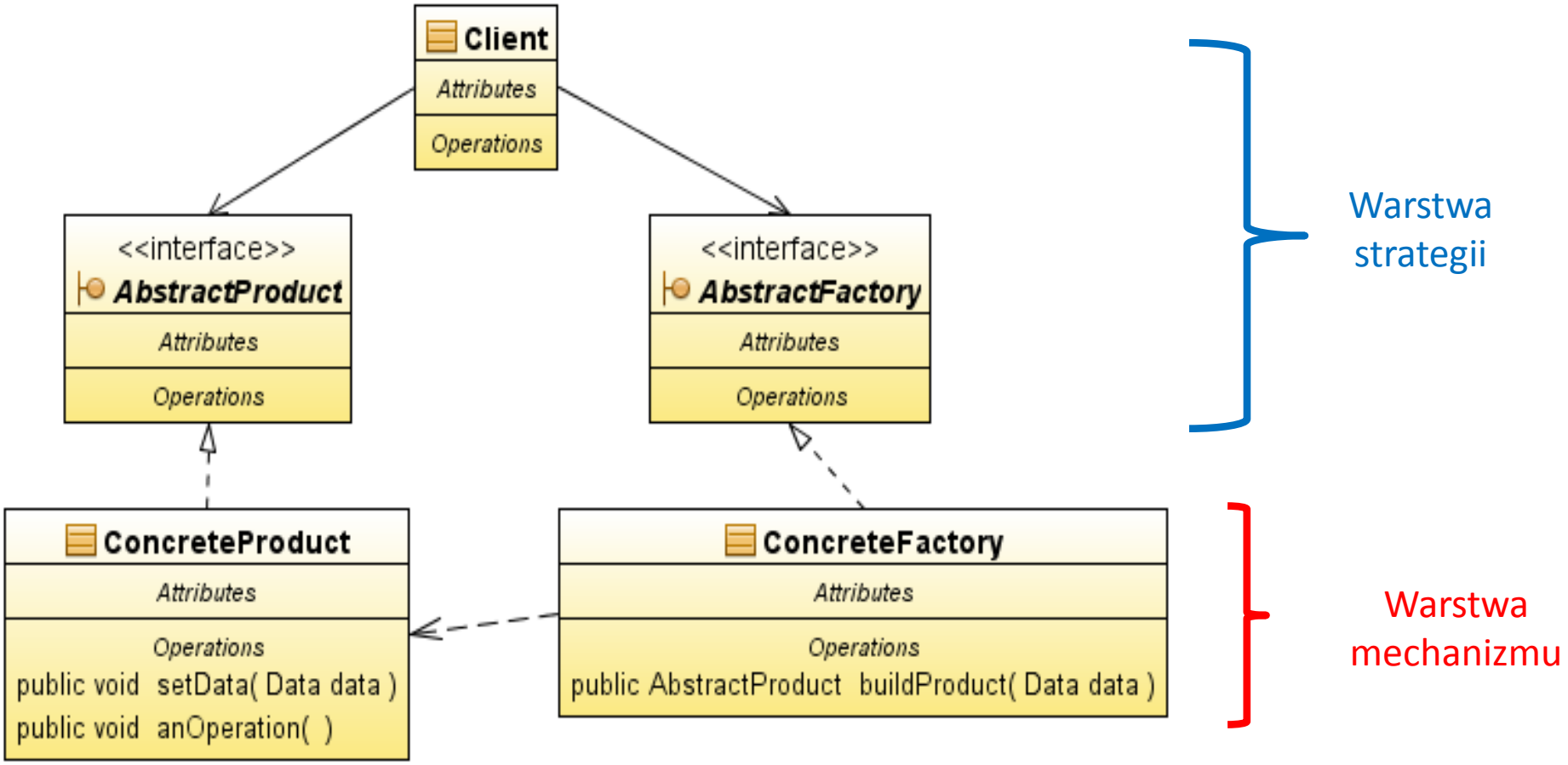
(4) Zasady procesu zwinnego

4) Zależność od abstrakcji – **zasada naiwna**

- Kod nie powinien być zależny od konkretnej klasy
  - Żadna zmienna referencyjna nie powinna być typu konkretnej klasy
  - Żadna klasa nie powinna dziedziczyć po konkretnej klasie
  - Żadna metoda nie powinna przedefiniowywać metody zdefiniowanej w klasie bazowej

