Sztuka Wytwarzania Oprogramowania

Wykład 6 - obiektowe wzorce projektowe, cz 1

Robert Nowak

25Z

Przedmiot SWO

Przedmiot dotyczący technik i praktyk stosowanych w wytwarzaniu i utrzymaniu oprogramowania. Nacisk położony jest na pracę projektanta kodu i programisty, ukazując kodowanie jako element większej całości.

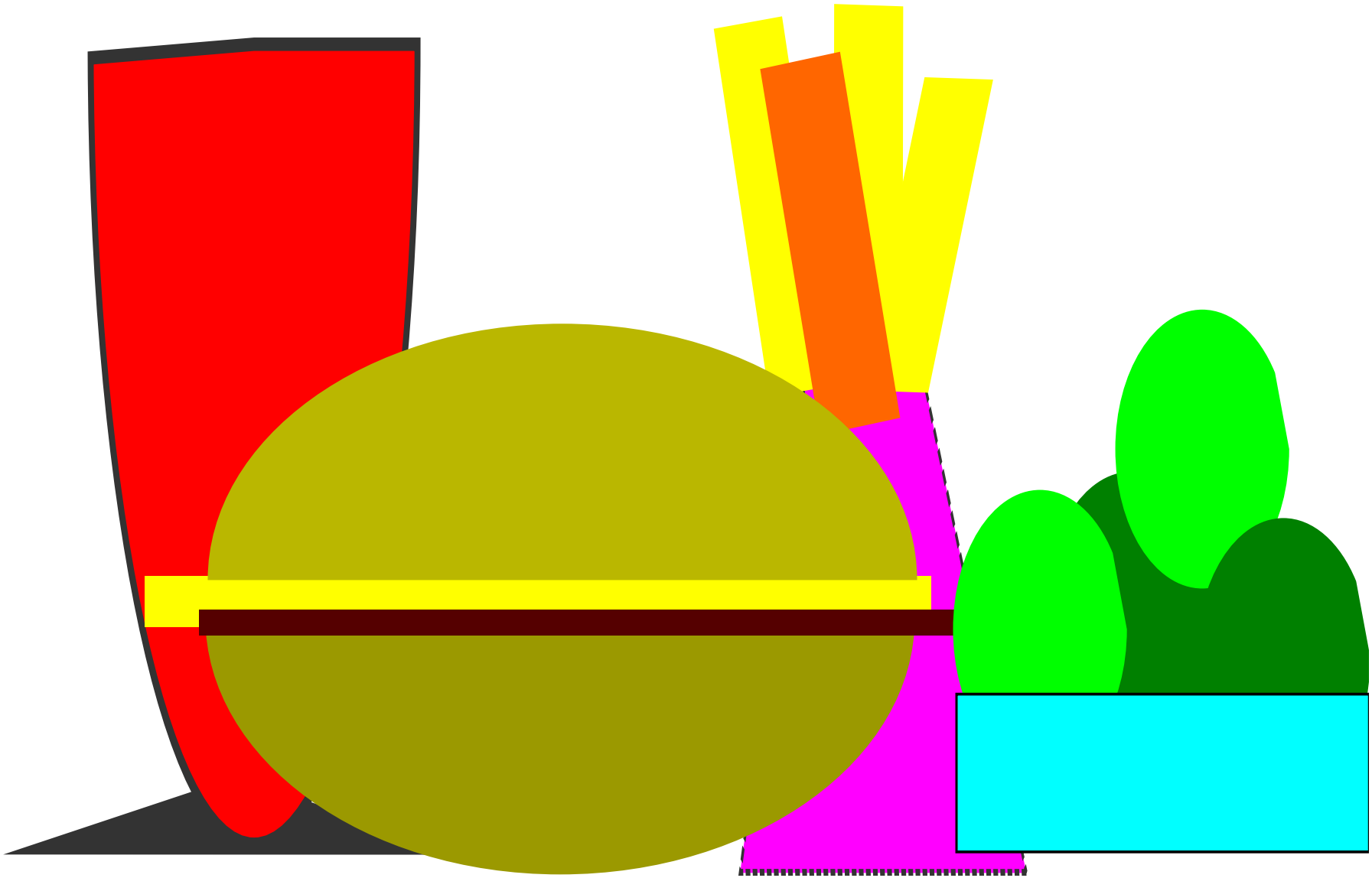
Kluczowa jest znajomość inżynierii oprogramowania, technik programowania, algorytmów, bibliotek i stosowanie narzędzi.

np: wykrywanie błędów w programach (wykorzystywać kompilator, testy, debugger itd.)

np. Zarządzanie kodami źródłowymi (stosowanie repozytorium)

np. Optymalizacja oprogramowania

Jakość w tworzeniu programowania



Tworzenie nowych klas

Budowa klas na podstawie już istniejących (wielokrotne wykorzystanie kodu):

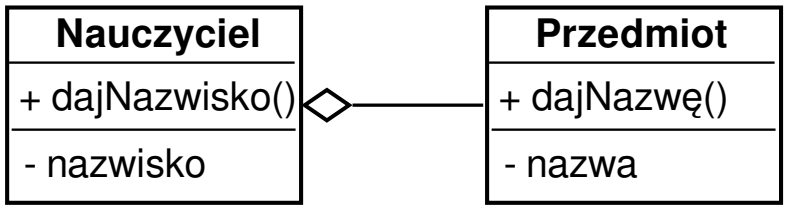
- agregacja,
- dziedziczenie.

Agregacja: prostsza kontrola - należy ją faworyzować

Język UML (Unified Modeling Language) - diagram klas.

