

Sztuka Wytwarzania Oprogramowania

Wykład 3 - obiektowe wzorce projektowe, cz 1

Robert Nowak

24L

Obiektowe wzorce projektowe

- ▶ standardowe rozwiązania typowych problemów
- ▶ sprawdzone w praktyce
- ▶ znajomość obiektowych wzorców projektowych pozwala lepiej programować obiektowo

Termin „wzorce projektowe” upowszechnił się po wydaniu „Design Patterns: ...”, Gamma, Helm, Johnson, Vlissides (1994), zawierająca 23 wzorce.

Plan wykładu:

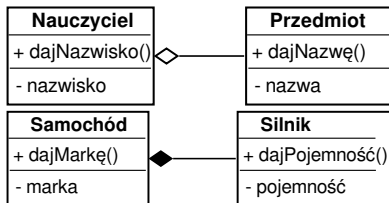
- ▶ prototyp
- ▶ kompozyt
- ▶ adapter
- ▶ proxy (leniwe tworzenie, leniwe kopiowanie)
- ▶ obserwator
- ▶ komenda
- ▶ dekorator

Powtórzenie: agregacja, dziedziczenie

```

class Lecture;
class Teacher {
    //agregacja
    std::vector<Lecture*> lec_;
};
class Engine;
class Car {
    //kompozycja
    Engine engine_;
};

```



```

//dziedziczenie
//klasa bazowa
class Person;
//klasa pochodna
class Student : public Person {
};

```

Wycinanie

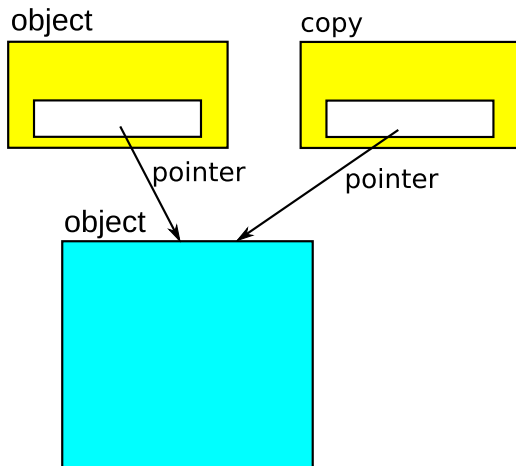
Działanie konstruktora kopiującego lub operatora przypisania:

```
class Pracownik { ... };  
class Kierownik : public Pracownik { ... };
```

```
Kierownik k(...);  
Pracownik p = k;
```

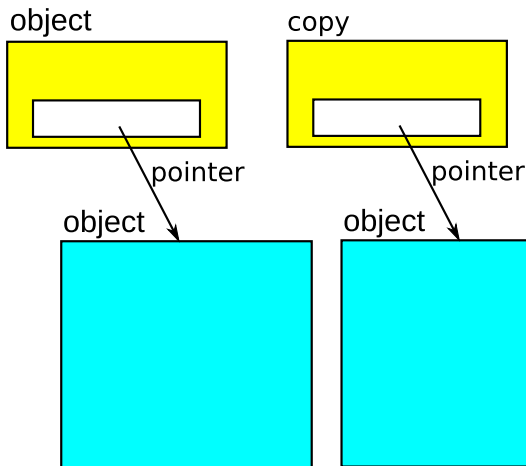
- ▶ kopiuje tylko część klasy,
- ▶ źródło niespodzianek i błędów,
- ▶ rozwiązanie: przekazywanie wskaźników lub referencji do obiektów.

płytki kopia (shell copy)



```
Obj* obj = new Obj();  
Obj* copy = obj;
```

głęboka kopia (deep copy)



```
Obj* obj = new Obj();  
Obj* copy = new Obj(*obj);
```

