

# TDD

Piotr Ślatała, Tomasz Żurkowski

4 grudnia 2009

# Plan prezentacji

## 1 Idea testów

# Plan prezentacji

- 1 Idea testów
- 2 TDD

# Plan prezentacji

- 1 Idea testów
- 2 TDD
- 3 Frameworki do testowania

# Plan prezentacji

- 1 Idea testów
- 2 TDD
- 3 Frameworki do testowania
- 4 Mocki

# Plan prezentacji

- 1 Idea testów
- 2 TDD
- 3 Frameworki do testowania
- 4 Mocki
- 5 Dependency Injection / kontenery

# Wstęp

## Programowanie obiektowe

W obiektowym paradygmacie programowania modeluje się projekt z użyciem obiektów - czyli elementów, które mają swój stan, oraz zachowanie, czyli metody które można na nich wykonać.

# Wstęp

## Programowanie obiektowe

W obiektowym paradygmacie programowania modeluje się projekt z użyciem obiektów - czyli elementów, które mają swój stan, oraz zachowanie, czyli metody które można na nich wykonać.

Taka technika programowania jest dla człowieka dość intuicyjna, stąd jej popularność. Wiąże ponadto dane z zachowaniem, czego paradygmat proceduralny nie oferował.



# Przykład proceduralny oraz OOP

```
int dlugosc = dlugoscTekstu(string);  
string kawalek = kawalekTekstu(string, od, do);  
bool porownanie = porownajDwaTeksty(string1, string2);
```

```
int dlugosc = string.Dlugosc();  
string kawalek = string.Wytnij(od, do);  
bool porownanie = string1.JestRowny(string2);
```

# Modelowanie świata

# Modelowanie świata

# Obiekty:

Obiekty:

- Koszyk

## Obiekty:

- Koszyk
- Owoc

## Obiekty:

- Koszyk
- Owoc
  - Jabłko
  - Gruszka

# Cechy programowania obiektowego

- *Abstrakcja*
- Enkapsulacja
- Polimorfizm
- Dziedziczenie



# Klasa

Definiuje stan i zachowanie rodzaju obiektu. Jej wystąpienie nazywamy "instancją" bądź "obiektem".

# Klasa

Definiuje stan i zachowanie rodzaju obiektu. Jej wystąpienie nazywamy "instancją" bądź "obiektem".

- Pola / atrybuty

# Klasa

Definiuje stan i zachowanie rodzaju obiektu. Jej wystąpienie nazywamy "instancją" bądź "obiektem".

- Pola / atrybuty
- Metody

# Cykl życia obiektu

Każdy obiekt musi kiedyś zostać utworzony i kiedyś się zakończyć.

---

<sup>1</sup>Chociaż nie zawsze! W językach takich jak Java czy C# za zwolnienie pamięci odpowiada Garbage Collector

# Cykl życia obiektu

Każdy obiekt musi kiedyś zostać utworzony i kiedyś się zakończyć.

- Utworzenie obiektu = wywołanie konstruktora

---

<sup>1</sup>Chociaż nie zawsze! W językach takich jak Java czy C# za zwolnienie pamięci odpowiada Garbage Collector

# Cykl życia obiektu

Każdy obiekt musi kiedyś zostać utworzony i kiedyś się zakończyć.

- Utworzenie obiektu = wywołanie konstruktora
- Odwołanie się do dowolnej ilości metod i pól

---

<sup>1</sup>Chociaż nie zawsze! W językach takich jak Java czy C# za zwolnienie pamięci odpowiada Garbage Collector

# Cykl życia obiektu

Każdy obiekt musi kiedyś zostać utworzony i kiedyś się zakończyć.

- Utworzenie obiektu = wywołanie konstruktora
- Odwołanie się do dowolnej ilości metod i pól
- Usunięcie obiektu = wywołanie destruktor (zwolnienie pamięci!) <sup>1</sup>

---

<sup>1</sup>Chociaż nie zawsze! W językach takich jak Java czy C# za zwolnienie pamięci odpowiada Garbage Collector

# Przykładowa klasa

```
class PrzykladowaKlasa {  
public:  
    void Wyswietl() {  
        cout << "wyswietlam\n";  
    }  
};  
  
int main() {  
    PrzykladowaKlasa przyklad;  
    PrzykladowaKlasa* przykladWskaznik;  
    przykladWskaznik = new PrzykladowaKlasa();  
    przyklad.Wyswietl();  
    przykladWskaznik->Wyswietl();  
    delete(przykladWskaznik);  
}
```



# Przykładowa klasa, nie inline

```
class PrzykładowaKlasa
{
    public:
        void Wyswietl();
};
void PrzykładowaKlasa::Wyswietl()
{
    cout << "wyswietlam\n";
}
```

# Tworzenie obiektów i wskaźniki

W kilku sytuacjach będziemy się posługiwać pojęciem wskaźnika.

# Tworzenie obiektów i wskaźniki

W kilku sytuacjach będziemy się posługiwać pojęciem wskaźnika.

- Słowo kluczowe **new** odpowiada za utworzenie obiektu

# Tworzenie obiektów i wskaźniki

W kilku sytuacjach będziemy się posługiwać pojęciem wskaźnika.

- Słowo kluczowe **new** odpowiada za utworzenie obiektu
- NazwaTypu\* - gwiazdka oznacza wskaźnik do danej struktury. Wskaźnik sam w sobie nie przechowuje żadnych danych, poza informacją, gdzie dana struktura w pamięci się znajduje.