(4) Zasady procesu zwinnego

Fabryka abstrakcyjna – *Abstract Factory*



(4) Zasady procesu zwinnego

Podsumowanie

- Kluczowy mechanizm niskiego poziomu
- Podwyższa odporność kodu na zmiany
- Prowadzi do tworzenia kodu wielokrotnego użycia

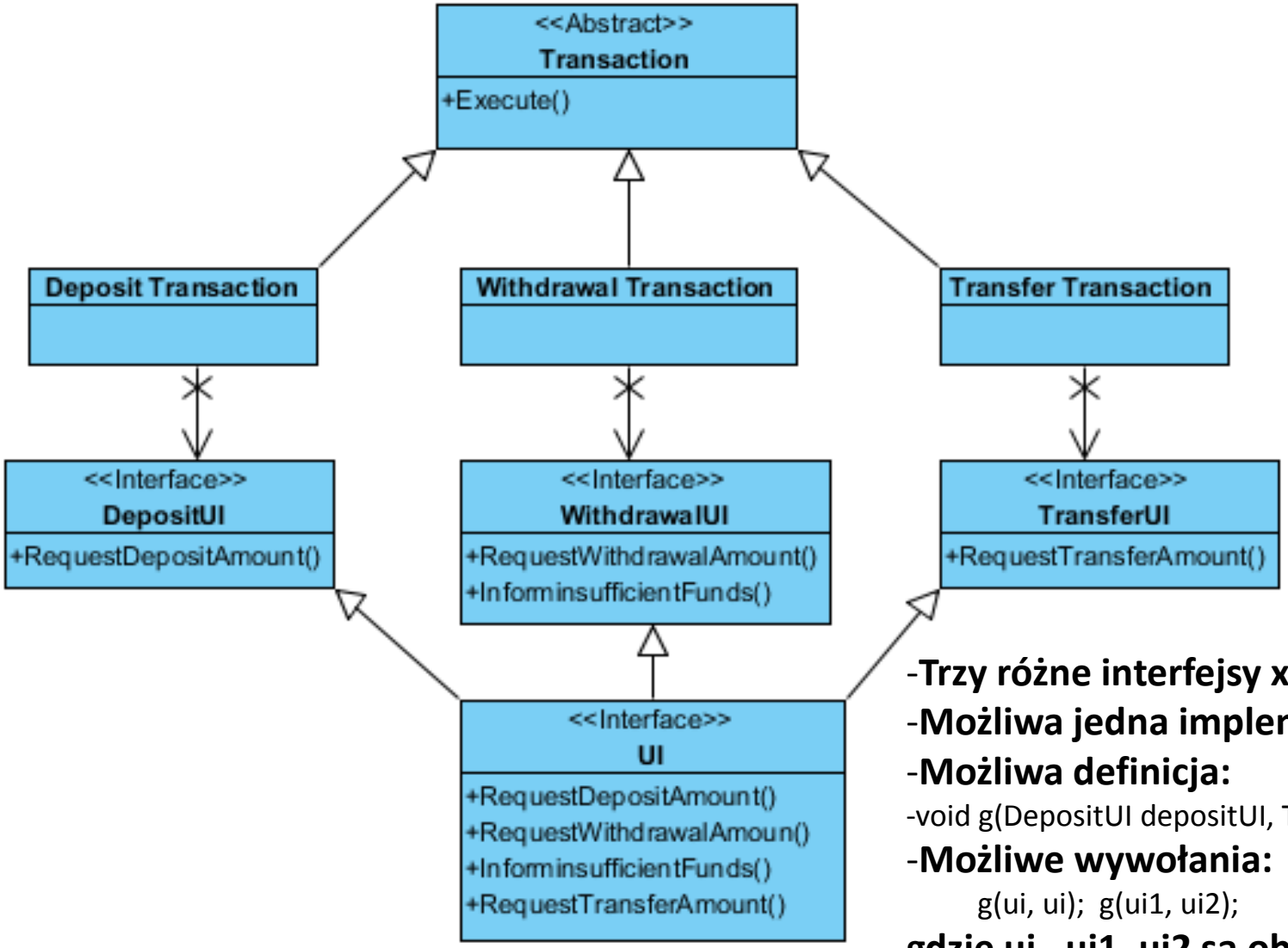
(5) Zasady procesu zwinnego

- **Zasada segregacji interfejsów (Interface Segregation Principle - ISP)**

  Klasa implementująca nie powinna być zmuszana do zależności od metod, których nie używa.

  1) Separacja przez implementowanie wielu interfejsów
  2) Dziedziczenie wielobazowe

(5) Zasady procesu zwinnego



-**Trzy różne interfejsy xxxUI**

-**Możliwa jedna implementacja UI**

-**Możliwa definicja:**

-void g(DepositUI depositUI, TransferUI transfer UI)

-**Możliwe wywołania:**

g(ui, ui); g(ui1, ui2);

gdzie ui , ui1, ui2 są obiektami klasy implementującej interfejs UI.

Programowanie ewolucyjne zwinne - cechy

Pojęcie *zwinnego programowania* zostało zaproponowane w 2001 w Agile Manifesto:

- **SCRUM**
- **XP**

Modele procesów tworzenia oprogramowania - przykłady

- Modele ewolucyjne zwinne
  - **Proces - programowanie zwinne** (*Agile software development*)
  - **Produkt – oprogramowanie obiektowe**

Manifest Zwinnego Tworzenia Oprogramowania z 2001 (http://agilemanifesto.org/iso/pl/)

Wytwarzając oprogramowanie i pomagając innym w tym zakresie, odkrywamy lepsze sposoby wykonywania tej pracy.

W wyniku tych doświadczeń przedkładamy:

Ludzi i interakcje ponad procesy i narzędzia.

Działające oprogramowanie ponad obszerną dokumentację.

Współpracę z klientem ponad formalne ustalenia.

Reagowanie na zmiany ponad podążanie za planem.

Doceniamy to, co wymieniono po prawej stronie, jednak bardziej cenimy to, co po lewej.

Kent Beck Mike Beedle Arie van Bennekum Alistair Cockburn	Ward Cunningham Martin Fowler James Grenning Jim Highsmith	Andrew Hunt Ron Jeffries Jon Kern Brian Marick

Zasady kryjące się za Manifestem Zwinnego Wytwarzania Oprogramowania

http://agilemanifesto.org/iso/pl/principles.html

Kierujemy się następującymi zasadami:

1. *Najważniejsze dla nas jest zadowolenie Klienta wynikające z wcześnie rozpoczętego i ciągłego dostarczania wartościowego oprogramowania.*
2. *Bądź otwarty na zmieniające się wymagania nawet na zaawansowanym etapie projektu. Zwinne procesy wykorzystują zmiany dla uzyskania przewagi konkurencyjnej Klienta.*
3. *Często dostarczaj działające oprogramowanie od kilku tygodni do paru miesięcy, im krócej tym lepiej z preferencją krótszych terminów.*
4. *Współpraca między ludźmi biznesu i programistami musi odbywać się codziennie w trakcie trwania projektu.*
5. *Twórz projekty wokół zmotywowanych osób. Daj im środowisko i wsparcie, którego potrzebują i ufaj im, ze wykonają swoją pracę.*
6. *Najwydajniejszym i najskuteczniejszym sposobem przekazywania informacji do i w ramach zespołu jest rozmowa twarzą w twarz .*

