Obiekty, aspekty, komponenty

Bartosz Walter

Rozdział zagadnień

Zasada rozdziału zagadnień (*Separation of Concerns)*

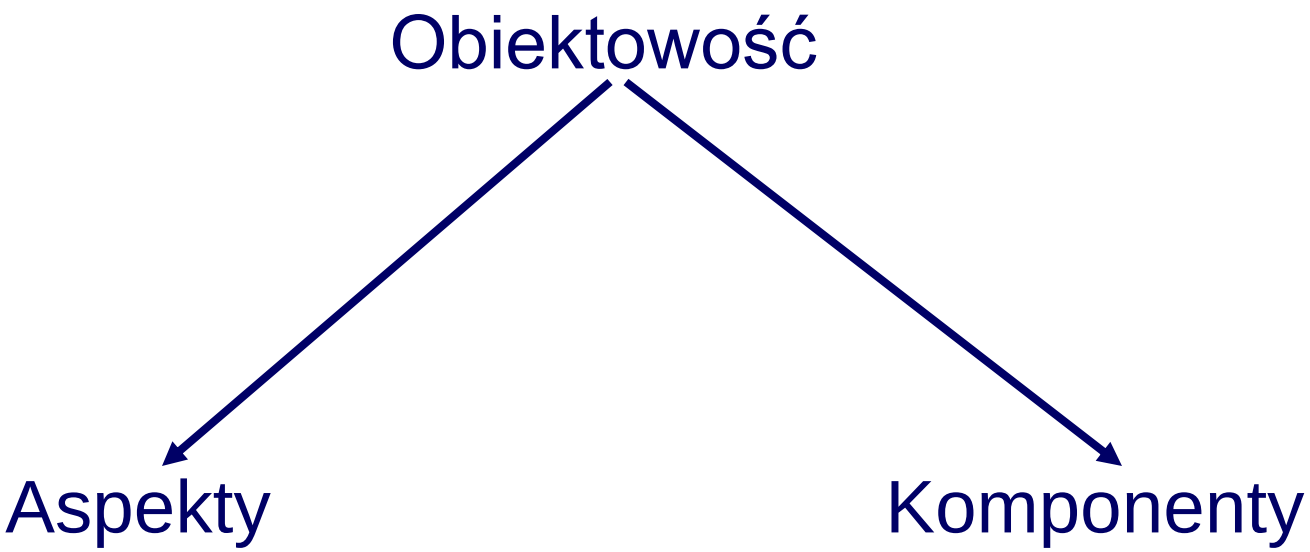
- program powinien być zdekomponowany w taki sposób, aby różne jego aspekty znajdowały się w dobrze odseparowanych od siebie modułach
- każdy aspekt zajmuje się jednym zagadnieniem
- zagadnienie to szczególny cel programu, koncepcja albo funkcja

D. Parnas "On the criteria to be used in decomposing systems..." (1972)

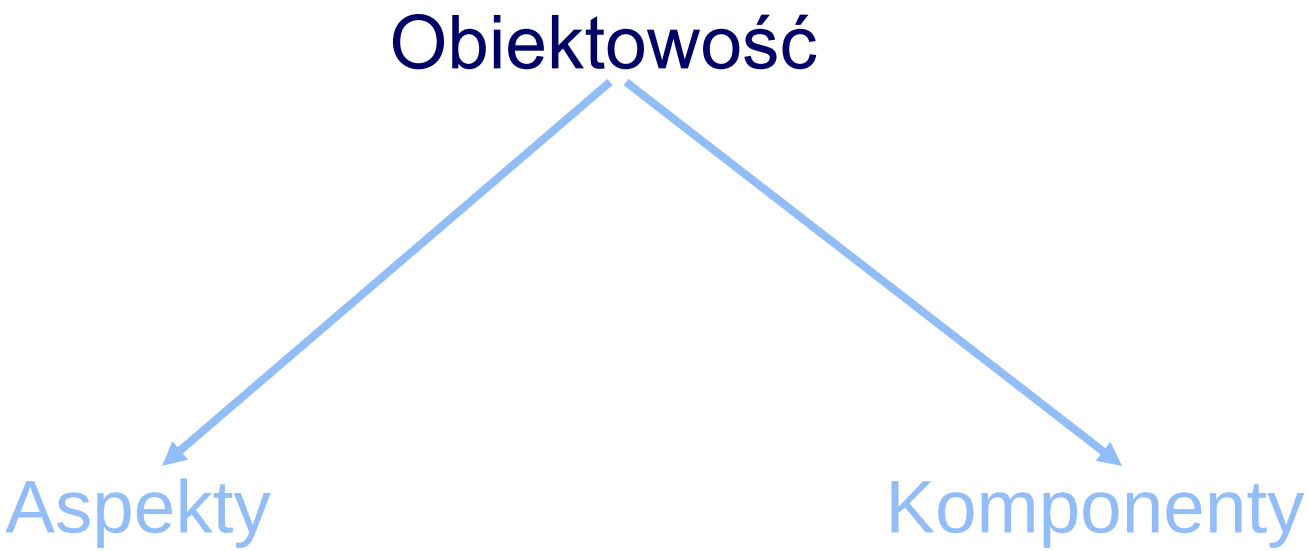
G. Polya "How to Solve It?" (1973)

E. Dijkstra "A Discipline of Programming" (1976)

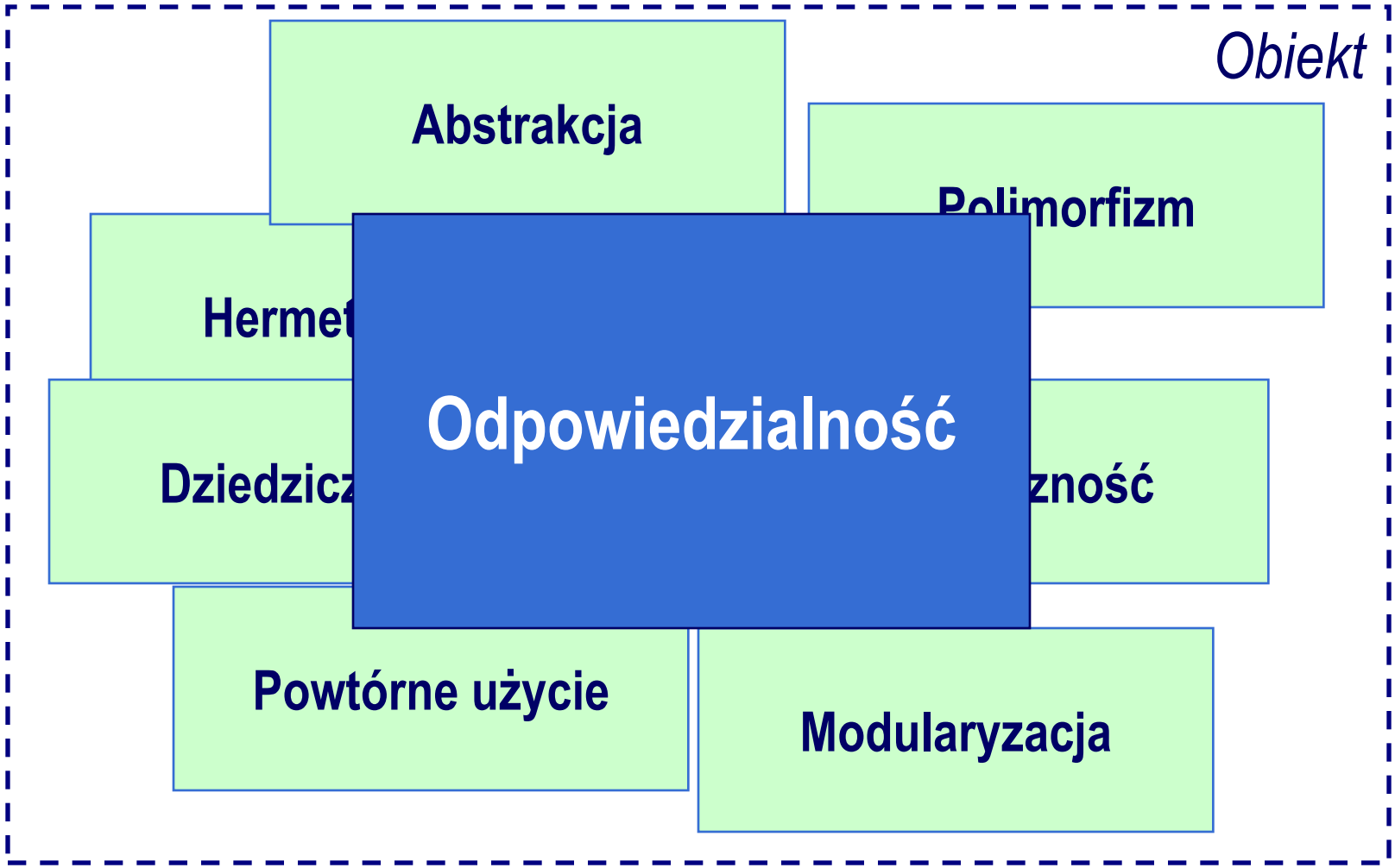
Plan wykładu



Plan wykładu



Obiektowość



Czym jest obiekt?

Definicja 1

Obiekt w reprezentuje element świata rzeczywistego;
jest strukturą posiadającą tożsamość, stan (pola) i zachowanie (metody).

Definicja 2

Obiekt w pełni odpowiada za pewien fragment świata rzeczywistego.
Sposób realizacji tej odpowiedzialności zależy od samego obiektu .

Obiektowość a podejście strukturalne

Przykład

Wykładowca przeprowadza egzamin. W zależności od oceny oraz rodzaju egzaminu, za który została wystawiona, student zalicza przedmiot, podchodzi do poprawki lub prosi o zaliczenie warunkowe.

Wykładowca powinien każdemu studentowi, w zależności od jego indywidualnej sytuacji, przekazać wskazówki dotyczące dalszego postępowania.

na podstawie przykładu A. Shallowaya i J. Trotta

Obiektowość a podejście strukturalne

Rozwiązanie 1

1. Utwórz listę studentów i ich ocen
2. Dla każdego studenta z listy
   a) określ jego ocenę
   b) określ dalszy sposób postępowania studenta
   c) przekaż informacje studentowi

Obiektowość a podejście strukturalne

Rozwiązanie 2

Wykładowca

1. Opublikuj listę studentów i ich ocen
2. Opublikuj informację o sposobie dalszego postępowania w zależności od oceny

Student

1. Znajdź swoją ocenę na liście udostępnionej przez Wykładowcę
2. Określ sposób postępowania na podstawie oceny
3. Postąp zgodnie z instrukcją

Obiektowość a podejście strukturalne

Pojawiają się nowe wymagania...