# Tworzenie obiektów i wskaźniki

W kilku sytuacjach będziemy się posługiwać pojęciem wskaźnika.

- Słowo kluczowe **new** odpowiada za utworzenie obiektu
- NazwaTypu\* - gwiazdka oznacza wskaźnik do danej struktury. Wskaźnik sam w sobie nie przechowuje żadnych danych, poza informacją, gdzie dana struktura w pamięci się znajduje.
- delete(nazwaObiektu) - służy do usunięcia obiektu z pamięci.

# Tworzenie obiektów i wskaźniki

W kilku sytuacjach będziemy się posługiwać pojęciem wskaźnika.

- Słowo kluczowe **new** odpowiada za utworzenie obiektu
- NazwaTypu\* - gwiazdka oznacza wskaźnik do danej struktury. Wskaźnik sam w sobie nie przechowuje żadnych danych, poza informacją, gdzie dana struktura w pamięci się znajduje.
- delete(nazwaObiektu) - służy do usunięcia obiektu z pamięci.

```
PrzykładowaKlasa* przykladWskaznik;
```

```
przykladWskaznik = new PrzykładowaKlasa();
```

```
przykladWskaznik->Wyswietl();
```

```
delete( przykladWskaznik );
```

# Specyfikatory dostępu

Elementy klasy mogą być zdefiniowane dla różnych poziomów dostępu

# Specyfikatory dostępu

Elementy klasy mogą być zdefiniowane dla różnych poziomów dostępu

- **public** - widoczne dla wszystkich



# Specyfikatory dostępu

Elementy klasy mogą być zdefiniowane dla różnych poziomów dostępu

- **public** - widoczne dla wszystkich
- **protected** - widoczne dla składowych klasy i potomnych

# Specyfikatory dostępu

Elementy klasy mogą być zdefiniowane dla różnych poziomów dostępu

- **public** - widoczne dla wszystkich
- **protected** - widoczne dla składowych klasy i potomnych
- **private** - widziane wyłącznie przez składowe klasy

# Specyfikatory dostępu

```
class Nazwa // : public InnaNazwa
{
public:
    int ZmiennaPubliczna;
    void ProceduraPubliczna();
protected:
    double ZmiennaProtected;
    void ProceduraProtected();
private:
    double ZmiennaPrywatna;
    void ProceduraPrywatna();
};
```

# Specyfikatory dostępu - public

```
int main() {  
    Nazwa obiekt;  
    obiekt.ProceduraPubliczna();  
    cout << obiekt.ZmiennaPubliczna << "\n";  
    //printf("%d\n", obiekt.ZmiennaPubliczna);  
  
    //ponizsze spowoduja blad kompilacji  
    //objekt.ProceduraProtected();  
    //objekt.ProceduraPrivate();  
}
```

# Specyfikatory dostępu - protected

```
class Nazwa2 : public Nazwa {  
    ...  
    void Metoda() {  
        this->ProceduraPubliczna();  
        cout << this->ZmiennaPubliczna << "\n";  
        this->ProceduraProtected();  
  
        //ponizsze spowoduje blad kompilacji  
        //this->ProceduraPrivate();  
    }  
}
```

# Specyfikatory dostępu - private

```
class Nazwa {  
    ...  
    void Metoda() {  
        this->ProceduraPubliczna();  
        cout << obiekt.ZmiennaPubliczna << "\n";  
        this->ProceduraProtected();  
        this->ProceduraPrivate();  
    }  
}
```

- Klasa abstrakcyjna - to klasa, której nie można utworzyć instancji

## Klasa abstrakcyjna, metoda wirtualna, dziedziczenie

- Klasa abstrakcyjna - to klasa, której nie można utworzyć instancji
- Metoda wirtualna - może być zdefiniowana w klasie potomnej



## Klasa abstrakcyjna, metoda wirtualna, dziedziczenie

- Klasa abstrakcyjna - to klasa, której nie można utworzyć instancji
- Metoda wirtualna - może być zdefiniowana w klasie potomnej
- Dziedziczenie

# Dziedziczenie i klasa abstrakcyjna - przykład

```
class Owoc {  
public:  
    bool CzyWarzywo() {  
        return false;  
    }  
    virtual bool CzyJadalne() = 0;  
};  
class Jablko : public Owoc {  
public:  
    bool CzyJadalne() {  
        return true;  
    }  
};
```

# Próba utworzenia wystąpienia klasy abstrakcyjnej

# Próba utworzenia wystąpienia klasy abstrakcyjnej

```
In function 'int _main()':  
error: cannot declare variable 'owoc' to be of  
abstract type 'Owoc' because the following  
virtual functions are pure within 'Owoc':  
virtual void Owoc::PodajNazwe()
```

Klasa abstrakcyjna, metoda wirtualna, dziedziczenie

# Konstruktor

# Konstruktor

```
class Owoc
{
public:
    Owoc()
    {
        //na początku nasz owoc nie jest zjedzony,
        //ani obrany
        czyZjedzone = false;
        czyObrany = false;
        cout << "Tworzenie Owoca\n";
    }
};
```

# Diagram klas

Przykładowy diagram:

# 1 Owoc



## Przykładowy program

- 1 Owoc
- 2 Jabłko

## Przykładowy program

- 1 Owoc
- 2 Jabłko
- 3 Gruszka

## Przykładowy program

- 1 Owoc
- 2 Jabłko
- 3 Gruszka
- 4 Koszyk

## Przykładowy program

- 1 Owoc
- 2 Jabłko
- 3 Gruszka
- 4 Koszyk
- 5 Uruchomienie