7. *Podstawową i najważniejszą miarą postępu jest działające oprogramowanie.*
8. *Zwinne procesy tworzą środowisko do równomiernego rozwijania oprogramowania. Równomierne tempo powinno być nieustannie utrzymywane poprzez sponsorów, programistów oraz użytkowników.*
9. *Poprzez ciągłe skupienie na technicznej doskonałości i dobremu zaprojektowaniu oprogramowania zwiększa zwinność.*
10. *Prostota – sztuka maksymalizacji pracy niewykonanej – jest zasadnicza.*
11. *Najlepsze architektury, wymagania i projekty powstają w samoorganizujących się zespołach.*
12. *W regularnych odstępach czasu zespół zastanawia się jak poprawić swoją efektywność, dostosowuje lub zmienia swoje zachowanie.*

Agile Manifesto (wersja źródłowa)

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Metodyka typu Agile

Scrum

- **sprint** (iteracja – (2-4 tygodnie),
- ***scrum meeting*** (spotkanie codziennie 15 min),
- ***sprint review*** *(podsumowanie sprintu)*)

Podstawowe zadanie metodyki:

- dostarczaniu kolejnych, coraz bardziej dopracowanych produktów,
- włączaniu się przyszłych użytkowników w proces wytwórczy,
- samoorganizacji zespołu wykonawców.

Role główne

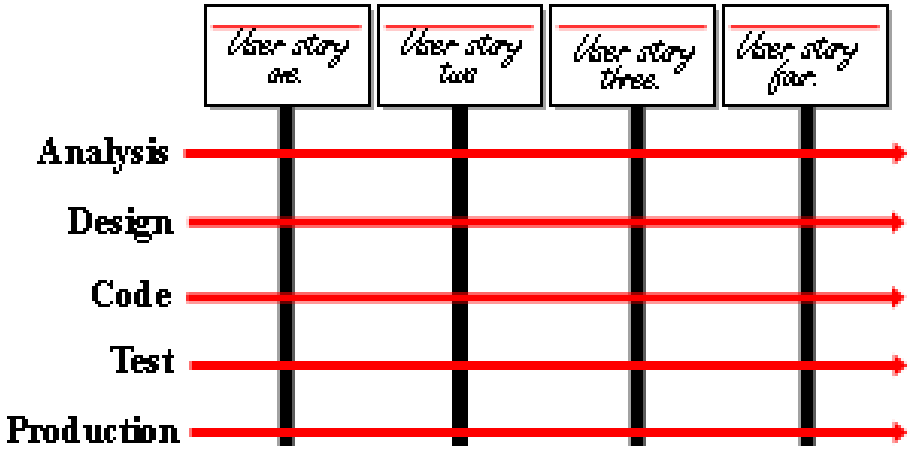
- **Zespół** - 5-9 osób, zaangażowana w tworzenie oprogramowania
- **Właściciel produktu** (**Product owner**) - osoba reprezentująca klienta, która może należeć do zespołu
- **Kierujący zespołem** **(Scrum master)** – osoba ułatwiająca działania zespołu