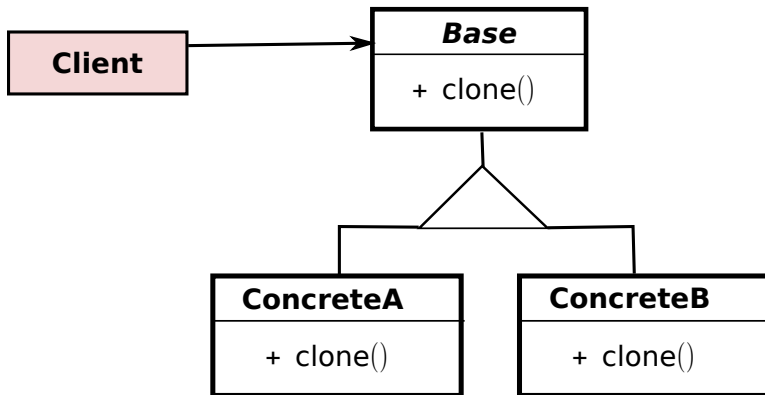
Prototyp

„głęboka” i „płytką” kopia

```
class B {};  
class D1 : public B {};  
class D2 : public B {};  
  
vector<B*> v; v.push_back(new D1); v.push_back(new D2);  
  
vector<B*> u = v; //Płytką kopia  
  
vector<B*> uu;  
for(const B* b : v)  
    uu.push_back(new B(*b)); //WYCINANIE!
```


Wzorzec prototypu

- ▶ odpowiedzialność przeniesiona na obiekty pochodne
- ▶ wykorzystanie mechanizmu funkcji wirtualnej



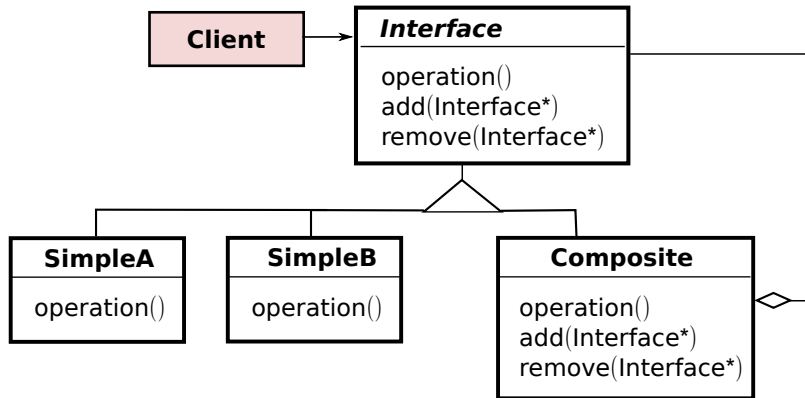
prototyp (clone, wirtualny konstruktor) - przykład

```
class B {
public:
    virtual B* clone() const = 0; //tworzy kopie danego obiektu
    B(const B&) = delete; //Zabroniony konstruktor kopiujący
};
class D1 : public B {
public:
    D1(const D1& r); //Konstruktor kopiujący
    virtual B* clone() const {
        return new D1(*this); //Już wie, jaki typ kopiować!
    }
};
vector<B*> v1, v2; //v2 będzie głęboką kopią v1
for(const B* b : v1 ) v2.push_back( b->clone() );
```

Kompozyt

Wzorzec kompozytu

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

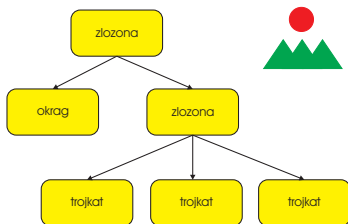


Kompozyt - przykład

```

class Fig {
public:
    virtual void draw() = 0;
    virtual ~Fig(){}
};
class Circle : public Fig {
public:
    virtual void draw(); //rysuje
    okrąg
};
class Composite : public Fig {
public:
    virtual void draw() { //dla każdego dziecka wywołuje 'draw'
        for(Fig* f : children_) f->draw();
    }
private:
    std::vector<Fig*> children_;
};

```



Adapter

Adapter (wrapper)

Wzorzec adaptera - dostosowanie interfejsu klasy do interfejsu, którego oczekuje użytkownik

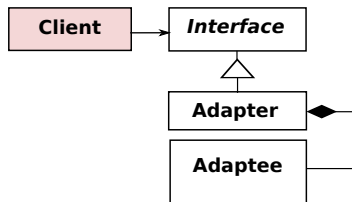
- ▶ obiekty pewnej klasy przechowują potrzebne dane
- ▶ obiekty zachowują się w odpowiedni sposób
- ▶ zmiana interfejsu jest kłopotliwa (niemożliwa)

gdy mamy typ, ale nie jest on umieszczony w odpowiedniej hierarchii klas

Rozwiązanie - nowa klasa, która dostosowuje interfejs. Wersje:

- ▶ agregacja (adaptery obiektów)
- ▶ dziedziczenie prywatne (adaptery klas)

Adaptory obiektów



Nowa klasa (adapter)

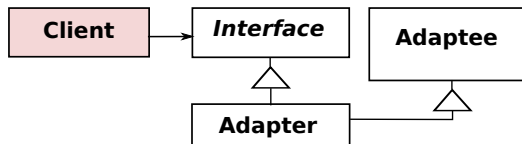
- ▶ agreguje istniejącą klasę
- ▶ posiada wymagany interfejs
- ▶ wywołuje odpowiednie metody

//Przykład adaptera obiektów

```

class Adapter : public Interfejs {
public:
    virtual int metoda1() { return obj_.metoda2(); }
private:
    Adaptowany obj_;
};
  
```


Adaptery klas



Nowa klasa (adapter)

- ▶ posiada wymagany interfejs (dziedziczy go publicznie)
- ▶ posiada obiekt istniejącej klasy (dziedziczenie prywatnie)

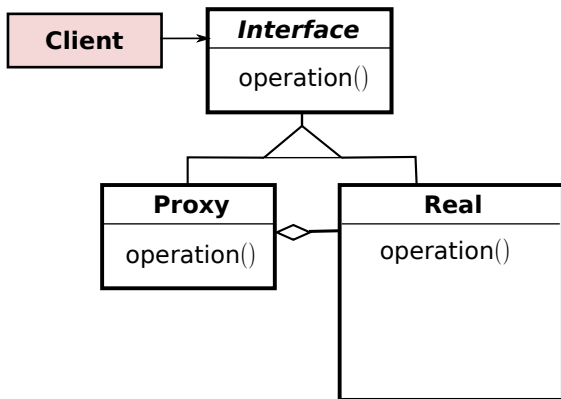
//Przykład adaptera klas

```

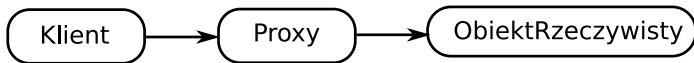
class Adapter : public Interfejs, private Adoptowany {
public:
    virtual int metoda1() { return metoda2(); }
};
  
```

Pośrednik (proxy)

Wzorzec proxy



Proxy to uchwyt (wskaźnik) do obiektu.



Wzorzec proxy - rodzaje

- ▶ tworzy obiekt przy pierwszym użyciu (Virtual Proxy)
- ▶ odkłada utworzenie głębokiej kopii obiektu (Copy-On-Write Proxy)
- ▶ zarządza czasem życia obiektu, niszczy obiekt na stercie (Smart Pointer)
- ▶ inne
 - ▶ obiekt w innej przestrzeni adresowej (Remote Proxy)
 - ▶ kontroluje prawa dostępu do obiektu (Protection Proxy)
 - ▶ synchronizuje dostęp do obiektu (Synchronization Proxy)

Virtual Proxy, tworzenie przy pierwszym użyciu

```
class Image { //Interfejs
public: virtual void draw() = 0;
};
Image* createImage() { return new ProxyImage(); }
//klasa 'ciężka', realizuje funkcjonalność
class RealImage : public Image { };
//klasa 'lekka', uchwyt
class ProxyImage : public Image {
public:
    ProxyImage() : realObj_(nullptr) {}
    virtual void draw(){ getImage()->draw(); } //pośredniczy
private:
    Image* real_; //uchwyt
    Image* getImage() {
        if(!real_) real_ = new RealImage();
        return real_;
    }
};
```