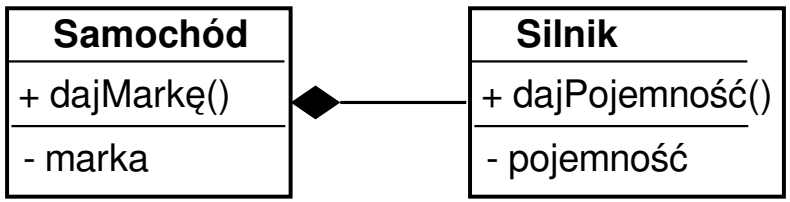
Agregacja - relacja „składa się z” lub „posiada”

Agregacja- gdy budowa z mniejszych kawałków większej całości.




```
class Przedmiot { /* ... */ };
class Nauczyciel {
private:
  std::vector<Przedmiot*> przedm_;
};
```

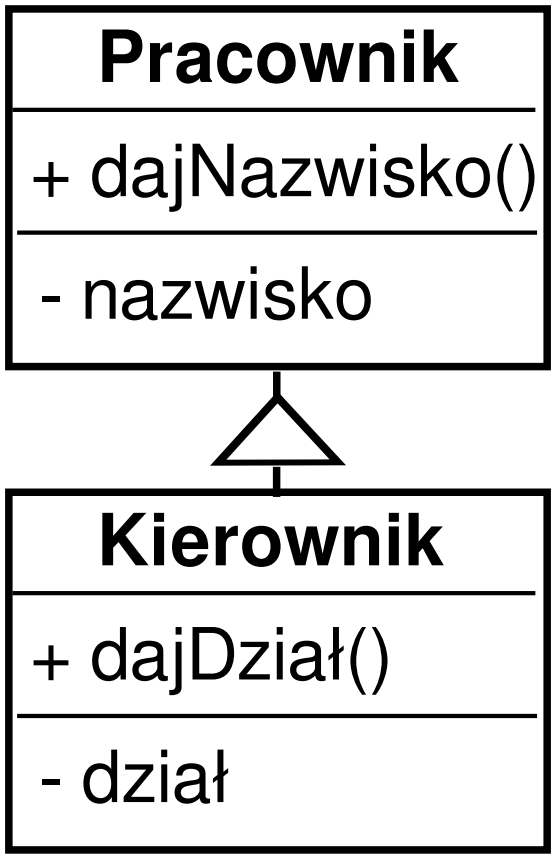
Kompozycja - agregacja, gdy obiekt składowy nie może istnieć bez obiektu głównego



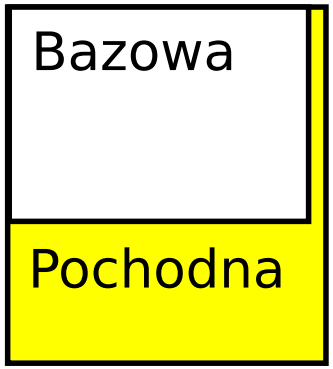

```
class Silnik { /* ... */ };
class Samochod {
private:
  Silnik silnik_;
};
```

Dziedziczenie - relacja „może być traktowany jako”

Dziedziczenie wprowadza powiązanie pomiędzy typami.




```
//klasa bazowa
class Pracownik {
  //...
};
//klasas pochodna
class Kierownik : public Pracownik {
  //...
};
```