- Studenci powtarzający przedmiot nie mają prawa do egzaminu warunkowego
- Studenci niepełnosprawni mogą poprosić o jeszcze jeden termin egzaminu
- Studenci z wymiany zagranicznej nie muszą zdawać tego egzaminu, mogą wybrać inny
- ...

Obiektowość a podejście strukturalne

Wnioski

Podział odpowiedzialności pozwolił

- uprościć algorytm
- zmniejszyć powiązania między obiektami
- zwiększyć otwartość systemu na zmiany

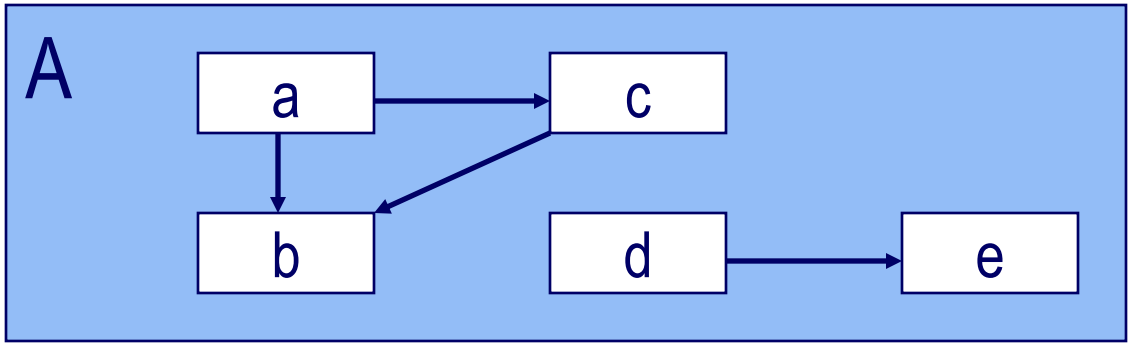
Opis klasy

Klasa Nauczyciel	
Odpowiedzialność • zarządzanie kolekcją • udostępnianie ocen uczniom	**Współdziałanie** • Kolekcja: dodawanie i usuwanie ocen • Uczeń: udostępnianie ocen

Spójność obiektu

Spójność obiektu (ang. *cohesion*)

- opis współpracy elementów obiektu
- stopień powiązania metod i pól obiektu przy wypełnieniu nałożonej na niego odpowiedzialności



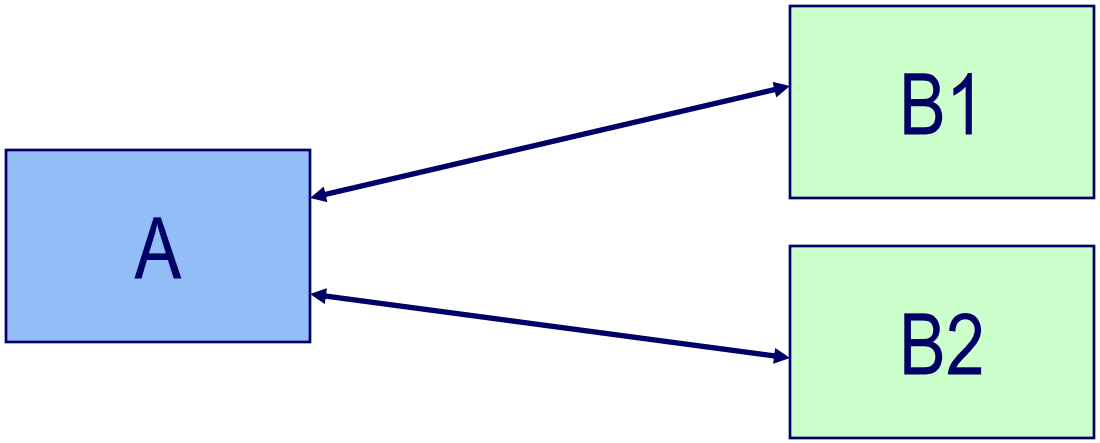
Wniosek

Prawidłowo zaprojektowana klasa jest spójna

Powiązania między obiektami

Powiązania między obiektami (ang. *coupling*)

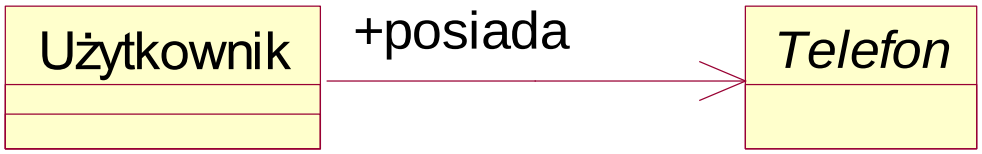
- opis zależności pomiędzy obiektami
- stopień powiązania obiektów, które nie są spokrewnione



Wniosek

Niski stopień powiązań wspiera abstrakcję i hermetyzację w systemie

Asocjacja



- *Telefon* należy do *Użytkownika*
- *Telefon* może zmienić *Użytkownika*, *Użytkownik* może zmienić *Telefon*
- *Użytkownik* zna swój *Telefon*, *Telefon* nie wie, kto jest jego właścicielem
- Istnienie *Użytkownika* nie ma wpływu na istnienie *Telefonu* i vice wersa

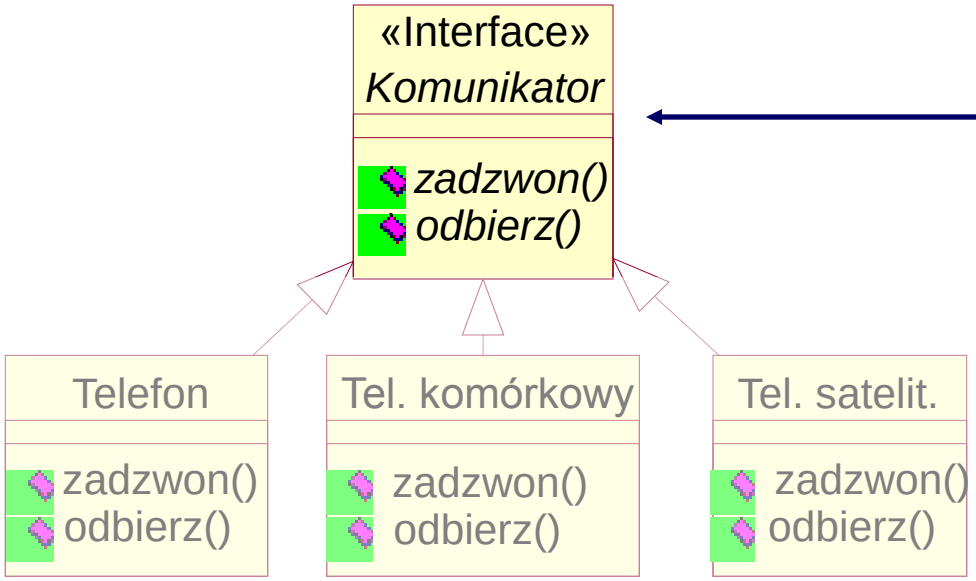
Agregacja

- *Katalog* zawiera *Książki*
- *Książka* może należeć do wielu *Katalogów* jednocześnie
- Istnienie *Katalogu* nie ma wpływu na istnienie *Książki* i vice versa

Kompozycja

- *Książka* składa się z *Rozdziałów*, *Rozdział* jest częścią *Książki*
- *Rozdział* może należeć tylko do jednej *Książki*
- Istnienie *Książki* decyduje o istnieniu jej *Rozdziałów*

Interfejsy



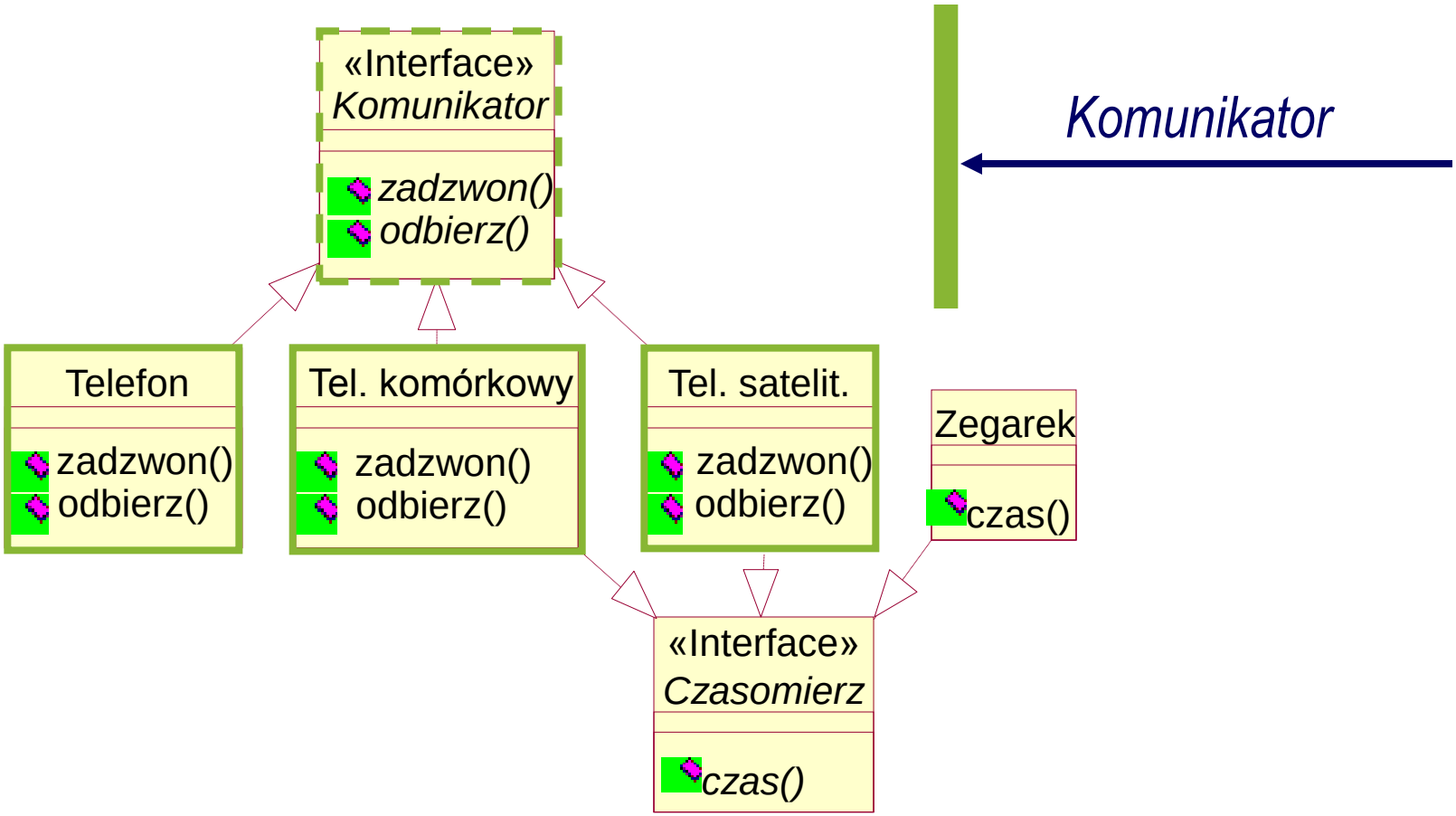
komunikator.zadzwon()

Wnioski

Interfejs mówi **CO** robi klasa, a nie **W JAKI SPOSÓB**.

