Zaawansowane programowanie w C++ (PCP)

Wykład 4 - wzorce projektowe.

dr inż. Robert Nowak

Powtórzenie

- klasy autonomiczne
- tworzenie nowych typów: dziedziczenie i agregacja
- dziedziczenie: przedefiniowywanie metod, problem przycinania, zasłanianie
- problem typu dla klas w hierarchii
- funkcje wirtualne
- wirtualny destruktor

Powtórzenie (2)

Zalety podejścia obiektowego:

- klasy autonomiczne
  - enkapsulacja (ochrona składowych)
  - konstrukcja i destrukcja
- polimorfizm
  - umożliwia definiowanie ogólnych cech pokrewnych typów;
  - można pisać ogólne funkcje działające dla wszystkich pochodnych pewnej klasy bazowej;
  - elastyczny system typów: można dodawać nowe typy bez modyfikacji już istniejącego kodu;

Klasy abstrakcyjne

- klasy, które mają sens jedynie jako klasa bazowa
- nie ma sensownej implementacji metod wirtualnych
- reprezentuje abstrakcyjne pojęcie

```
class Figura {
    virtual void rysuj() = 0;//Funkcja czysto wirtualna
};
```

- kompilator nie dopuszcza do tworzenia obiektów takich klas
- wykrywany błąd obcinania już na etapie kompilacji!
- nie musi dostarczać konstruktorów

klasa interfejsu - klasa abstrakcyjna, która nie przechowuje stanu.

Właściwości klasy bazowej

Klasa wartość (klasa autonomiczna):

- brak metod wirtualnych
- konstruktor, konstruktor kopiujący, operator przypisania, destruktor
- najczęściej obiekt automatyczny lub składowa klasy
- przekazywany przez wartość lub stałą referencję

Klasa bazowa dla hierarchii klas:

- używa metod wirtualnych, powinna mieć wirtualny destruktor
- najlepiej gdy abstrakcyjna albo prywatny konstruktor kopiujący i operator przypisania (zapobiega wycinaniu)
- najczęściej obiekt na stercie
- przekazywana przez wskaźnik lub referencję

Metody wirtualne a konstruktory

Mechanizm funkcji wirtualnych nie działa w konstruktorach i destruktorach.

```
class Base {
public:
    Base(){ f(); }
    virtual void f(){ i=1; }
protected:
    int i;
};
class Derived {
public:
    Derived() : Base() { }
    virtual void f(){ i=2; }
};

Derived d; //Jaka jest wartość składowej i?
Base& b = d;
b.f(); //Jaka jest wartość składowej i?
```

Dziedziczenie prywatne i chronione


```
class D: public B { ... };//Dziedziczenie publiczne
class D: protected B { ... };//Dziedziczenie chronione
class D: private B { ... };//Dziedziczenie prywatne
```

Ochrona dla różnych rodzajów dziedziczenia:

rodzaj	Dostęp do		
	składowych publicznych B	składowych protected B	konwersji D* $\longrightarrow$ B*
publiczne	wszystkie funkcje	metody D oraz klas pochodnych po D	wszystkie funkcje
chronione	metody D oraz klas pochodnych po D		
prywatne	metody D		

Wzorce projektowe

- standardowe rozwiązania często pojawiających się problemów projektowych
- sprawdzone w praktyce
- najczęściej dotyczą programowania obiektowego
- znajomość wzorców projektowych pozwala lepiej zrozumieć obiektowe podejście do programowania

Wzorzec singletonu

Obiekt globalny (problemy):

- inicjalizacja,
- dostęp,
- jedna kopia w programie.

```
class Singleton {
public:
     static Singleton& getInstance() {
          static Singleton instance; //lokalna zmienna statyczn
          return instance;
     }
     /* pozostałe metody */
private:
     //Prywatne konstruktory
     Singleton() { /* inicjacja składowych */ }
     Singleton(const Singleton&);
     /* pozostałe składowe */
};
```