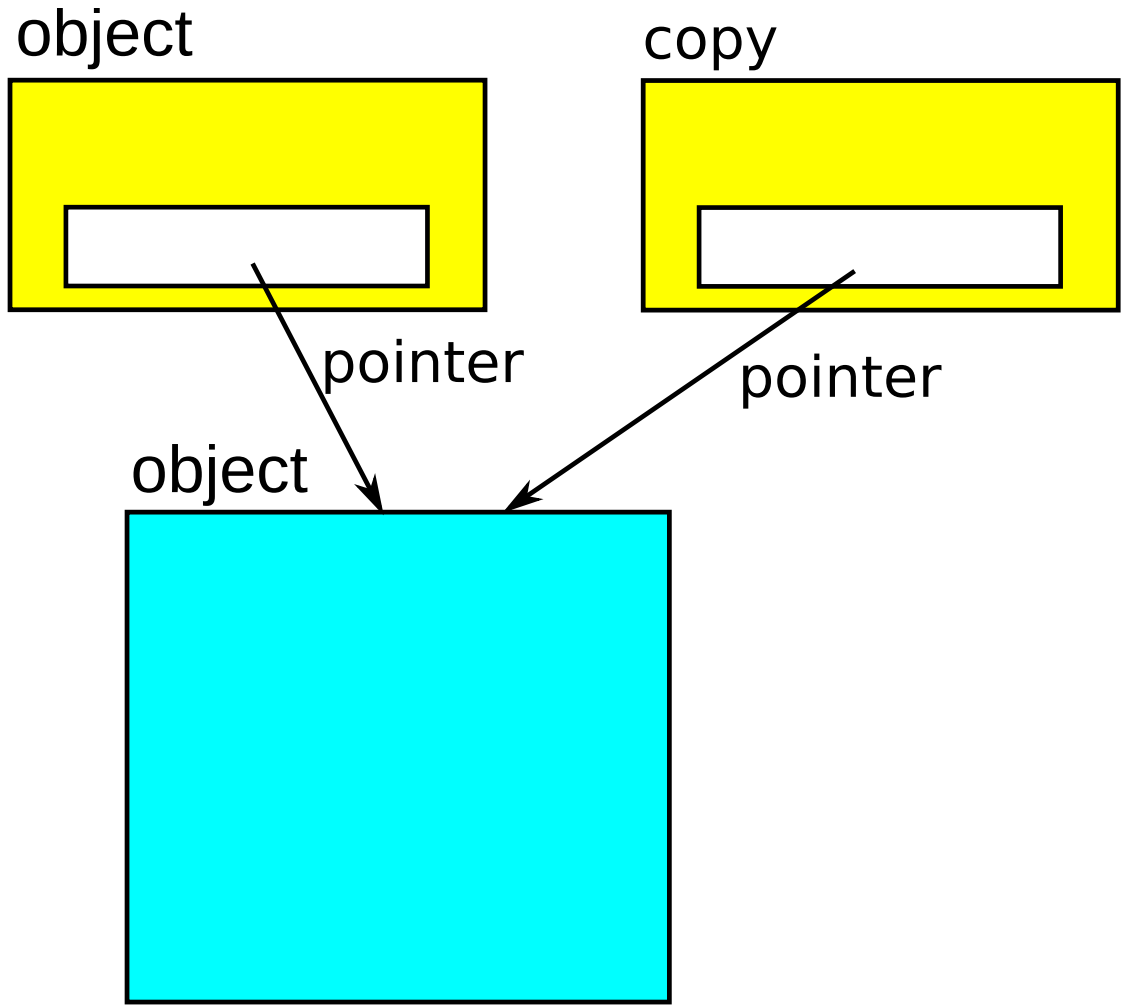
Obiektowe wzorce projektowe

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

Termin 'wzorce projektowe' upowszechnił się po wydaniu książki „Design Patterns: Elements of Reusable Object-Oriented Software" autorstwa Erich Gamma, Richald Helm, Ralph Johnson, John Vlissides" (1994), zawierająca 23 wzorce.

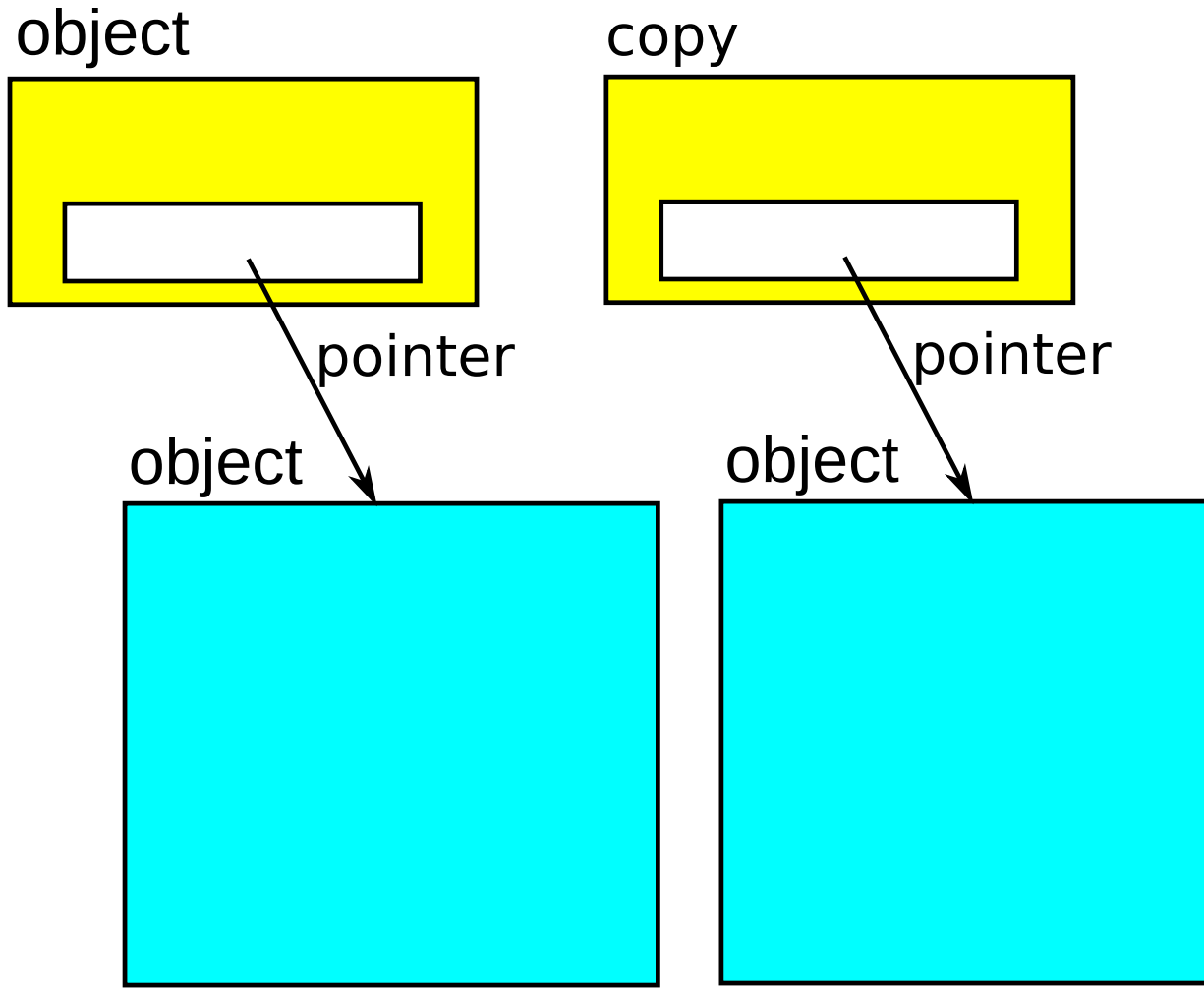
płytka kopia (shell copy)

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



głęboka kopia (deep copy)