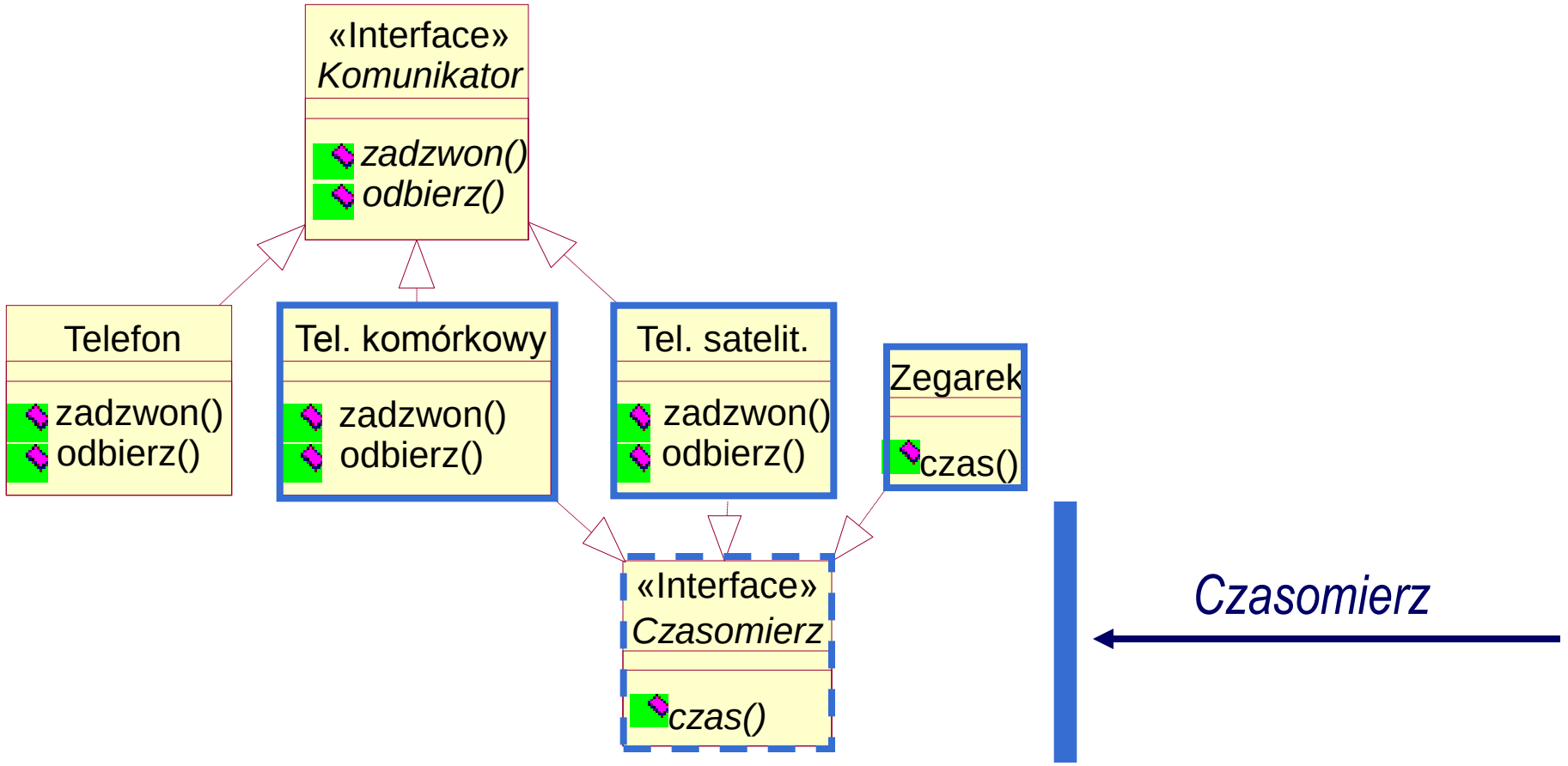
Interfejs opisuje **ROLĘ** obiektu.

Wielokrotne interfejsy



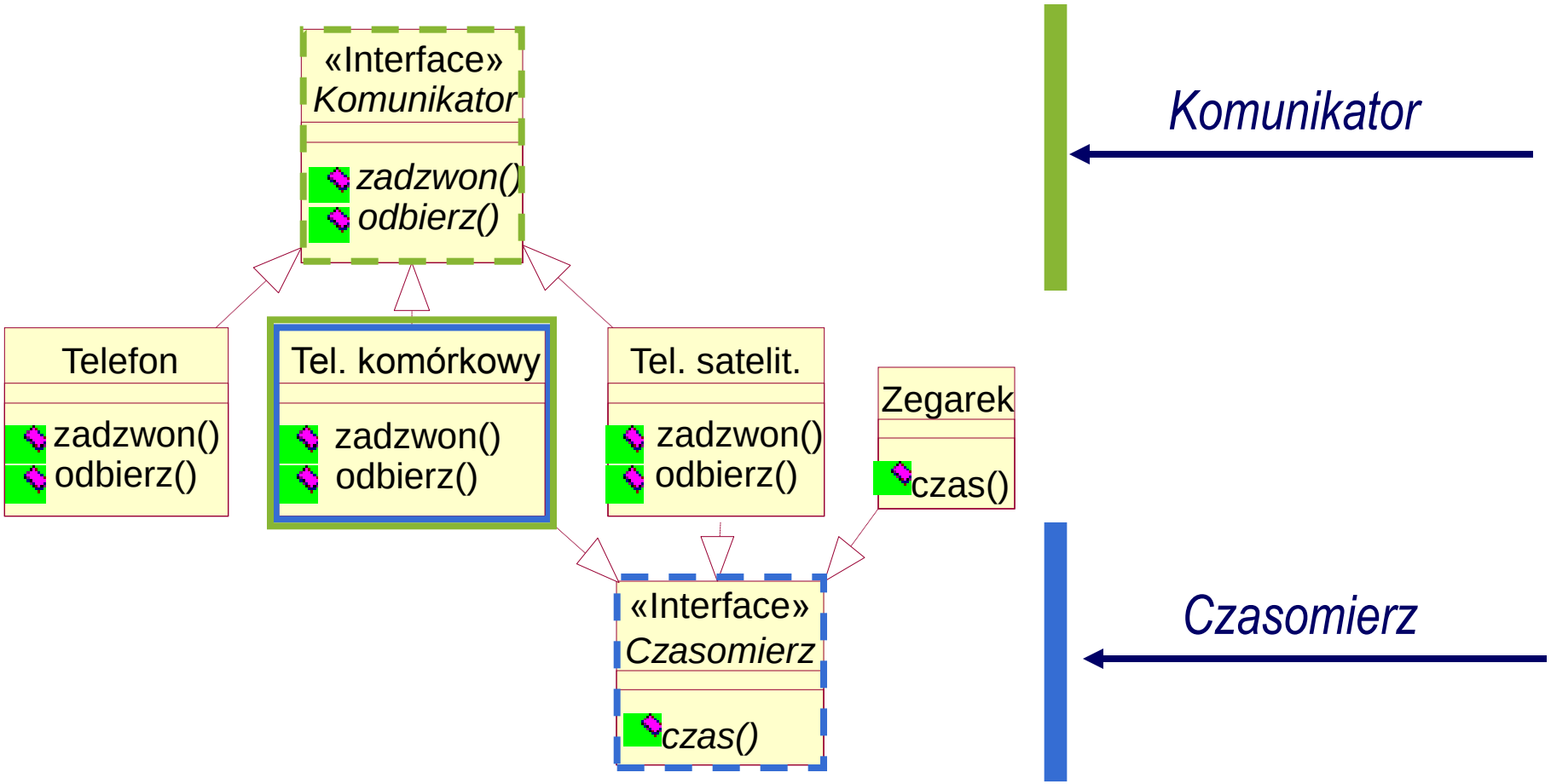

```
Komunikator komunikator = new TelefonKomorkowy();
komunikator.zadzwon();
```

Wielokrotne interfejsy




```
Czasomierz czasomierz = new Zegarek();
czasomierz.czas();
```

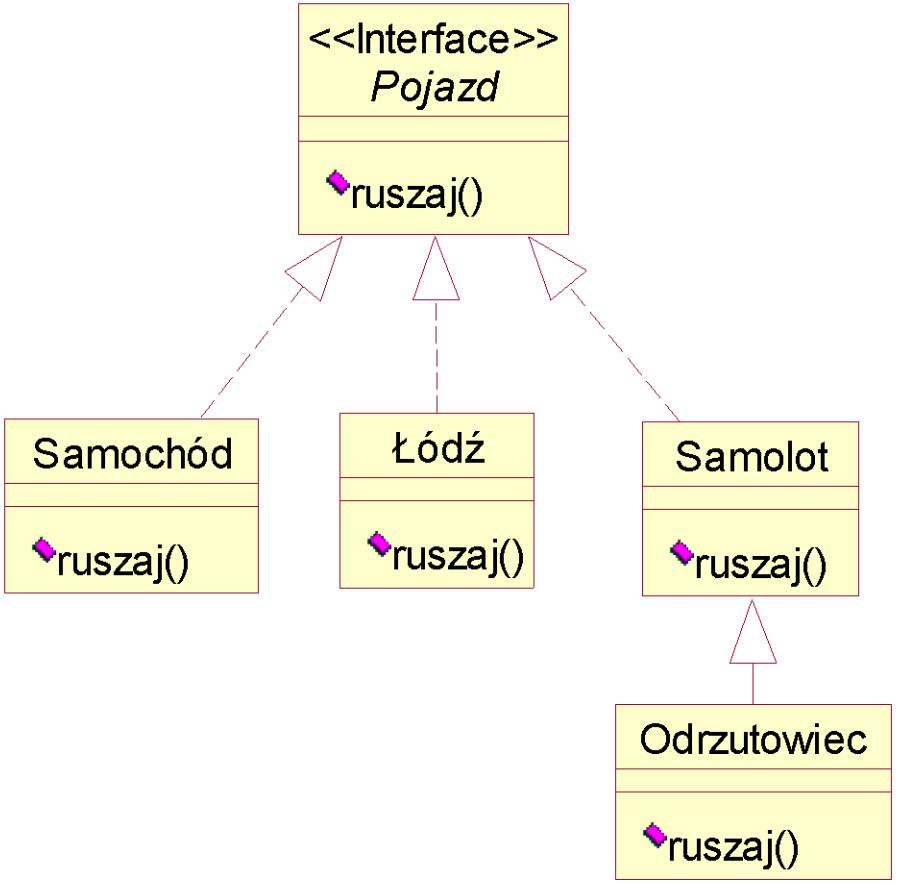
Wielokrotne interfejsy



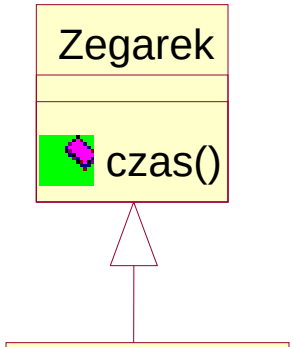

```
Komunikator komorka = new TelefonKomorkowy();
Czasomierz komorka = new TelefonKomorkowy();
```