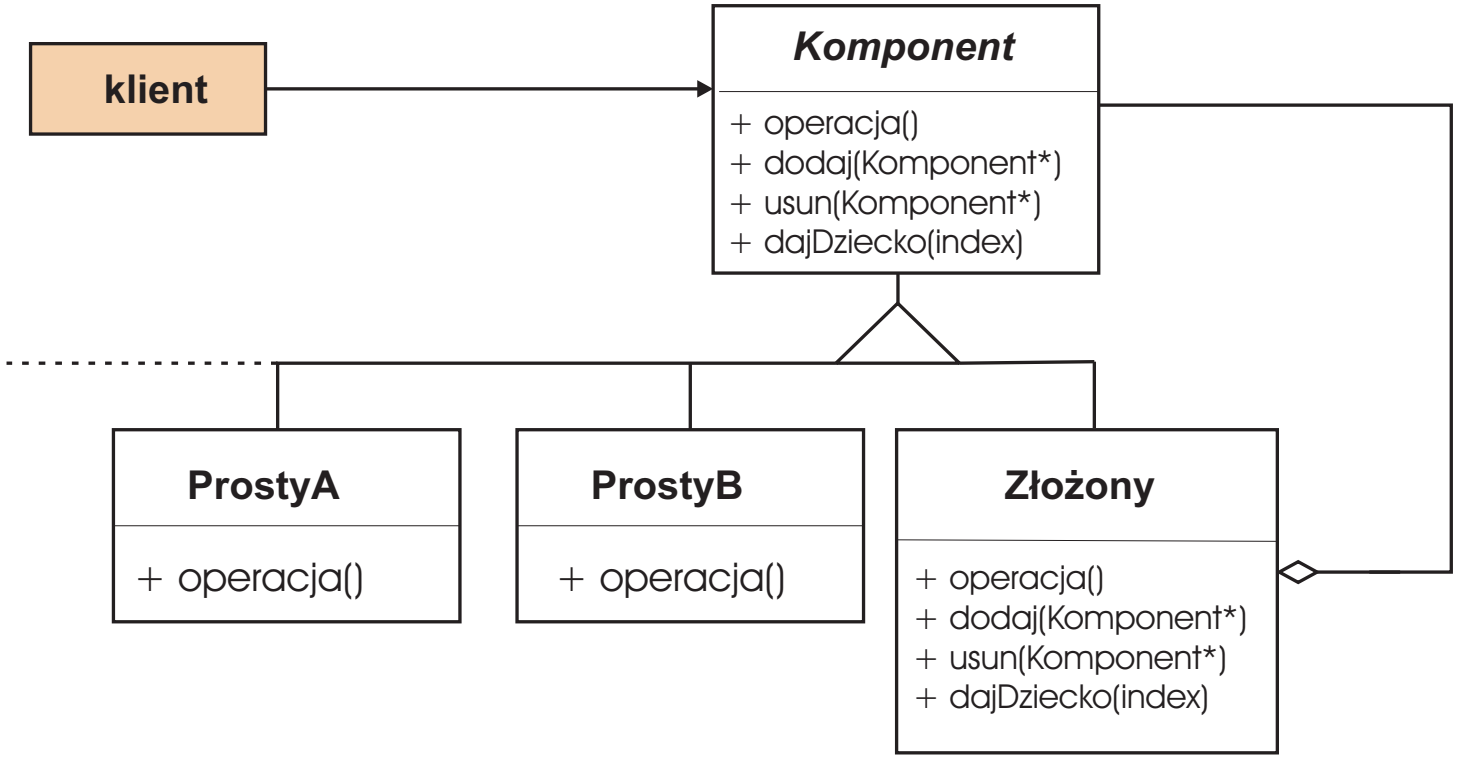
Wzorzec prototypu (metoda clone)

- Kopiowanie bez znajomości typu oraz bez wycinania
- Wzorzec prototypu:
  - odpowiedzialność przeniesiona na obiekty pochodne
  - wykorzystanie mechanizmu funkcji wirtualnej

```
class Base {
public:
    virtual Base* clone() = 0;//tworzy kopie danego obiektu
private:
    Base(const Base&);//Zabroniony konstruktor kopiujący
};
class Derived : public Base {
public:
    virtual Base* clone(){
        //Tutaj już wie, jaki typ kopiować!
        return new Derived(*this);
    }
};
```

