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

1. Co rozumiemy pod pojęciem testu?

Wstęp

1. Co rozumiemy pod pojęciem testu?
2. Co i po co testujemy?

Rodzaje testów

1. Testy jednostkowe

Rodzaje testów

1. Testy jednostkowe
2. Testy integracyjne

Rodzaje testów

1. Testy jednostkowe
2. Testy integracyjne
3. Testy akceptacyjne (funkcjonalne)

Test driven development

Proces normalny

1. Design aplikacji

Test driven development

Proces normalny

1. Design aplikacji
2. Implementacja

Test driven development

Proces normalny

1. Design aplikacji
2. Implementacja
3. Testy

Test driven development

Test driven development

1. Testy

Test driven development

Test driven development

1. Testy
2. Minimum kodu

Test driven development

Test driven development

1. Testy
2. Minimum kodu
3. Design

Test driven development

Czyli innymi słowy

1. Implementujemy testy z wykorzystaniem nieistniejących obiektów.

Test driven development

Czyli innymi słowy

1. Implementujemy testy z wykorzystaniem nieistniejących obiektów.
2. Generujemy niezbędne "stuby"

Test driven development

Czyli innymi słowy

1. Implementujemy testy z wykorzystaniem nieistniejących obiektów.
2. Generujemy niezbędne "stuby"
3. Uruchamiamy testy

Test driven development

Czyli innymi słowy

1. Implementujemy testy z wykorzystaniem nieistniejących obiektów.
2. Generujemy niezbędne "stuby"
3. Uruchamiamy testy

Krok 1

RED

Test driven development

1. Implementujemy minimum kodu "przechodzącego" testy

Test driven development

1. Implementujemy minimum kodu "przechodzącego" testy

Krok 2

GREEN

Test driven development

1. Implementujemy minimum kodu "przechodzącego" testy

Krok 2

GREEN

Krok 3

Refactor

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.

[1] 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

[1] 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

[1] 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 destruktora (zwolnienie pamięci!) [1]

[1] 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 PrzykladowaKlasa
{
        public:
                void Wyswietl();
};
void PrzykladowaKlasa::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.

```
PrzykladowaKlasa* przykladWskaznik;

przykladWskaznik = new PrzykladowaKlasa();

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
  //obiekt.ProceduraProtected();
  //obiekt.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 - to klasa, której nie można utworzyć instancji
- Metoda wirtualna - może być przedefiniowana w klasie potomnej

- Klasa abstrakcyjna - to klasa, której nie można utworzyć instancji
- Metoda wirtualna - może być przedefiniowana 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()
```

Konstruktor

Konstruktor

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

Diagram klas

Przykładowy diagram:

1 Owoc

1. Owoc
2. Jabłko

1. Owoc
2. Jabłko
3. Gruszka

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

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