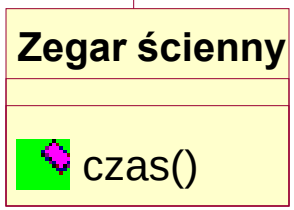
Dziedziczenie

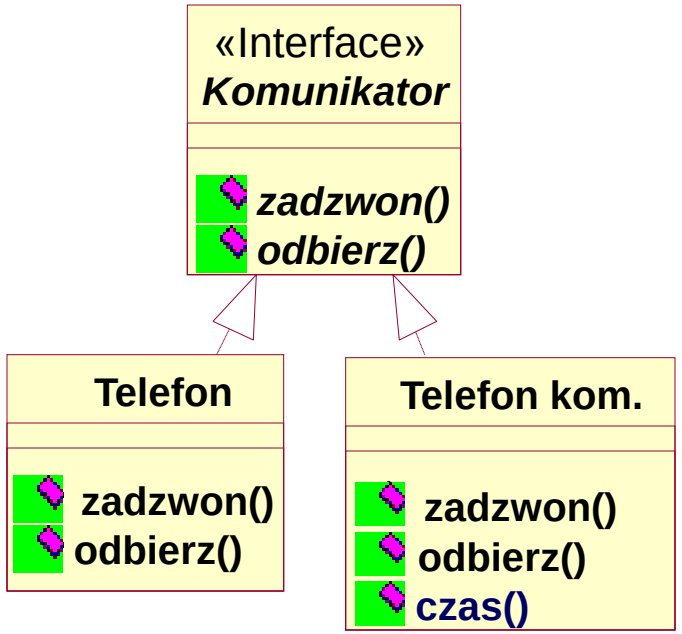
- *Odrzutowiec jest rodzajem Samolotu*
- *Odrzutowiec* może zastąpić *Samolot*
- *Odrzutowiec* posiada niejawną instancję *Samolotu*
- związek między *Odrzutowcem* i *Samolotem* jest nierozerwalny



Dziedziczenie vs. użycie interfejsu



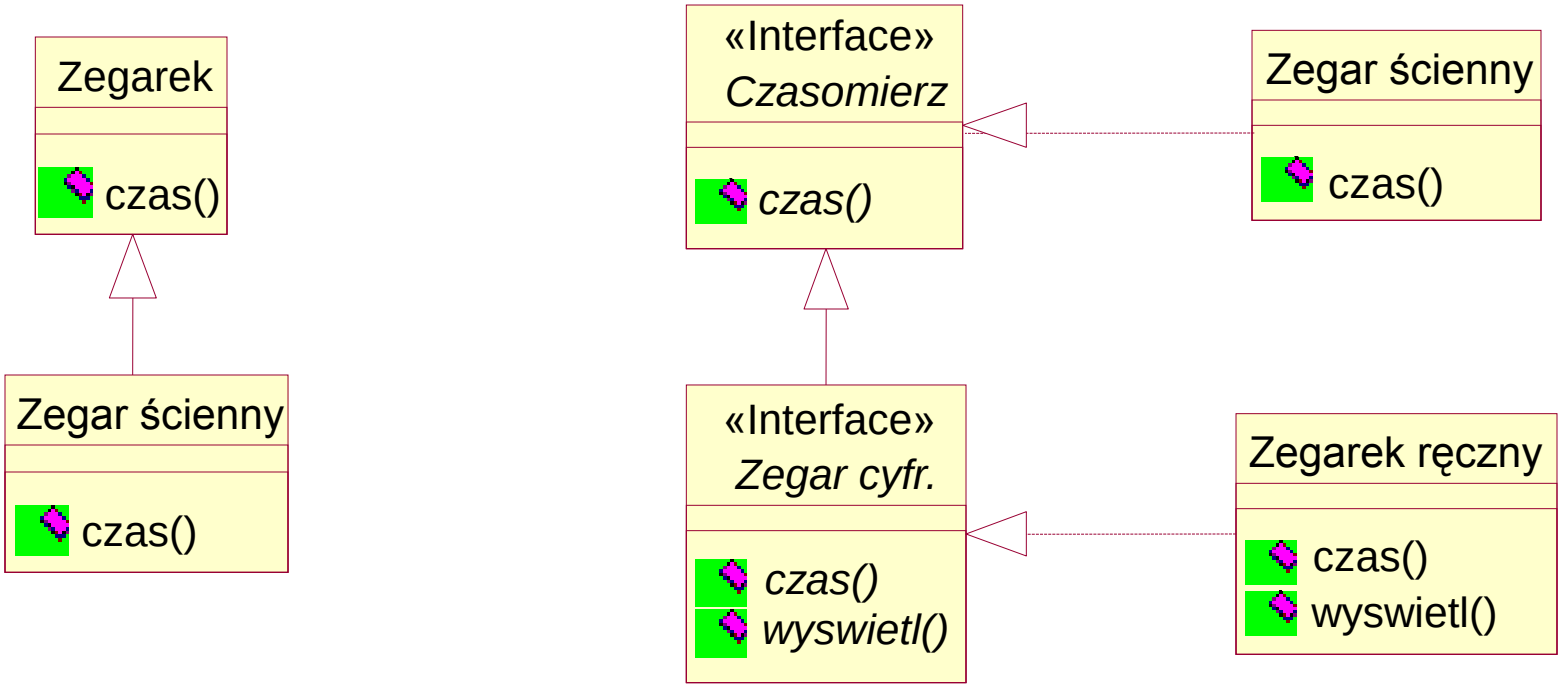




- **Zegar ścienny** jest rodzajem **Zegarka**. Posiada wszystkie jego cechy i działa tak samo
- **Zegar ścienny** dziedziczy zarówno typ, jak i implementację

- **Telefon i Telefon kom.** są **Komunikatorami**. Oba obiekty mogą inicjować i odbierać rozmowy
- **Telefon** i **Telefon kom.** współdzielą tylko typ

Dziedziczenie klas vs. dziedziczenie interfejsów



Wnioski

Dziedziczenie klas przekazuje podklasie typ i implementację

Dziedziczenie interfejsów dotyczy tylko typu

Hermetyzacja

Definicja 1

Hermetyzacja oznacza ukrywanie danych przed niepożądanym dostępem

Definicja 2

Hermetyzacja oznacza ukrywanie każdej decyzji projektowej, która może ulec zmianie: interfejsu, implementacji, zachowania metody, danych

Wniosek

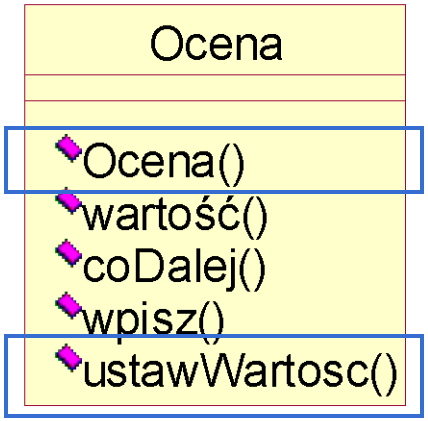
Należy identyfikować zmienność w systemie i **hermetyzować ją**

Hermetyzacja danych (1)

Przykład

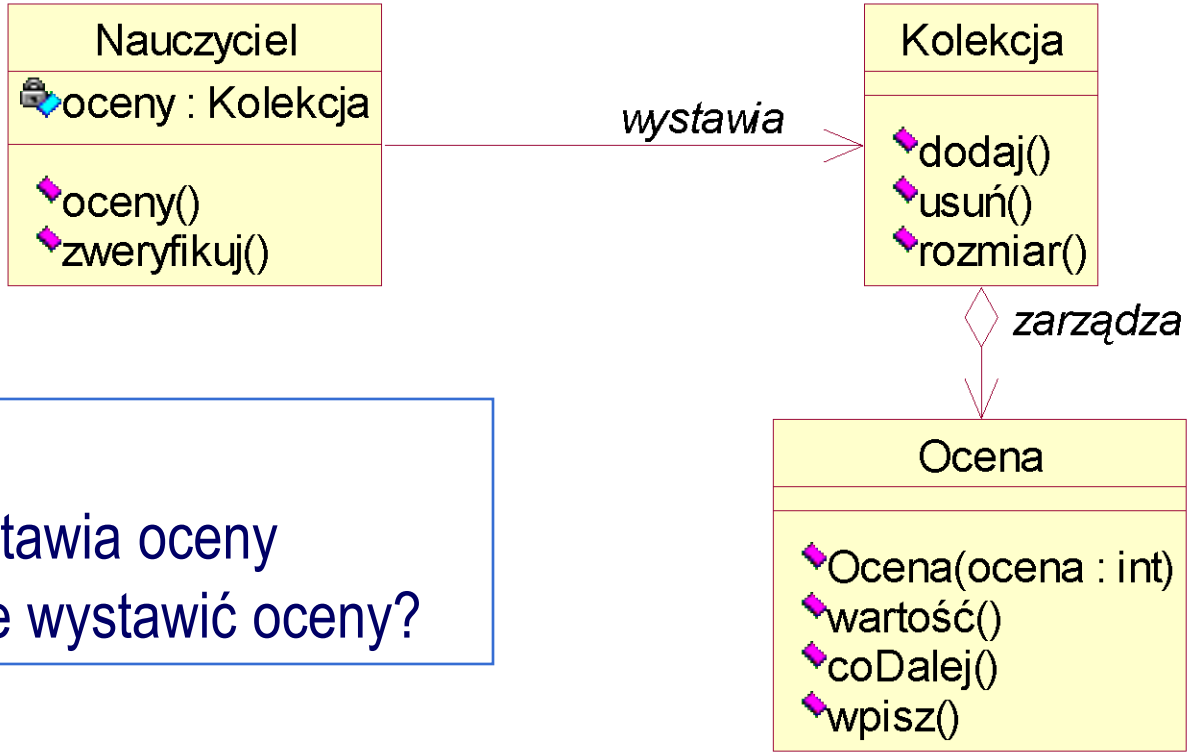
Ocena może zmienić swoją wartość

Konstruktor tworzy ułomny obiekt




```
Ocena ocena = new Ocena();
ocena.ustawWartosc(Ocena.BDB);
```