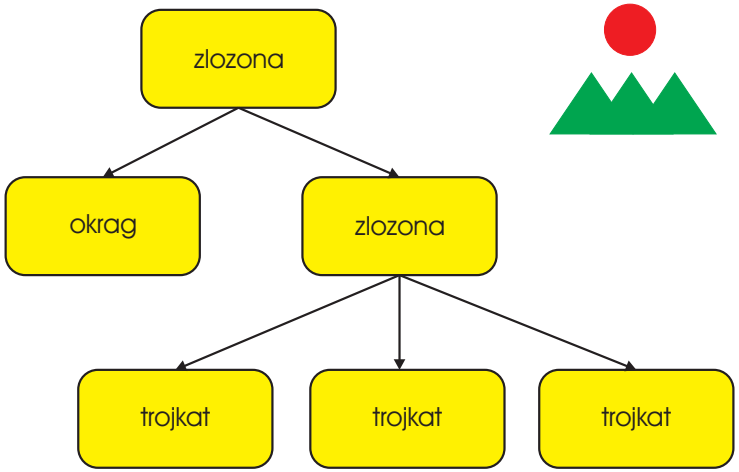
Wzorzec kompozycji

- reprezentacja drzewiastych struktur obiektów
- traktowanie w ten sam sposób obiektów prostych i złożonych
- łatwo dodawać nowe klasy do hierarchii



Kompozycja - przykład

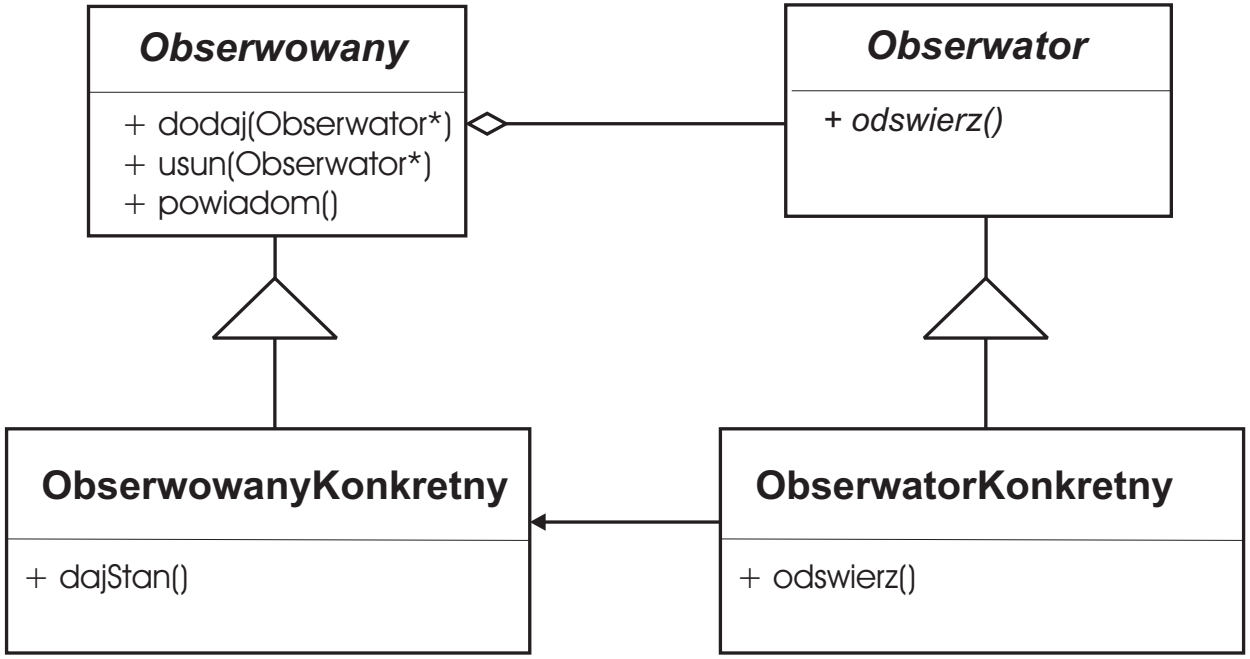

```
class Figura {
public:
    virtual void draw() = 0;
    virtual ~Figura(){}
};
class Okrag : public Figura {
public:
    //rysuje okrąg
    virtual void draw();
};
```




```
class Zlozona : public Figura {
public:
    //dla każdego dziecka wywołuje metodę 'draw'
    virtual void draw();
private:
    std::vector<Figura*> dzieci;
};
```

Wzorzec obserwatora

- zmiana stanu jednego obiektu wymaga zmiany innych
- obiekt powiadamia o zmianie stanu nie zakładając niczego o obiektach, które są powiadamiane

