

# Sztuka Wytwarzania Oprogramowania

## Wykład 4 - obiektowe wzorce projektowe, cz 2

Robert Nowak

24Z

# *Obiektowe wzorce projektowe*

standardowe, sprawdzone w praktyce, rozwiązania często pojawiających się problemów projektowych

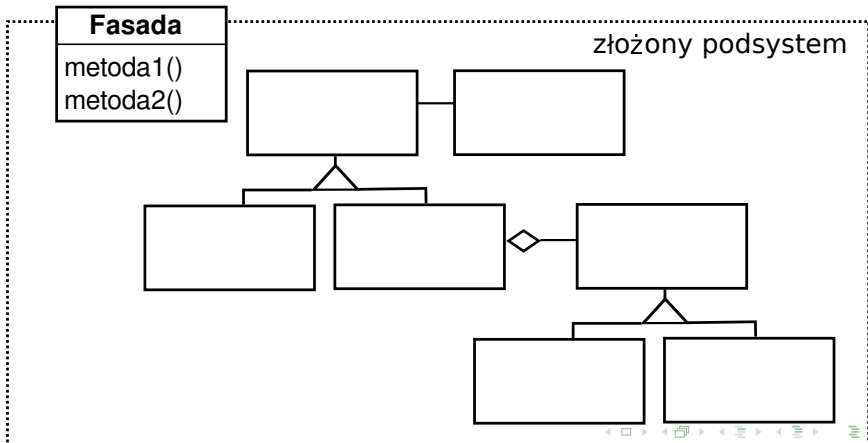
# Plan wykładu

- ▶ fabryki
  - ▶ fabryka skalowalna, fabryka prototypów
  - ▶ fabryka abstrakcyjna
  - ▶ singleton
- ▶ wizytator
- ▶ wielometoda
- ▶ most
  - ▶ ukrywanie implementacji (pImpl)
  - ▶ wstrzykiwanie zależności

# *Fasada*

# Wzorzec fasady

- ▶ uproszczenie korzystania ze złożonego systemu
- ▶ zapewnienie wymienności podsystemu



# *Fabryka skalowalna, fabryka prototypów*

## Różnice między inicjacją obiektu a jego użyciem

Polimorfizm (dziedziczenie, funkcje wirtualne)

- ▶ pozwala tworzyć abstrakcje i posługiwać się nimi
- ▶ wspomaga wielokrotne wykorzystanie kodu i przenośność

Mechanizmy te nie są dostępne podczas inicjacji

```
class Bazowa { /* ... */ };  
class KonkretnaA : public Bazowa { /* ... */ };  
class KonkretnaB : public Bazowa { /* ... */ };
```

```
Bazowa* pb = new KonkretnaA; //Trzeba podać typ
```

Inicjacja:

- ▶ wymaga podania konkretnego typu
- ▶ typ musi być znany w momencie kompilacji

# Fabryki obiektów

Gdy istnieje informacja o typie, ale forma jest nieodpowiednia dla kompilatora (przykład: odczyt/zapis obiektów z/do pliku)

```
class Figura {
public:
    enum { KWADRAT, KOLO, /* ... */ };
    virtual bool zapisz(std::ostream& os) = 0;
    /* ... */
};
class Kwadrat : public Figura { /* ... */

bool Kwadrat::zapisz(std::ostream& os) override {
    os << KWADRAT;
    //zapisuje poszczególne składowe
};
```



# Prosta fabryka obiektów

```
Figura* create(std::istream& is) {  
    int typ;  
    is >> typ;  
    Figura* obj; //wczytywany obiekt  
    switch(typ) {  
        case KWADRAT:  
            obj = new Kwadrat();  
            break;  
        case KOLO: //...  
        default: //błąd - nieznany typ  
    }  
    obj->read(is); //wczytuje składowe  
    return obj;  
}
```

- ▶ problemy przy dodawaniu nowych klas konkretnych
- ▶ zależność od wszystkich klas w hierarchii