Hermetyzacja danych (2)



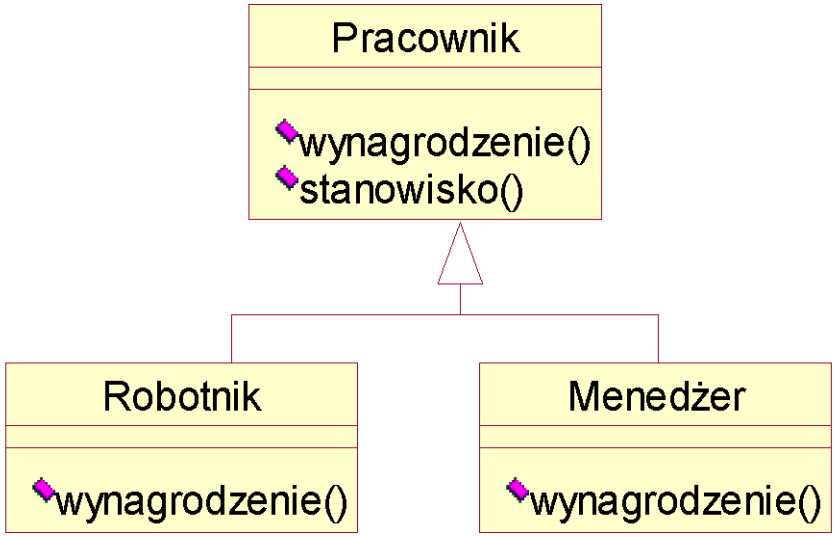
Przykład

Wykładowca wystawia oceny

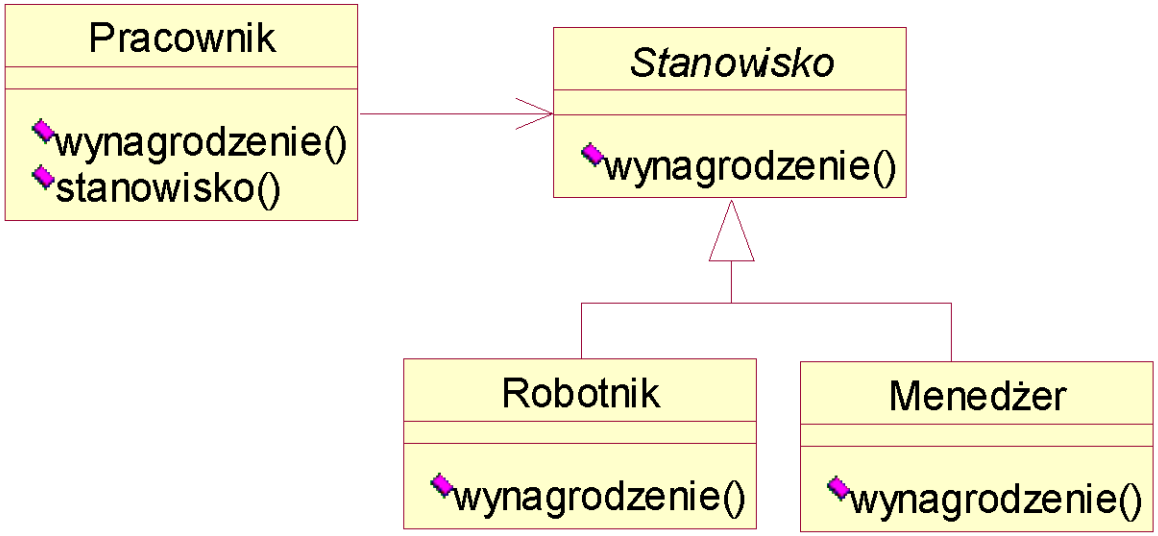
Kto jeszcze może wystawić oceny?

```
Kolekcja oceny = nauczyciel.oceny();
oceny.dodaj(new Ocena(Ocena.BDB));
```

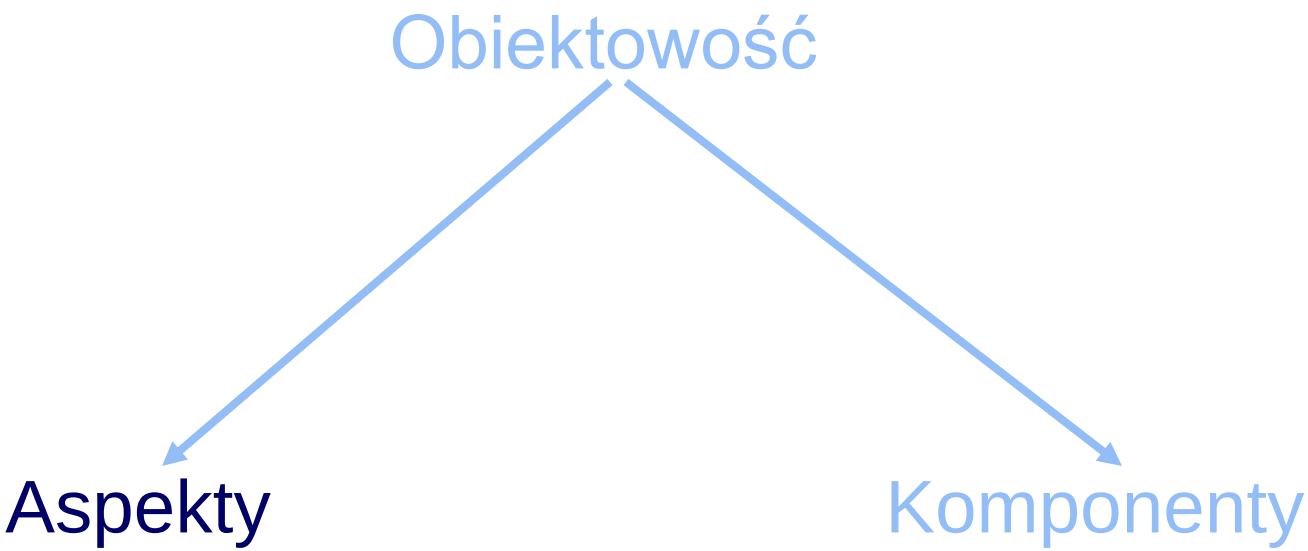
Oceń jakość tego projektu



Rozwiązanie



Plan wykładu



Rozdział zagadnień

Zasada rozdziału zagadnień (*Separation of Concerns)*

- program powinien być zdekomponowany w taki sposób, aby różne jego aspekty znajdowały się w dobrze odseparowanych od siebie częściach systemu
- każdy aspekt zajmuje się jednym zagadnieniem
- zagadnienie to szczególny cel programu, koncepcja albo funkcja

D. Parnas "On the criteria to be used in decomposing systems..." (1972)

G. Polya "How to Solve It?" (1973)

E. Dijkstra "A Discipline of Programming" (1976)

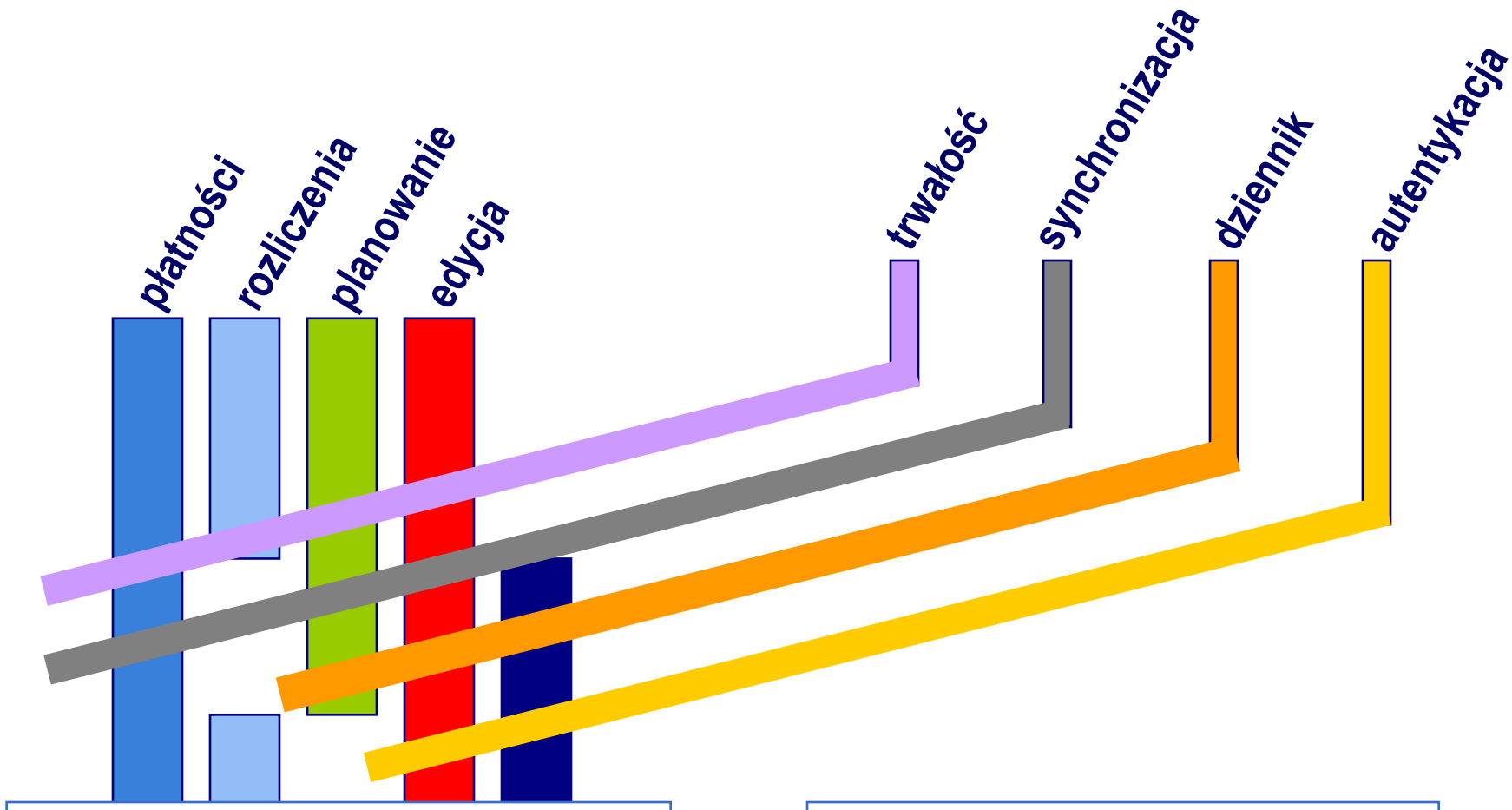
Nisza aspektowa

W programowaniu obiektowym

- system jest zbiorem współdziałających, samodzielnych i odpowiedzialnych jednostek – obiektów
- klasy ukrywają implementację, prezentując jedynie interfejsy
- polimorfizm jest podstawowym mechanizmem łączenia podobnych koncepcji

...ale jak prawidłowo zaimplementować cechy, które przecinają wiele niespokrewnionych ze sobą obiektów?