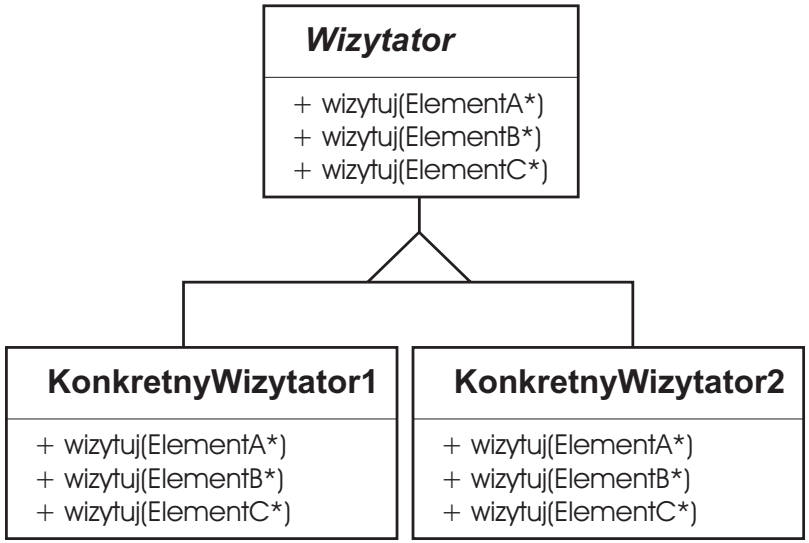


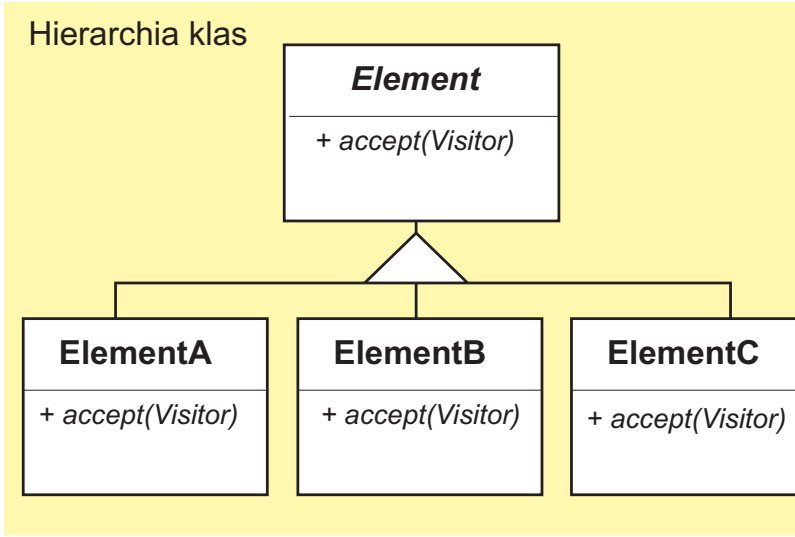
Wzorzec obserwatora (2)


```
//Abstrakcyjny obserwator
class Observer {
public:
    virtual void update() = 0;
    virtual ~Observer(){}
};
//Abstrakcyjny cel obserwacji
class Subject {
    void add(Observer* o){ observers_.push_back(o); }
    void notify(){ /* wywolaj o->update() dla wszystkich */ }
    virtual ~Subject() = 0;
private:
    std::vector<Observer*> observers_;
};
//Czysto wirtualny destruktor także musi być zaimplementowany!
Subject::~Subject(){}
```

Wzorzec wizytatora

- ustalona hierarchia klas
- wiele metod, metody mogą się zmieniać





Wizytator (2)


```
class Visitor;//Deklaracja
class Element { //Klasa bazowa
    virtual void accept(Visitor& v) = 0;
};
/* Deklaracje konkretnych elementów */
class Visitor { //Abstrakcyjny wizytator
public:
    virtual void visit(ElementA&) = 0;
    virtual void visit(ElementB&) = 0;
    virtual void visit(ElementC&) = 0;
};
class ElementA : public Element {
public:
    virtual void accept(Visitator& v) {
        v.visit(*this);//Woła metodę visit(ElementA&)
    }
    //Dalsza część implementacji klasy ElementA
};
```

Wzorce projektowe - podsumowanie

- wzorce kreacyjne
  - fabryka abstrakcyjna (abstract factory)
  - **prototyp (prototype)**
  - **singleton (singleton)**
- wzorce strukturalne
  - adapter (wrapper)
  - fasada (fasade)
  - most (bridge)
  - proxy (proxy)
  - **kompozyt (composite)**
- wzorce czynnościowe
  - iterator (iterator)
  - polecenie (command)
  - **odwiedzający (visitor)**
  - **obserwator (observer)**
  - interpreter (interpreter)
  - strategia (strategy)
  - stan (state)

Wzorce projektowe - podsumowanie (2)

E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Wzorce projektowe*, WNT, 2005.