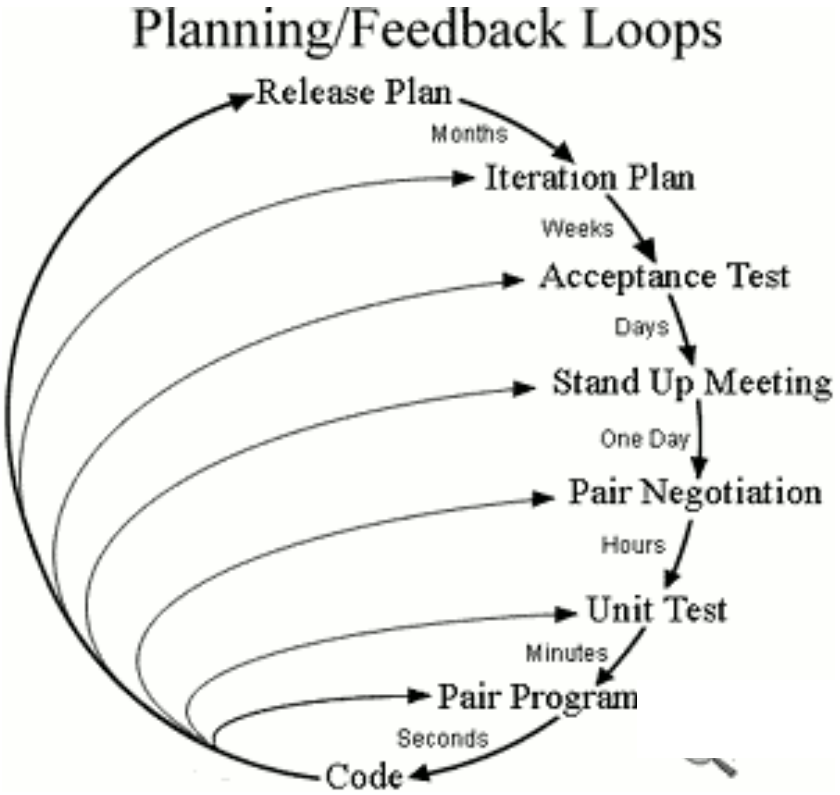
Modele procesów tworzenia oprogramowania - przykłady

- Modele ewolucyjne zwinne
  - Proces - programowanie ekstremalne XP(Extreme Programming) - praktyki XP
  - Produkt - oprogramowanie obiektowe

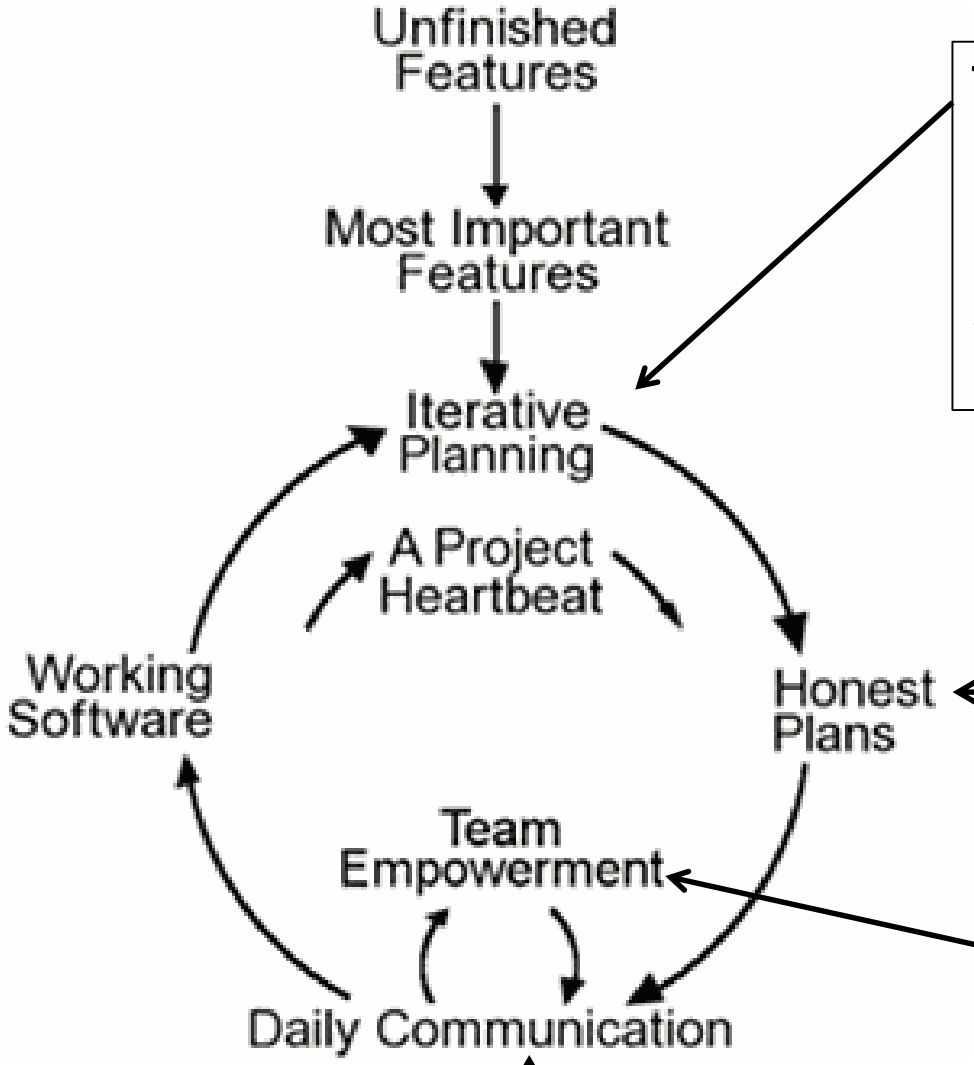
XP podejście do iteracyjno-rozwojowego tworzenia oprogramowania