Copy-On-Write Proxy, leniwe kopiowanie



```

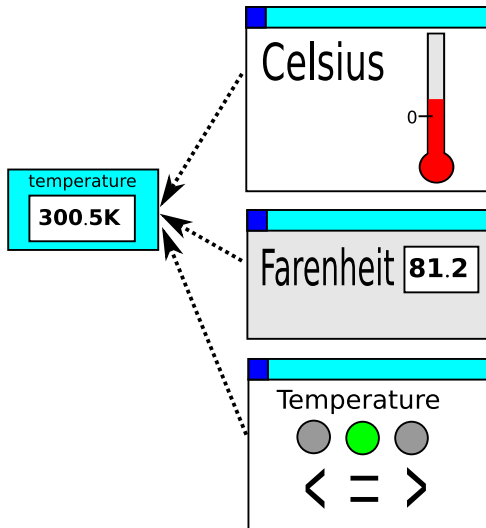
class CFoo { //leniwe kopiowanie dla Foo, Foo zawiera licznik
public:
    CFoo(const CFoo& c) { join(c.foo_); }
    ~CFoo() { unjoin(); }
    int get() const { return foo_->val_; }
    void set(int v) {
        if(foo_->count_ == 1) foo_->val_ = v;
        else { --foo_->count_; foo_ = new Foo(v); } //kopia głęboka
    }
private:
    void join(Foo* f) { foo_ = f; ++foo_->count_; }
    void unjoin() { --foo_->count_; if(foo_->count_ == 0) delete foo_; }
    Foo* foo_;
};

```

Obserwator

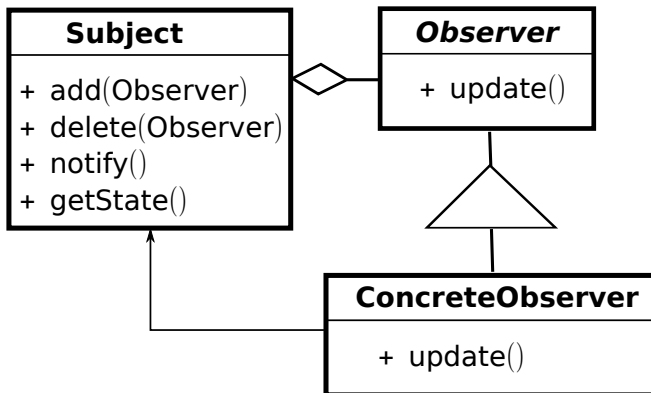
Wzorzec obserwatora

- ▶ zmiana stanu jednego obiektu wymaga zmiany innych



Wzorzec obserwatora

obiekt powiadamia o zmianie stanu nie zakładając niczego o obiektach, które są powiadamiane



Wzorzec obserwatora (2)

```
class Observer { //Abstrakcyjny obserwator
public:
    virtual void update() = 0;
    virtual ~Observer(){}
};

class Subject { //Abstrakcyjny cel obserwacji
    void add(Observer* o){ obs_.push_back(o); }
    void notify(){ //wywołaj o->update() dla wszystkich
        for(Observer* o : obs_)
            o->update();
    }
    virtual ~Subject() = 0; //czysto wirtualny destruktor
private:
    std::vector<Observer*> obs_;
};

Subject::~~Subject(){} //destruktor musi być zaimplementowany!
```

Obserwator - zastosowanie

MVC
(Model View Controller)

- ▶ Model = Subject
- ▶ View = Observer

QT (Trolltech) boost::signals

- ▶ Signal = Subject
- ▶ Slot = Observer

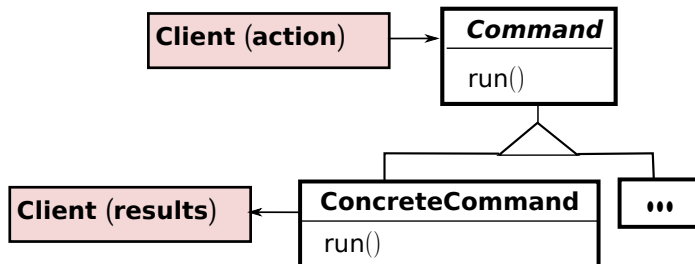
powiązanie pomiędzy zdarzeniami (signals, events, publisher)
a akcjami (slots, event targets, subscribers).

Dodatkowe udogodnienia:

- ▶ obiekty nie muszą martwić się o czas życia
- ▶ obiekty mogą być w różnych wątkach

Komenda

Wzorzec komendy - obiekt przechowuje akcję oraz parametry



```

//Przykład - wołanie funkcji
window.resize(0, 0, 200, 300); //Zmienia rozmiar okna
//Przykład - to samo za pomocą komendy
Komenda* cmd =
new KomendaResize( window, &Window::resize, 0, 0, 200, 300);
/* ... */
cmd->run(); //opóźnione wykonanie
  
```

Komenda: generalizacja wskaźnika do funkcji

```
//Konwencja w C++, komenda to funktor, klasa dostarczająca operator()  
struct Funktor {  
    void operator()(int i);  
};  
Funktor f; //tworzy obiekt  
f(5); //woła obiekt z parametrami
```