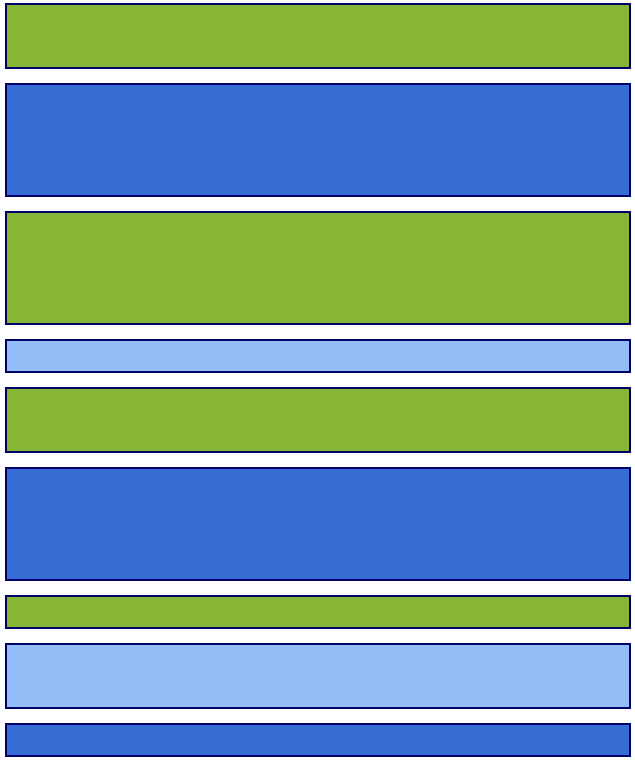
Podział wymagań



Wymagania podstawowe
(biznesowe)

Wymagania systemowe
(techniczne)

Modularyzacja kodu



program obiektowy

program aspektowy

OOP a AOP

Programowanie obiektowe

- grupowanie podobnych koncepcji za pomocą hermetyzacji i dziedziczenia
- podstawowa jednostka modularyzacji: klasa

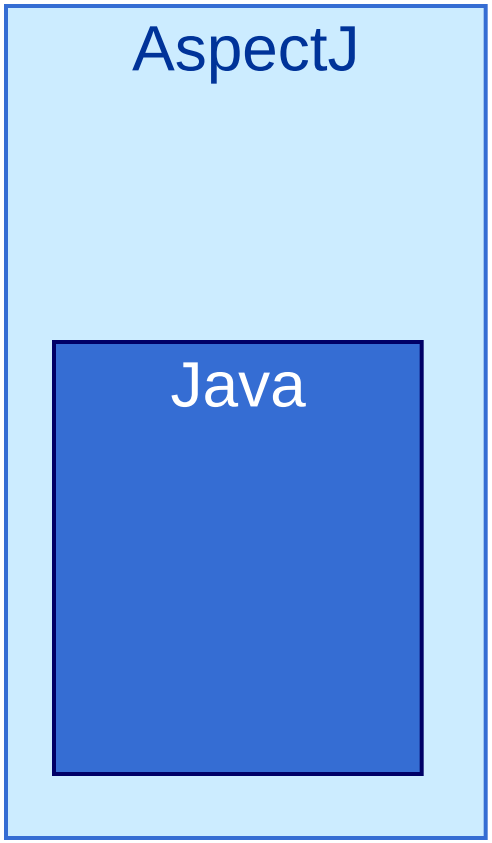
Programowanie aspektowe

- grupowanie podobnych koncepcji w niezwiązanych ze sobą klasach, niezależnie od dziedziczenia
- dodatkowy mechanizm modularyzacji: aspekt

Język AspectJ

AspectJ

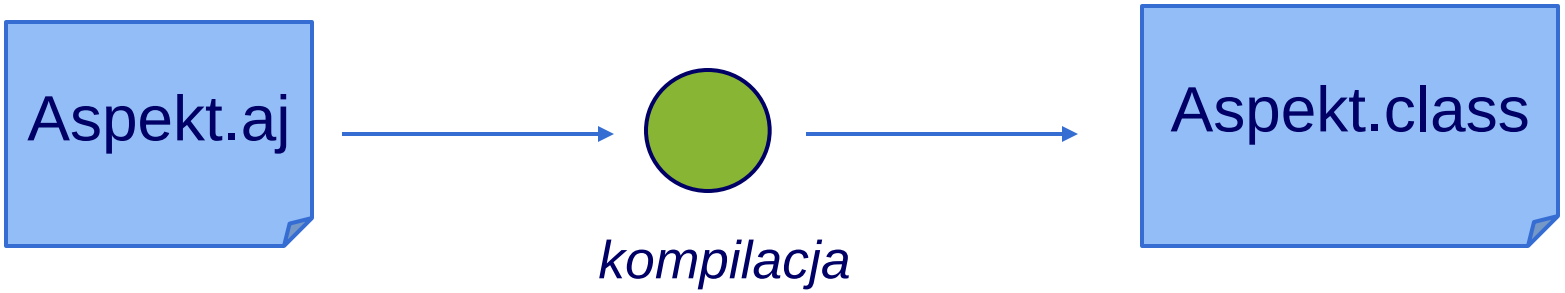
- G. Kiczales (2001), Xerox Palo Alto Research Center
- aspektowe rozszerzenie Javy
- aspekt jako specyficzna klasa
- możliwość zmiany zachowania i struktury kodu
- łączenie aspektów i klas na poziomie bajtkodu
- własny kompilator *ajc*



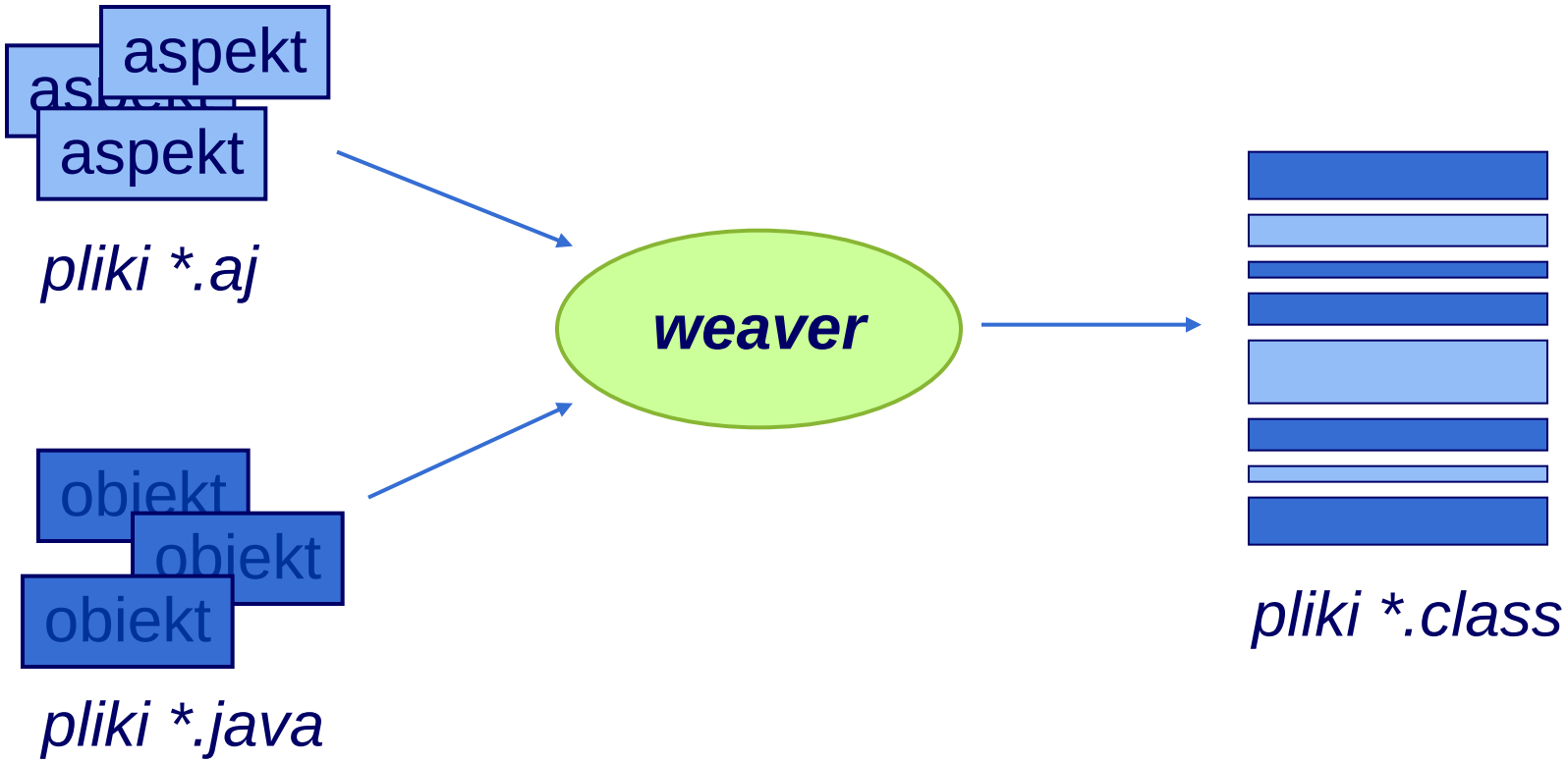
Aspekt

Aspekt (w języku AspectJ)

- specjalizowana klasa, która może przecinać inne klasy
- podlega dziedziczeniu
- jednostka modularyzacji (obok klasy)
- posiada typowe elementy klasy
- łączy punkty cięcia i porady