http://www.agile-process.org





Cykl życia oprogramowania XP - http://www.agile-process.org



Trzy poziomy planowania:

1. Planowanie wydania obejmujące kilka iteracji
2. Planowanie pojedynczej iteracji (sprint)
3. Planowanie obejmujące 1-dniowy zakres pracy (fragment iteracji)

Rzetelny plan obejmujący zakres prac w ciągu 1 tygodnia (SCRUM – 30 dni). Plan obejmuje elastyczne wykorzystanie modelowania, projektowania i implementacji

Elastyczne zarządzanie zespołem (sieciowe, specjalistyczne, nieegoistyczne)

Wynika z postępu prac określonych w planach jednodniowych

Praktyki XP [2]

- Planowanie:
  - zespół planuje czas i budżet,
  - zespół planuje ryzyko
  - zespół planuje kolejność opowiadań (opis działania systemu wykonany przez klienta lub wykonawcę)
  - klient definiuje:
    - Zakres działań
    - Terminy ukończenia wersji
    - priorytety
- Metafora systemu – sposób działania systemu
- Prosty projekt – do testowania zakresu działań
- Programowanie parami – przy jednej stacji roboczej (cały zespół około 10 osób)
- Testy jednostkowe i akceptacji – przed i po wykonaniu właściwego kodu
- Refaktoryzacja – w ciągu tworzenia i rozwijania kodu

Praktyki XP [2]

- Uwspólnienie kodu - przez parokrotne przypisywanie zespołom różnych zadań związanych z innymi zadaniami, każdy wykonawca zna obraz całego kodu
- Ciągła integracja kodu
- Przedstawiciel klienta jako członek wykonawców (testy akceptacji, ekspert domen)
- 40-godzinny tydzień pracy – ciągła aktywność zespołu wykonawców
- „Małe wersje”: tworzone są wersje niewielkie z przydatnymi funkcjami
- Standardy kodowania – najpierw definiowane, a potem stosowane

Cechy

- Poszczególne elementy wzajemnie się wspierają
- Zachowanie kontroli nad procesem
- Ogranicza wstępne zbieranie danych o wymaganiach
- Ogranicza analizę i modelowanie projektu
- Ogranicza planowanie na rzecz późniejszej elastyczności – ogranicza liczbę klas i dokumentacji
- Wydajne tworzenie małych i średnich "projektów wysokiego ryzyka” oparte na synergii stosowania rozmaitych praktyk zapewniające eliminację wad i wykorzystania zalet tych praktyk.

Kwestie kontrowersyjne

- Brak dokładnej specyfikacji.
- Stałe angażowanie strony klienta.
- Zbyt swobodne zmiany kodu