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



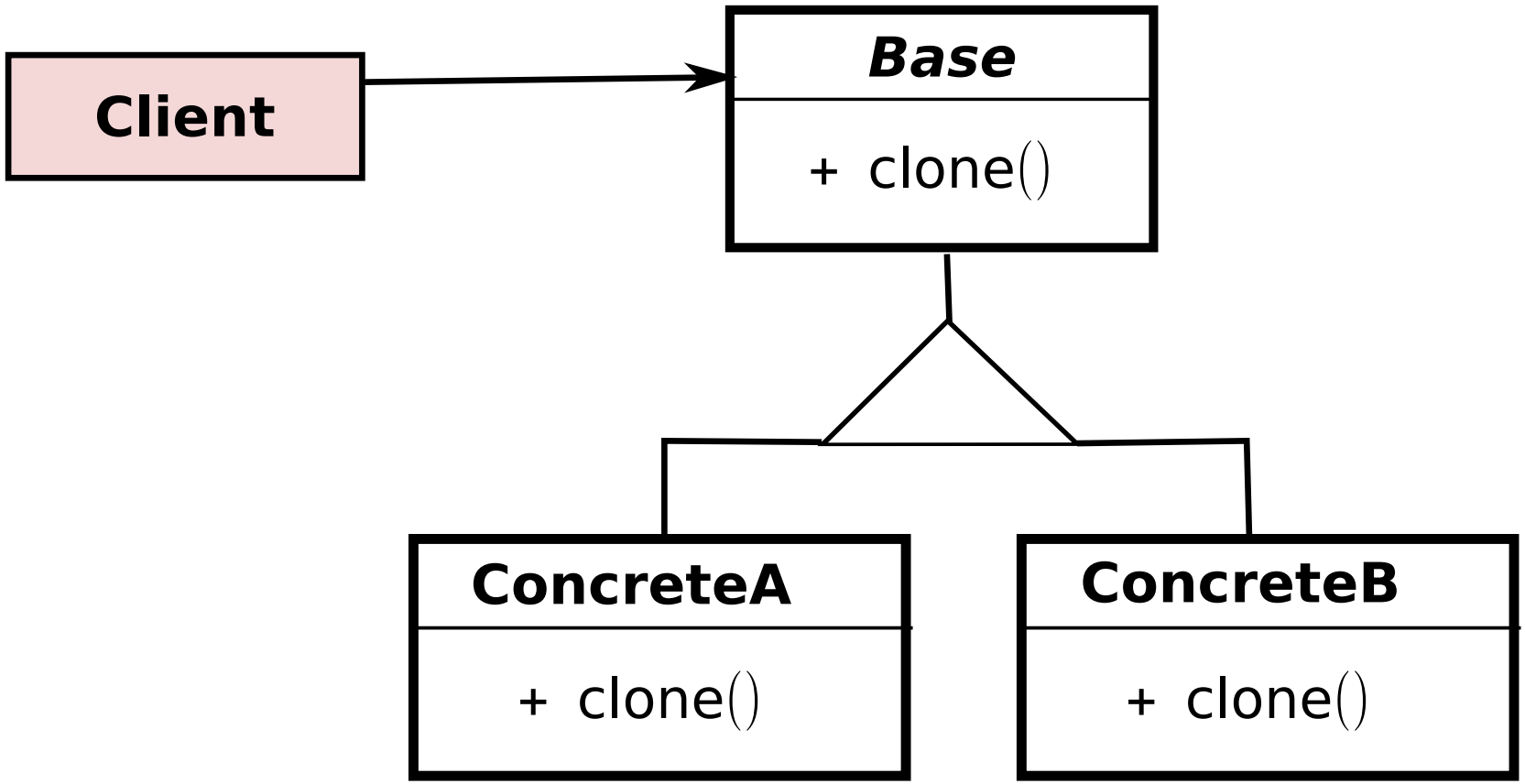
Prototyp

„głębokia” i „płytka” 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; //Płytka 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