Zasada działania AspectJ



Prosty aspekt w AspectJ

```
public class HelloWorld {
   public static void hej(String tekst) {
      System.out.println(tekst);
   }
   public static void hejKolego(String tekst, String imie) {
      System.out.println(imie + ", " + tekst);
   }
   public static int hejLudzie(String tekst, String[] imiona) {
      for (int i = 0; i < imiona.length; i++)
         System.out.println(imiona[i] + ", " + tekst);
      return imiona.length;
   }
}
```

na podstawie przykładu R. Laddada (2002)

Prosty aspekt w AspectJ

```
public aspect Maniery {
    pointcut wywolanieHej () :
        call(public static void HelloWorld.hej*(..));

    before() : wywolanieHej() {
        System.out.println("Dzień dobry!");
    }

    after() : wywolanieHej() {
        System.out.println("Dziękuję!");
    }
}
```

na podstawie przykładu R. Laddada (2002)

Punkty złączeń

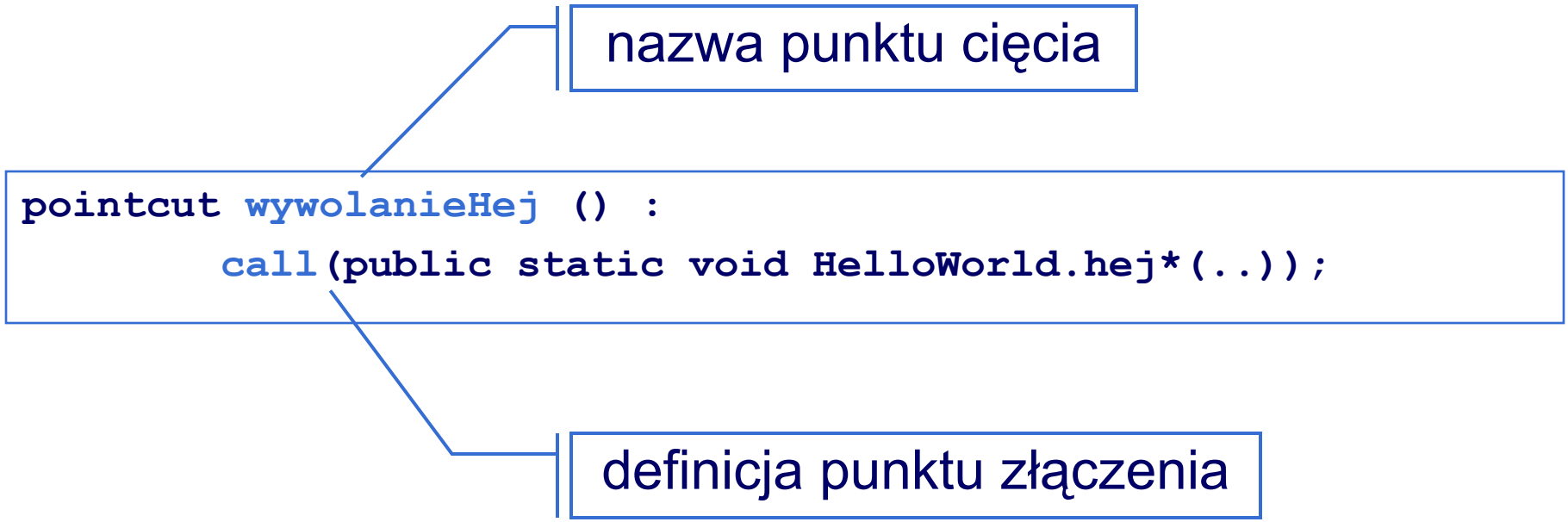
Punkty złączenia (ang. *joinpoints*) są dowolnymi, identyfikowalnymi miejscami w programie.

Przykłady:

- wywołanie metody i konstruktora
- wykonanie metody i konstruktora
- dostęp do pola
- obsługa wyjątku
- statyczna inicjacja klasy

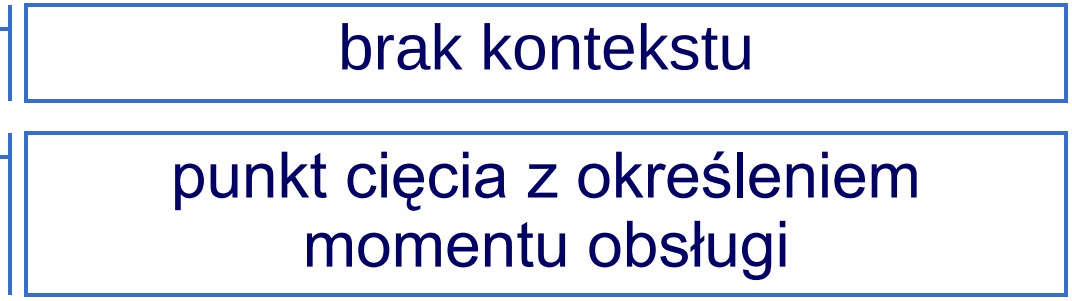
Punkty cięcia

Punkt cięcia (ang. *pointcut*) jest zdefiniowaną kolekcją punktów złączenia.



Porada

Porada (ang. *advice*) jest fragmentem kodu programu wykonywanym przed, po lub zamiast osiągnięcia przez program punktu cięcia.




```
before() : call (public * Klasa.metoda(..)) {
    // kod porady, który będzie wykonany przed
    // wywołaniem metody metoda w klasie Klasa
}
```

Rodzaje porad

Rodzaje porad w AspectJ

- **before()**

  wykonywana tuż przed punktem cięcia
- **after() {returning | throwing}**

  wykonywana tuż po punkcie cięcia (poprawnym lub zgłaszającym wyjątek)
- **around(context)**

  otaczająca punkt cięcia i decydująca o jego wykonaniu lub nie

Przykład: aspekt logujący wywołania metod

```
public aspect Logger {
  pointcut wywolanieMetody() :
    call(* * (..)) && ! within(Logger);

  before() : wywolanieMetody() {
    System.out.println("Przed wywołaniem metody "
      + thisJoinPoint.getSignature());
  }
  after() returning : wywolanieMetody() {
    System.out.println("Po wywołaniu metody "
      + thisJoinPoint.getSignature());
  }
  after() throwing  : wywolanieMetody() {
    System.out.println("Metoda " +
      thisJoinPoint.getSignature() + " zgłosiła wyjątek");
  }
}
```

Przykład: do czego służy ten aspekt?

Jaki jest efekt użycia tego aspektu?

```
aspect A {
   before(): call(* *(..)) {
      System.out.println("przed");
   }
   after(): call(* *(..)) {
      System.out.println("po");
   }
}
```

Przykład: wersja 2

Jaki jest efekt użycia tego aspektu?

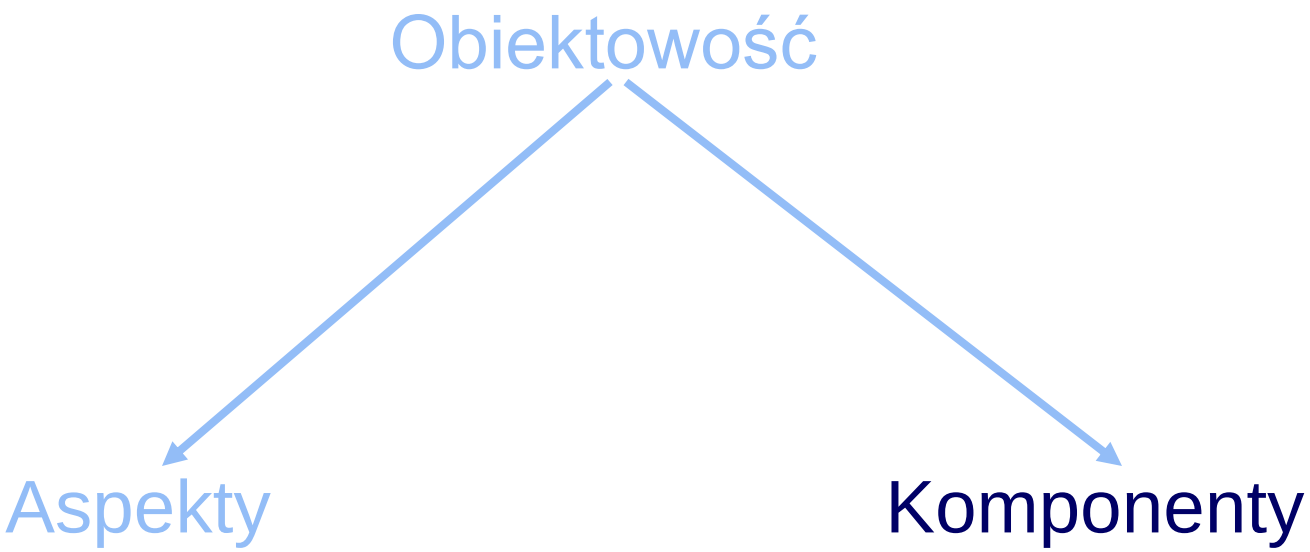
```
aspect A {
   before(): call(* *(..)) {
      System.out.println("przed");
   }
   after() returning: call(* *(..)) {
      System.out.println("po");
   }
}
```

Przykład: wersja 3

Ta wersja nie posiada tej wady...

```
aspect A {
   before(): call(* *(..)) && !within(A) {
      System.out.println("przed");
   }
   after() returning: call(* *(..)) && !within(A) {
      System.out.println("po");
   }
}
```

Plan wykładu



Obiekty a komponenty

Czego jeszcze brak obiektom?

Programowanie obiektowe	***Programowanie komponentowe***
• łączenie obiektów w kodzie programu • późne wiązanie wywołań • opcjonalna hermetyzacja • użycie polimorfizmu • użycie dziedziczenia klas	• łączenie przez konfigurację • późne wiązanie wywołań i ładowanie kodu na żądanie • obowiązkowa hermetyzacja • dziedziczenie interfejsów • powtórne użycie na poziomie binariów

Komponent

Komponent jest

- podstawową jednostką oprogramowania
- z kontraktowo (deklaratywnie) opisanymi interfejsami, i
- podanymi wprost zależnościami.

Komponent może zostać skonfigurowany i wdrożony niezależnie od programisty, który go stworzył.