# Fabryka skalowalna

każdy typ konkretny sam się rejestruje (brak zależności)

```
class FigFactory {
public:
    using CreateFigFun = Figura* (*)();
    //Rejestruje nowy typ w fabryce
    bool RegisterFig(int id, CreateFigFun fun);
    //Tworzy obiekt danego typu
    Figura* create(int id);
private:
    using Callbacks = std::map<int, CreateFigFun>;
    Callbacks callbacks_;
};
//Kreator dla danego typu
Figura* CreateKwadratFun() {
    return new Kwadrat;
};
```

## Fabryka skalowalna (2)

```
//Implementacja metody create
Figura* FigFactory::create(int id) {
    Callbacks::const_iterator i = callbacks_.find(id);
    if(i == callbacks_.end() )
        //Błąd - nieznan typ figury
    else
        return (i->second)();
}
```

problem z identyfikatorem

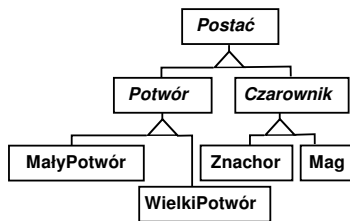
- ▶ wykorzystanie RTTI (type\_id)
- ▶ generowanie identyfikatorów

## Fabryka prototypów - wzorzec prototypu (clone)

```
class FigCloneFactory {
public:
    //Rejestruje nowy typ w fabryce
    bool RegisterFig(int id, Figure* prototype);
    //Tworzy obiekt danego typu
    Figure* create(int id) {
        //Woła metodę clone na odpowiednim prototypie
        return prototypes_.find(id)->second->clone();
    }
private:
    std::map<int, Figure*> prototypes_;
};
```

# *Fabryka abstrakcyjna*

## Fabryka abstrakcyjna (abstract factory)



```

class AbstractFactoryPostac {
public:
    virtual Potwor* utworzPotwora()=0;
    virtual Czarownik* utworzCzarownika()=0;
    /* ... */
};
    
```

```

class FactoryPocztakujacy : public AbstractFactoryPostac {
    Potwor* utworzPotwora() override { return new MałyPotwor; }
    Czarownik* utworzCzarownika() override { return new Znachor; }
};

class FactoryZaawansowany : public AbstractFactoryPostac {
    Potwor* utworzPotwora() override { return new WielkiPotwor; }
    Czarownik* utworzCzarownika() override { return new Mag; }
};
    
```

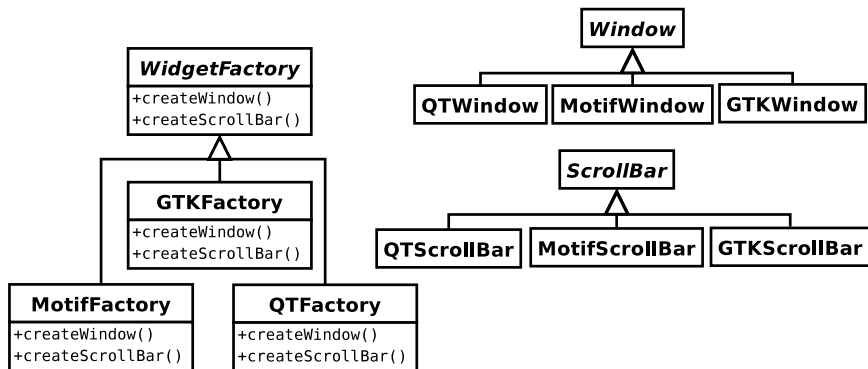
# Fabryka abstrakcyjna - przykład użycia

## Fabryka abstrakcyjna

- ▶ produkty abstrakcyjne (np. Potwor)
- ▶ produkty konkretne (np. MalyPotwor)

```
class Gra {  
    public:  
    void poziom() {  
        if(/* użytkownik wybrał poziom początkujący */)   
            pFactory_ = new FactoryPoczatkujacy;  
        else  
            pFactory_ = new FactoryZaawansowany;  
    }  
    private:  
    AbstractFactoryPostac* pFactory_;  
};
```

## Fabryka abstrakcyjna - przykład 2





# *Singleton*

# Wzorzec singletona

Obiekt globalny (problemy):