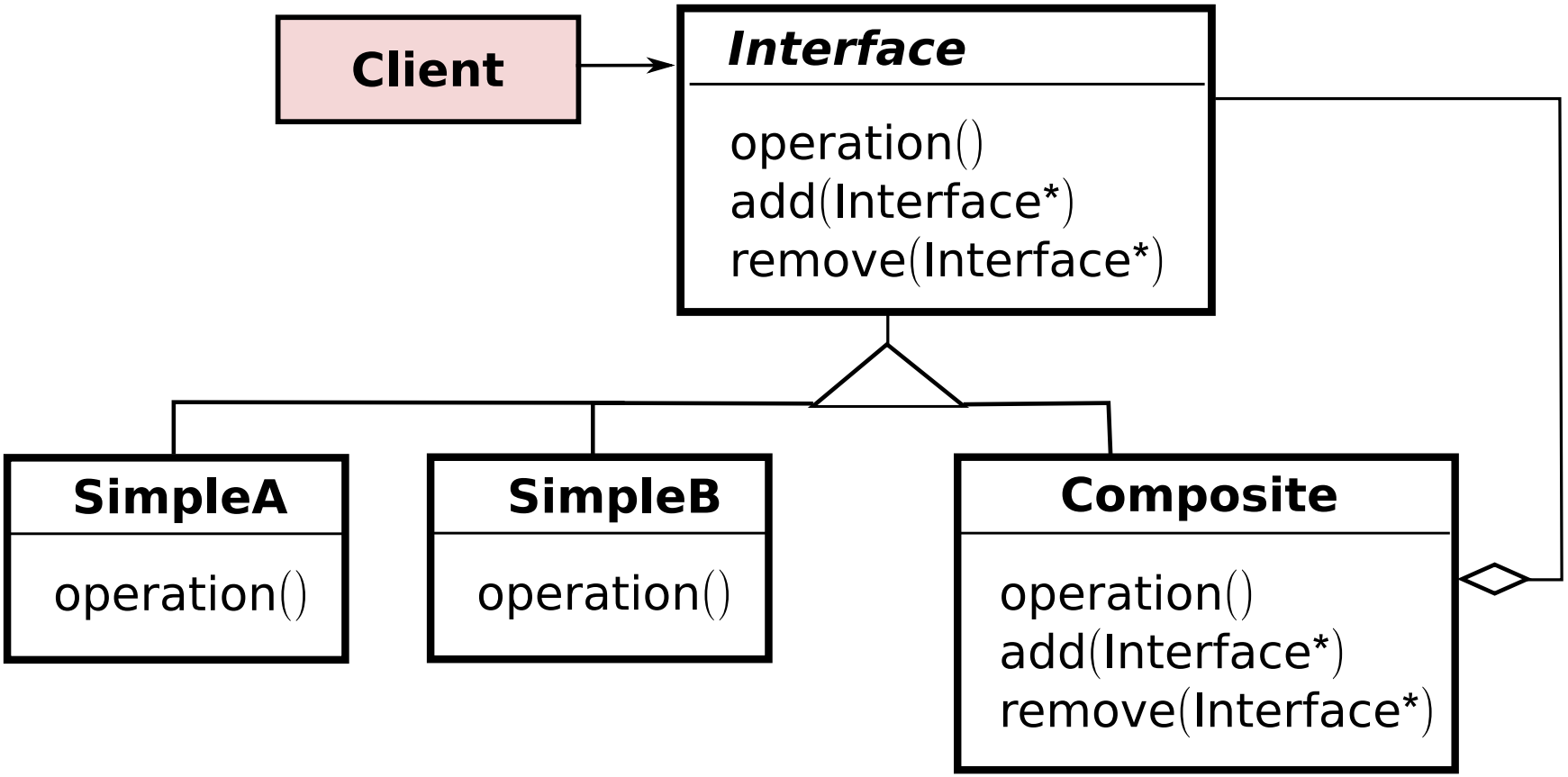
prototyp (clone, wirtualny konstruktor) - przykład

```
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
  B* clone() const override {
    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

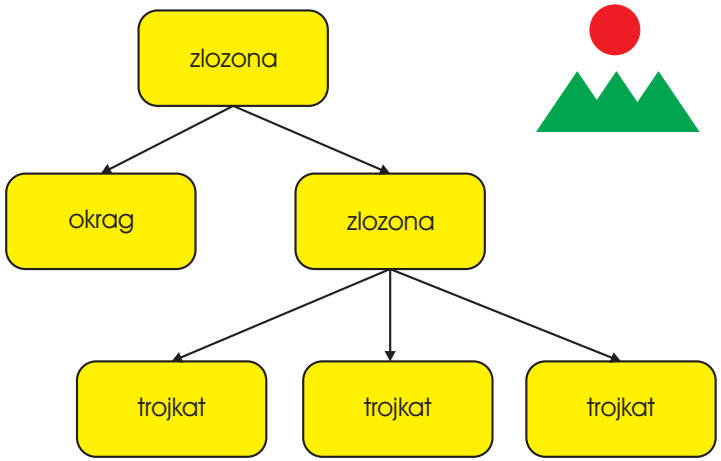
Wzorzec kompozytu

- 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:
  void draw() override; //rysuje
okrąg
};
class Composite : public Fig {
public:
  void draw() override { //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

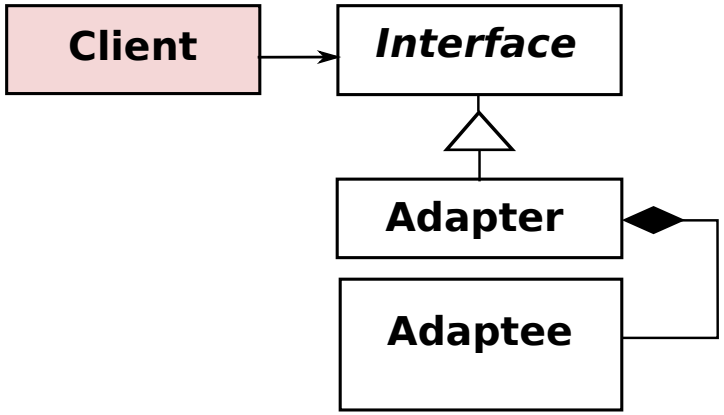
- obiekty pewnej klasy przechowują potrzebne dane
- obiekty zachowują się w odpowiedni sposób
- zmiana interfejsu jest kłopotliwa (niemożliwa)

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)

Adaptery obiektów

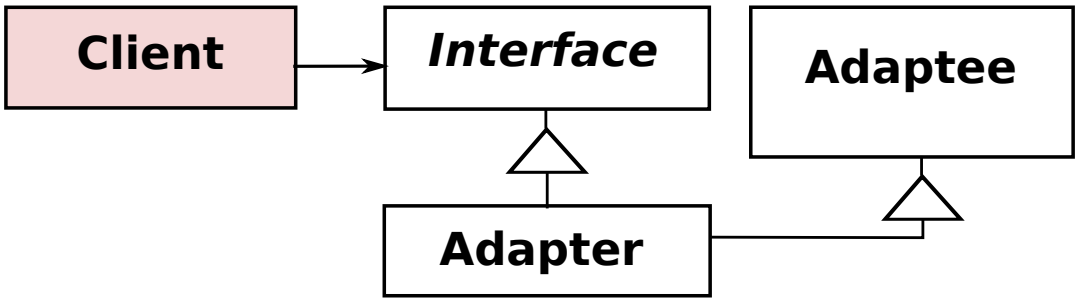


Nowa klasa (adapter)

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