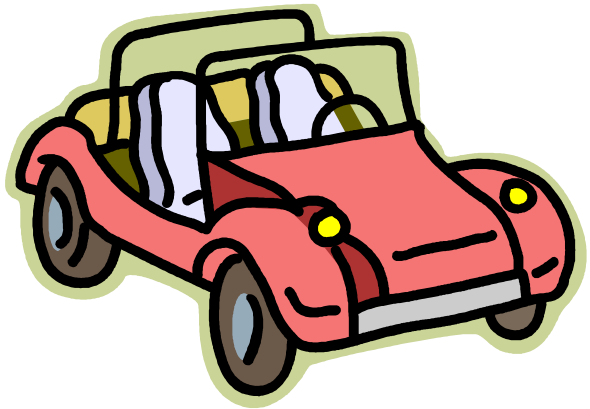
Clemens Szyperski
"Oprogramowanie komponentowe. Obiekty to za mało" (1998)

Własności komponentu

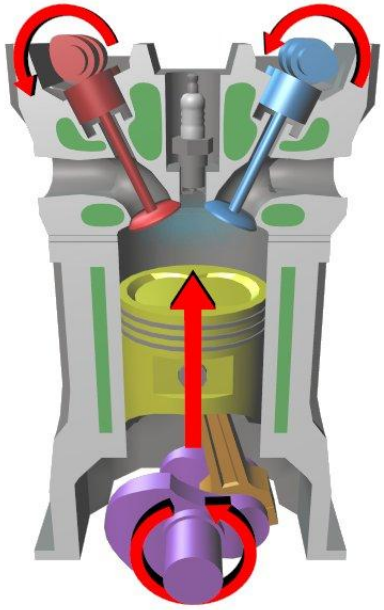
Podstawowe własności komponentu

- Komponenty nigdy nie sprawują kontroli nad sobą
  - zasada hollywoodzka: *proszę do nas nie dzwonić, to my oddzwonimy...*
- Kontener jest specjalnym elementem zarządzającym wszystkimi komponentami
  - steruje procesem tworzenia komponentami
  - rozwiązuje zależności pomiędzy komponentami
  - zarządza cyklem życia komponentów

Przykład

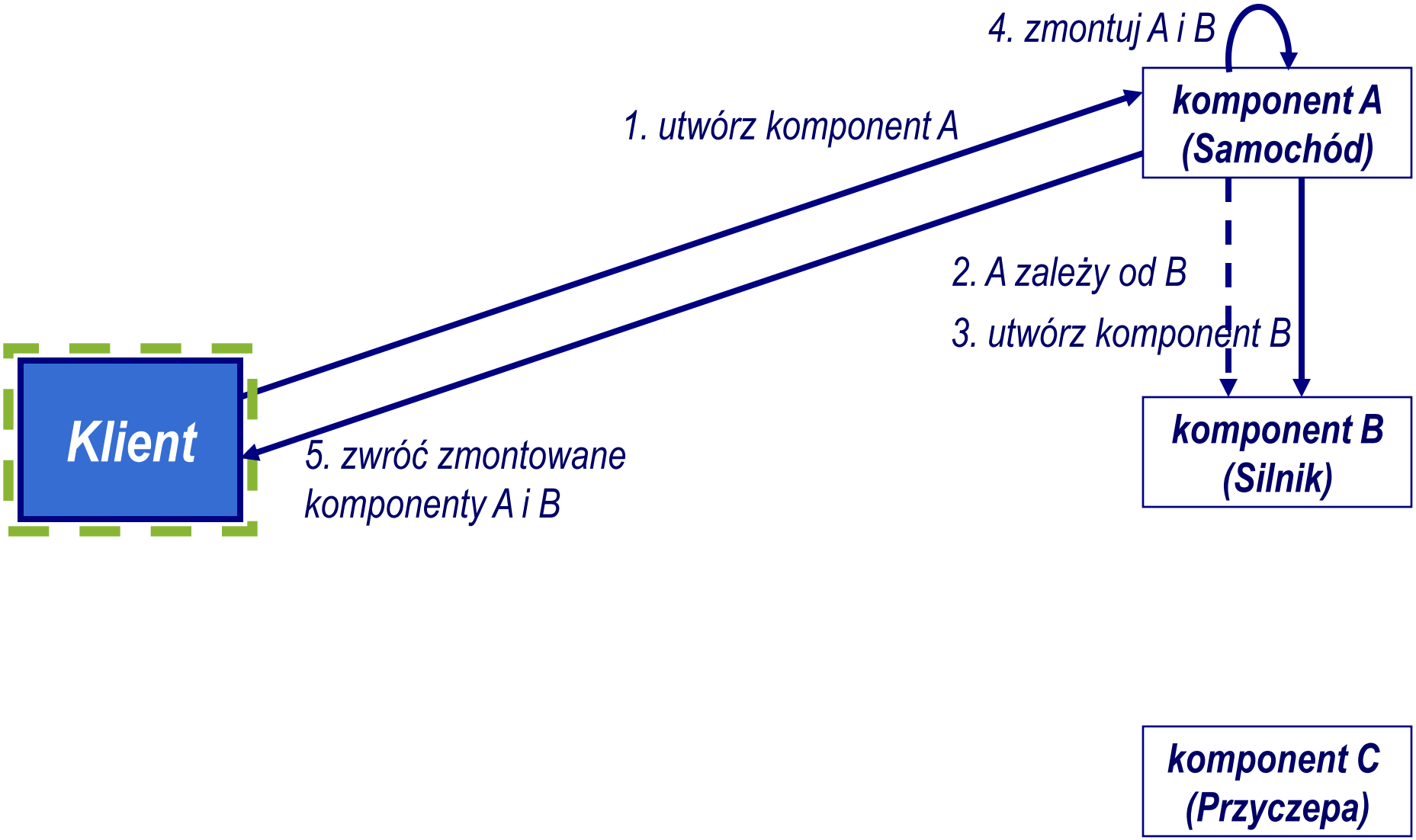


zawiera

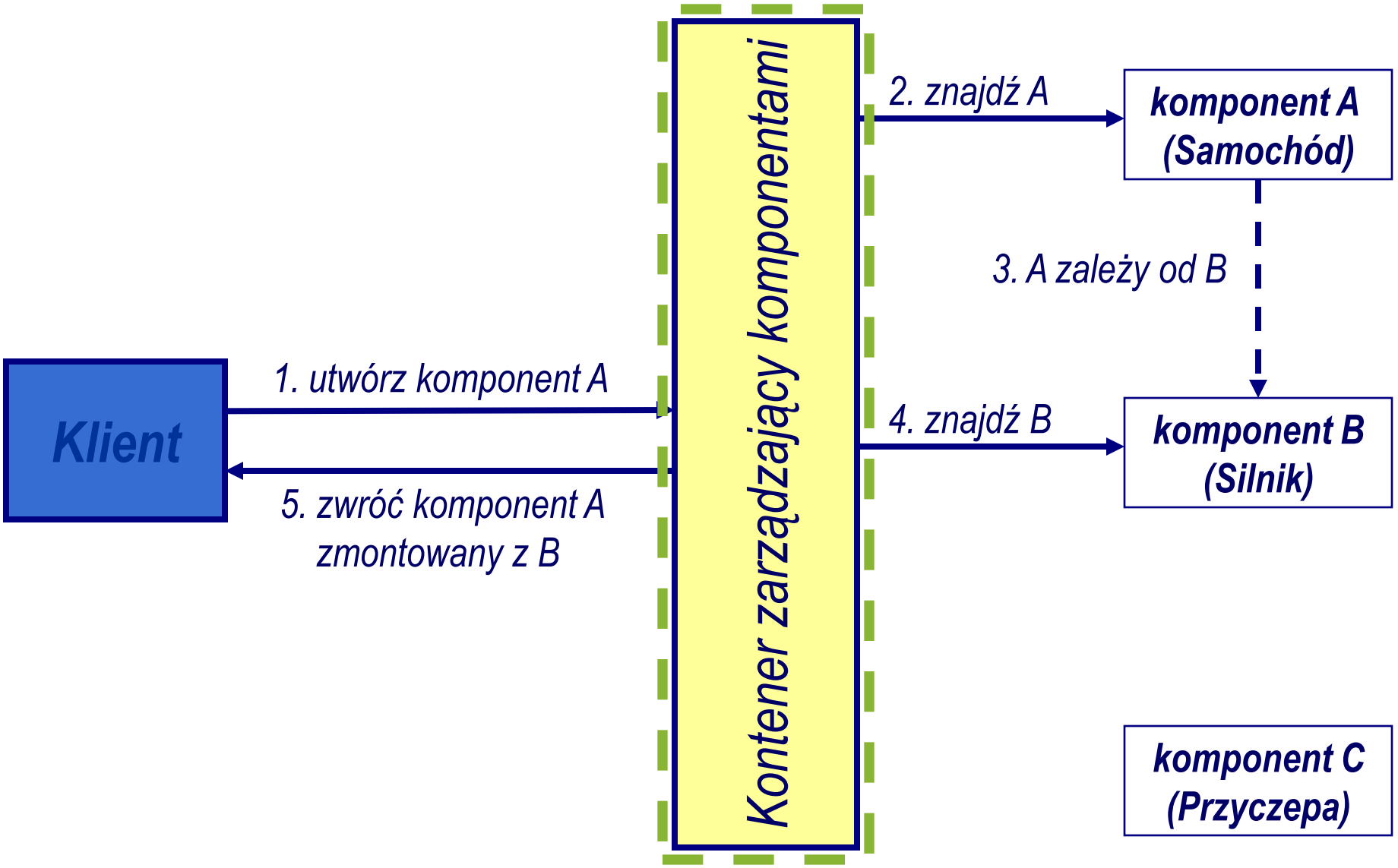


Sprawny Samochód musi zawierać Silnik.

Tradycyjne rozwiązywanie zależności



Odwrócone rozwiązywanie zależności



Wybrane sposoby rozwiązywania zależności

Interface injection	Komponenty implementują dedykowany interfejs, poprzez który otrzymują obiekt służący do wyszukiwania zależności
Setter injection	Zależności są przekazywane przez właściwości obiektu (metody *setXXX()* zgodne z konwencją JavaBeans)

Wyszukiwanie zależności poprzez interfejs

```
public interface Serviceable {
    public void service(ServiceManager manager);
}

public class Samochod implements Serviceable {
    private ServiceManager manager = null;
    private Silnik silnik = null;

    public void service(ServiceManager manager) {
        this.manager = manager;
    }

    public void zbuduj() {
        this.silnik = (Silnik) manager.lookup("Silnik");
    }
}
```

Wstrzykiwanie zależności przez właściwości

```
public class Samochod {
    private Silnik silnik = null;

    public Samochod() {
    }

    public void setSilnik(Silnik silnik) {
        this.silnik = silnik;
    }
}
```

Kontener

Samochód -> Silnik -> Silnik1_6

Odwrócenie sterowania

Odwrócenie sterowania

- pasywne API komponentu
- komunikacja przez interfejsy
- automatyczne spełnianie zależności
- przeniesienie odpowiedzialności za rozwiązanie zależności na kontener

Podsumowanie

- Obiektowość jest podejściem do problemu modularyzacji programu
- Istotą obiektowości jest pojęcie odpowiedzialności
- Aspekty stanowią rozszerzenie obiektowości o nowy rodzaj jednostki
- Komponenty, w odróżnieniu od obiektów, są zewnętrznie konfigurowalne