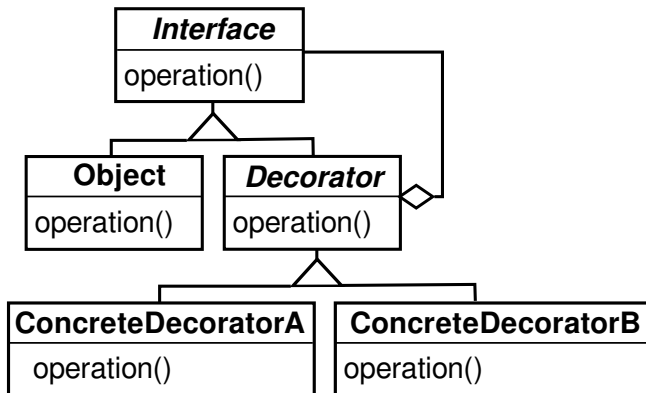
Zalety stosowania komend:

- ▶ komenda - semantyka wartości, więc można je:
 - ▶ kopiować (prototyp, `clone()`),
 - ▶ dostarczać jako argument,
 - ▶ zwracać jako wynik,
 - ▶ przechowywać w kolekcjach;
- ▶ możliwość dostarczenia wycofywania:
 - ▶ funkcja odwrotna,
 - ▶ zapamiętywanie stanu i odtwarzanie;
- ▶ można je wykorzystać do przetwarzania równoległego.

Dekorator

Dekorator

- ▶ zmiana funkcjonalności dla obiektów (w czasie działania)
- ▶ alternatywa dla dziedziczenia (inna funkcjonalność w czasie kompilacji)



Dekorator - przykład

```
class Coffee {
public:
    virtual std::string getIngr() = 0; //the ingredients of the coffee
};
class SimpleCoffee : public Coffee { //concrete class
public:
    virtual std::string getIngr() { return "Coffee"; }
};
class Decorator : public Coffee {
public:
    Decorator(Coffee* d) : dec_(d) {}
    virtual std::string getIngr() { return dec_.getIngr(); }
private:
    Coffee* dec_;
};
```

Dekorator - przykład (2)

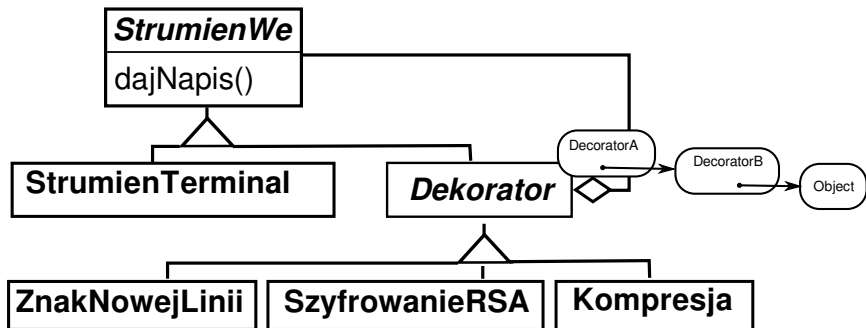
```
class Milk : public Decorator {
public:
    Milk(Cofee* decorated) : Decorator(decorated) {}
    virtual std::string getIngr() {
        return Decorator::getIngr() + " Milk"; }
};

class Sugar : public Decorator {
public:
    Sugar(Cofee* decorated) : Decorator(decorated) {}
    virtual std::string getIngr() {
        return Decorator::getIngr() + " Sugar"; }
};
```

„Podobne” wzorce

- ▶ kompozyt, ale dekorator posiada tylko jeden komponent
- ▶ proxy kontroluje dostęp, dekorator dodaje funkcjonalność
- ▶ adapter to inny interfejs, dekorator to inna funkcjonalność

Dekorator - przykład 2



```

StrumienWe* s1 = new StrumienTerminal(); //prosty strumień
StrumienWe* s2 = new Kompresja(new StrumienTerminal()); //z kompresją
StrumienWe* s3 = new KodowanieRSA(
    new ZnakNowejLinii(
        new StrumienTerminal(), "\r\n" ), /* klucz RSA
*/ );
  
```

Dziękuję

robert.nowak@pw.edu.pl