```
//Przykład adaptera obiektów
class Adapter : public Interfejs {
public:
  int metoda1() override { return obj_.metoda2(); }
private:
  Adaptowany obj_;
};
```

Adaptery klas



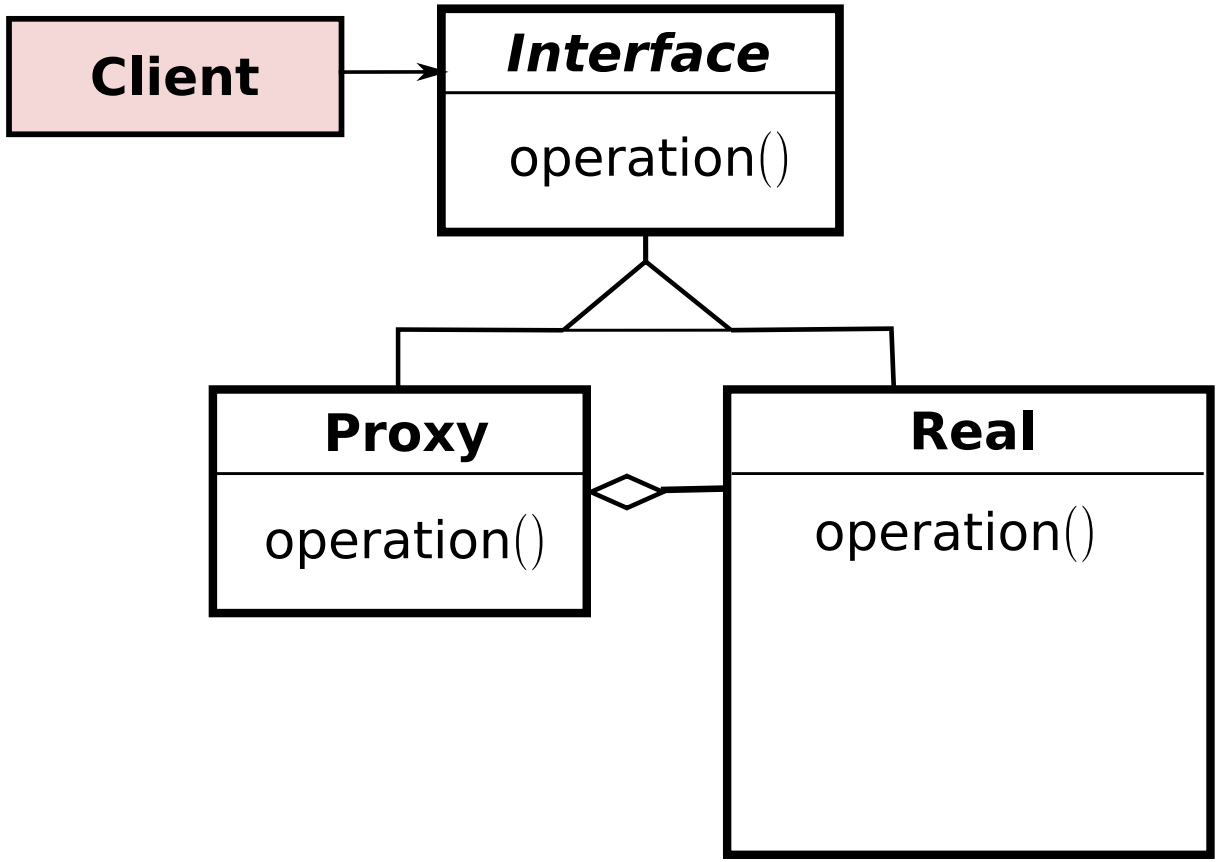
Nowa klasa (adapter)

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

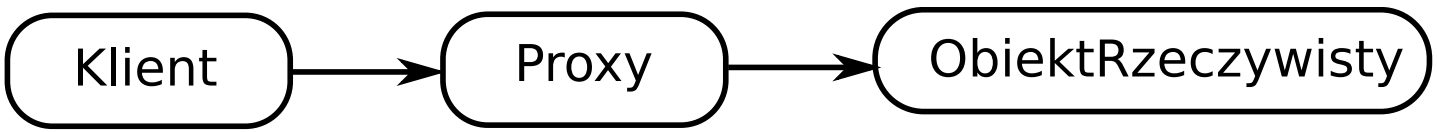
```
//Przykład adaptera klas
class Adapter : public Interfejs, private Adoptowany {
public:
  int metoda1() override { return metoda2(); }
};
```

Pośrednik (proxy)

Wzorzec proxy



Proxy to uchwyt (wskaźnik) do obiektu.



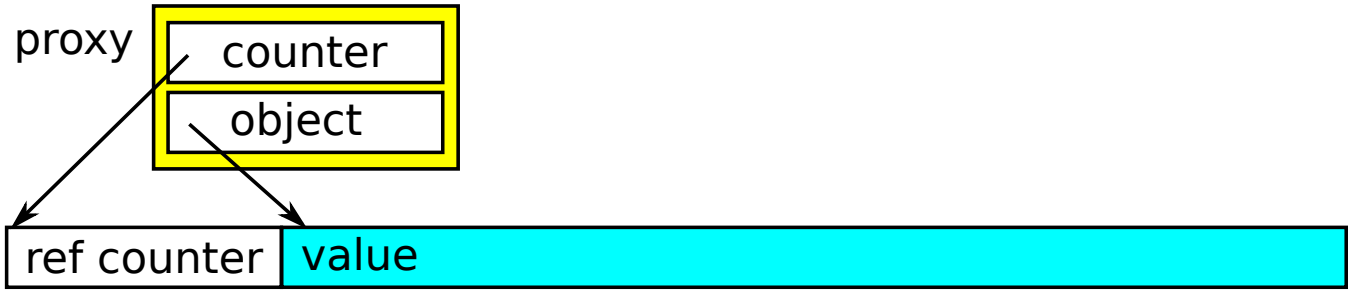
Wzorzec proxy - rodzaje

- 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) {}
  void draw() override { getImage()->draw(); } //pośredniczy
private:
  Image* real_; //uchwyt
  Image* getImage() {
    if(!real_) real_ = new RealImage();
    return real_;
  }
};
```