- ▶ inicjacja
- ▶ jedna kopia w programie

```
class Singleton {
public:
    static Singleton* getInstance() { //Dostęp do obiektu
        if(!pInstance_)
            pInstance_ = new Singleton;
        return pInstance_;
    }
private: Singleton(); //Prywatny konstruktor
    Singleton(const Singleton&) = delete; //zabroniony, C++11
    Singleton& operator=(const Singleton&) = delete; //zabroniony, C++11
    static Singleton* pInstance_;
};
Singleton* Singleton::pInstance_ = nullptr; //w pliku .cpp
```

# *Wizytator*

## Wzorzec wizytatora - wstęp

Modyfikacje ze względu na wprowadzane zależności:

- ▶ dodawanie klas w hierarchii - proste
- ▶ dodawanie metod do interfejsu - złożone

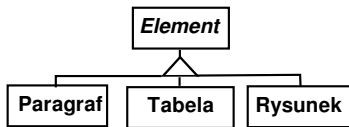
Stosuje się wzorzec wizytatora gdy:

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

**Przykład:** edytor dokumentów

Operacje:

- ▶ generowanie wydruku
- ▶ generowanie statystyk
- ▶ sprawdzanie pisowni
- ▶ ...



## Rozwiązanie (uciążliwe) - bezpośrednio

```
//Klasa do statystyk
class Stat {
public:
    void
    dodajZnaki(int z){
        znaki_ += z;
    };
    //...
private:
    int znaki_;
    int slowa_;
    int rysunki_;
};
```

```
class Element {
public: //Interfejs w klasie bazowej
    virtual void updateStat(Stat& s) = 0;
};
//Implementacja w każdej klasie konkretnej
void Paragraf::updateStat(Stat& s) {
    s.dodajZnaki(/* znaki w paragr. */);
    s.dodajSłowa(/* słowa w paragr. */);
}
void Rysunek::updateStat(Stat& s) {
    s.dodajRysunki(1);
}
```

- ▶ przetwarzanie zaimplementowane w wielu miejscach
- ▶ każda klasa w hierarchii jest zależna od klas potrzebnych do przetwarzania

## Rozwiązanie (błędne) - badanie typu

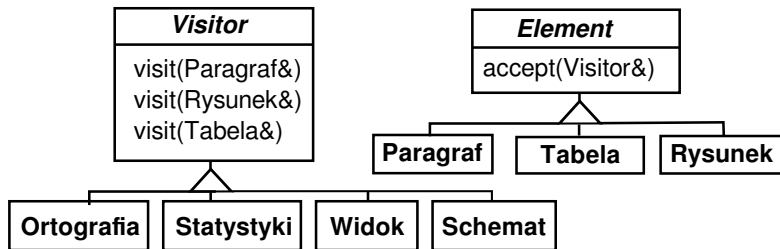
- ▶ przetwarzanie w jednym miejscu
- ▶ brak zależności od klas potrzebnych do przetwarzania

```
void Stat::Update(const Element& elem) {  
    if(const Paragraf* p = dynamic_cast<const Paragraf*>(&elem) ) {  
        znaki_ += p->znaki();  
        slowa_ += p->slowa();  
    } else if (dynamic_cast<const Rysunek*>(&elem) ) {  
        ++rysunki_;  
    }  
    /* ... */  
}
```

- ▶ problemy z badaniem typu
- ▶ czasochłonne

## Wizytator - hierarchia klas wizytujących

- ▶ Element - klasa bazowa dla odwiedzanej hierarchii klas (hierarchia rzadko zmieniana)
- ▶ Wizytator - klasa bazowa. Oddzielna metoda dla każdego z typów w wizytowanej hierarchii.



## Rozwiązanie wykorzystujące wizytator

Operacja zależna od dwu typów: typ wizytatora i typ elementu

```
class Element { //Klasa bazowa
public:
    virtual void accept(Wizytator& v) = 0;
};
class Paragraf : public Element {
public:
    void accept(Wizytator& v) override {
        v.visit(*this); //Woła visit(Paragraf&)
    }
};
class Wizytator { //Abstrakcyjny wizytator
public:
    virtual void visit(Paragraf&) = 0;
    virtual void visit(Rysunek&) = 0;
    virtual void visit(Tabela&) = 0;
};
```



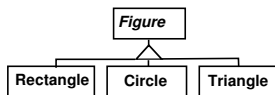
# Implementacja konkretnego wizytatora

```
//Konkretny wizytator
class Stat : public Wizytator {
public:
    void visit(Paragraf& p) override {
        znaki_ += p.znaki(); slowa_ += p.slowa();
    }
    void visit(Rysunek&) override {
        ++rysunki_;
    }
    /* ... */
};

//Obliczanie statystyk
Stat s;
for(Element* e : document->elements) //elementy dokumentu
    e->accept(s);
//Tutaj już mamy obliczone statystyki
```

# *Wielometoda*

## Wielometody



```

class Figure { //dostarcza wizytatora
public:
    virtual void accept(Visitor& v) const = 0;
};
  
```

```

struct Visitor { //wizytator bazowy
    virtual void visit(const Rectangle&) = 0;
    virtual void visit(const Circle&) = 0;
    virtual void visit(const Triangular&) = 0;
};

class Rectangle : public Figure {
public: //przykładowa implementacja metody accept
    void accept(Visitor& v) override { v.visit(*this); }
};

double intersect(const Circle& a, const Circle& b);
double intersect(const Circle& a, const Rectangle& b);
double intersect(const Rectangle& a, const Rectangle& b);
  
```

# Wielometody - implementacja bezpośrednia (uciążliwa)

```
double intersect(const Figure& a, const Figure& b) {
    if(Circle* pa = dynamic_cast<Circle*>(&a) ) {
        if(Circle* pb = dynamic_cast<Circle*>(&b) ) {
            return intersect(*pa, *pb); //tutaj już ma konkretne typy
        }
        else if(Rectangle* pb = dynamic_cast<Rectangle*>(&b) ) {
            return intersect(*pa, *pb);
            //kolejne wybory dla drugiego typu
        }
    }
    else if(Rectangle* pa = dynamic_cast<Rectangle*>(&a) ) {
        if(Circle* pb = dynamic_cast<Circle*>(&b) ) {
            return intersect(*pb, *pa);
        }
        else if(Rectangle* pb = dynamic_cast<Rectangle*>(&b) ) {
            return intersect(*pa, *pb);
            //...
        }
    }
    //kolejne wybory dla pierwszego typu
}
```

# Wielometody - użycie wizytatorów

```
double intersect(Figure& a, Figure& b) {
    IntersectVisitor visitor(a);
    b.accept(visitor);
    return visitor.value_;
}

struct IntersectVisitor : public Visitor { //rozstrzyga dwa typy
    IntersectVisitor(Figure& fig) : fig_(fig), value_(0.0) {}
    void visit(Rectangle& r) override { //pierwszy typ to prostokąt
        RectangleVisitor rectVisitor(r); //odpowiedni wizytator
        fig_.accept( rectVisitor );
        value_ = rectVisitor.value_;
    }
    void visit(Circle& c) override { //pierwszym typem jest koło
        //kod jak wyżej, ale używa CircleVisitor
    }
    Figure& fig_;
    double value_; //wartość zwracana
};
```

## Wielometody - użycie wizytatorów (2)

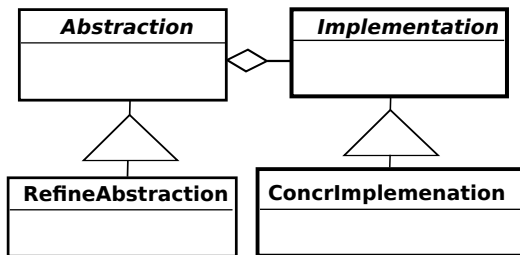
```
struct RectangleVisitor : public Visitor {
    RectangleVisitor(Rectangle& r) : r_(r), value_(0.0) {}
    void visit(Rectangle& r) override { value_ = intersect(r, r_); }
    void visit(Circle& c) override { value_ = intersect(c, r_); }
    //...
    Rectangle& r_; //pierwszy obiekt przechowywany jako typ konkretny
    double value_;
};

struct CircleVisitor : public Visitor { /* podobnie */ };

struct TriangleVisitor : public Visitor { /* podobnie */ };
```