Copy-On-Write Proxy, leniwe kopiowanie

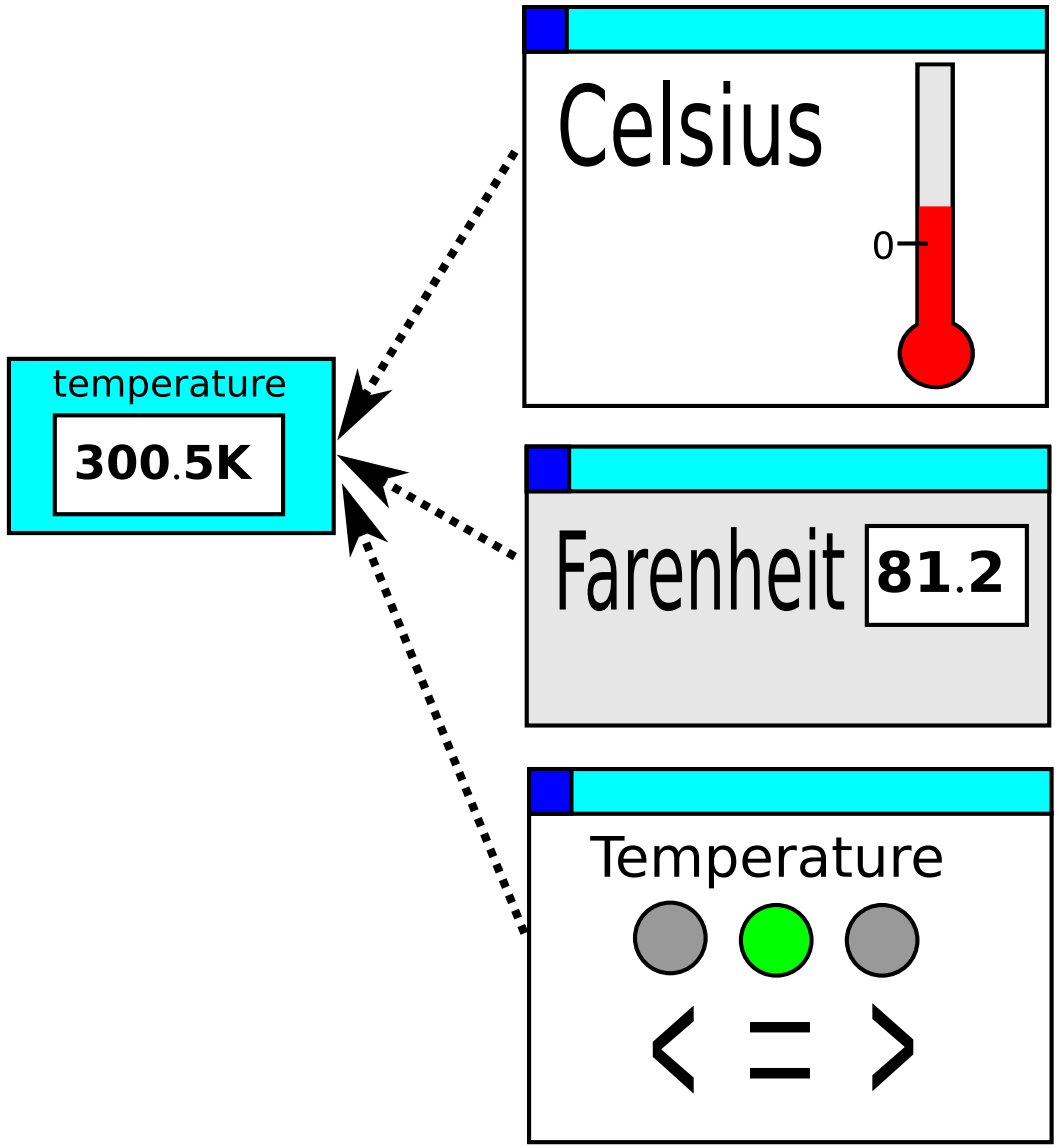



```
class CFoo { //leniwe kopiowanie dla Foo, Foo zawiera licznik
public:
    CFoo(const CFoo& c) { join(c.foo_); }
    ~CFoo() { unjoin(); }
    int get() const { return foo_->val_; }
    void set(int v) {
        if(foo_->count_ == 1) foo_->val_ = v;
        else { --foo_->count_; foo_ = new Foo(v);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

Obserwator

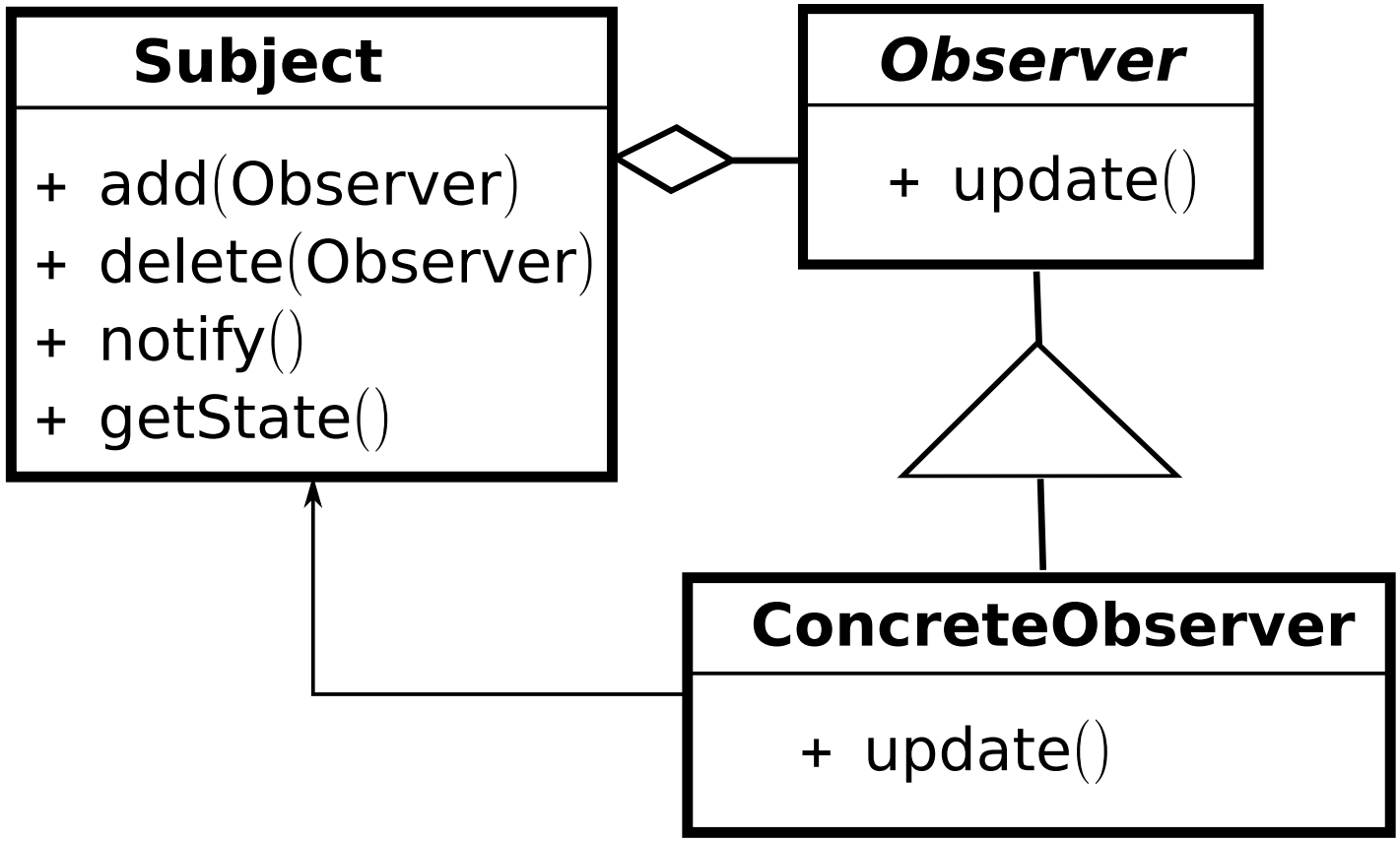
Wzorzec obserwatora

- 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

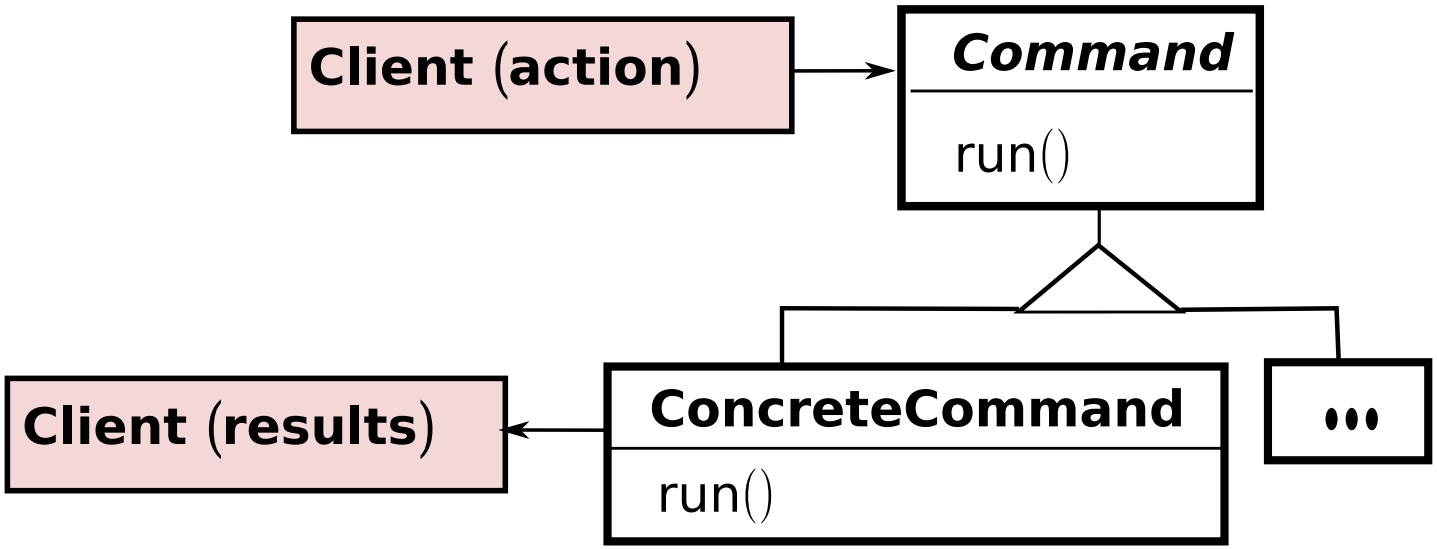
- Signal = Subject
- Slot = Observer

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.

Dziękuję

robert.nowak@pw.edu.pl