# *Most*

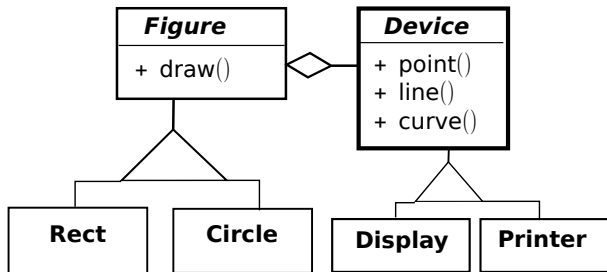
## Wzorzec Most



- ▶ separuje abstrakcję od implementacji - dwie hierarchie klas
- ▶ pozwala konfigurować powiązanie pomiędzy abstrakcją a implementacją w czasie działania



## Most - przykład



- ▶ Adapter - łączy dwie hierarchie (nie chcemy zmieniać klasy Adaptowany)
- ▶ Most - zakładamy separację, na etapie projektu

## Przykładowy problem, gdzie pomoże wzorec mostu

```
/* warcaby wloskie, niemieckie, hiszpanskie roznia sie mozliwoscia bicia
pionem do tyłu oraz mozliwoscia ruchu damki o wiele pol */
struct State {}; //stan w grze, pol. pionow i inf. kto sie rusza
using States = vector<State>;
class Player {
    virtual State nextMove(State s) = 0;
};
/* chcemy dostarczyc 3 typy gracza: gracz losowy - dowolny dozwolony ruch),
człowiek - pobiera ruch od uzytkownika, AI - wykorzystuje algorytm Mini-Max
*/
class ItalianRandomPlayer : public Player {
public:
    State nextMove(State s) override;
private:
    States validMovementsItalian(const State& s) const;
};
/* i podobnie 8 dodatkowych klas: ItalianHumanPlayer, ... */
```

# Wzorzec „pimpl” (zdegenerowany Most)

minimalizuje zależności, ukrywa implementację

```
//PLIK FOO.H
class Foo {
public:
    //Interfejs klasy
private:
    struct Impl;//Deklaracja
    Impl* pImpl_; //uchwyt, najczęściej std::unique_ptr
};
//PLIK FOO.CPP
struct Foo::Impl {
    //składowe oraz metody prywatne
}
Foo::Foo() { /* inicjacja i implementacja */ }
Foo::~~Foo() { delete pImpl_; /* niepotrzebne gdy unique_ptr */}
```

## Wzorzec projektowy, wstrzykiwanie zależności

wstrzykiwanie zależności (Dependency injection) - wzorzec projektowy, zmniejszenie powiązań pomiędzy obiektami.

Uchwyt do obiektu w konstruktorze, zamiast składowej.

```
class NeedsFoo {
public:
    NeedsFoo(Foo* foo) : foo_(foo) {}
private:
    Foo* foo_;
};
//przykład użycia, obiekt wstrzykiwany zarządzany przez klasę pochodną
class ConcreteNeedsFoo : public NeedsFoo {
    ConcreteNeedsFoo() : NeedsFoo(&foo_) {}
private:
    Foo foo_;
};
```

## Powtórzenie - wzorce projektowe

### Kreacyjne:

- ▶ **fabryka abstrakcyjna**
- ▶ **fabryka obiektów**
- ▶ **prototyp (prototype)**
- ▶ **singleton**
- ▶ metoda fabryczna

### Strukturalne:

- ▶ **adapter (wrapper)**
- ▶ **most (bridge)**
- ▶ **kompozyt (composite)**
- ▶ **dekorator (decorator)**
- ▶ **proxy (proxy)**
- ▶ **fasada (facade)**

### Czynnościowe:

- ▶ **polecenie (command)**
- ▶ **wizytator (visitor)**
- ▶ **obserwator (observer)**
- ▶ **wielometoda (multimethod)**
- ▶ iterator
- ▶ interpreter
- ▶ strategia (strategy)
- ▶ stan (state)
- ▶ mediator
- ▶ łańcuch powiązań (chain of responsibility)

# *Dziękuję*

robert.nowak@pw.edu.pl