Kurs programowania

Wykład 13

Wojciech Macyna

Czym jest UML?

UML składa się z dwóch podstawowych elementów:

- notacja: elementy graficzne, składnia języka modelowania,
- metamodel: definicje pojęć języka i powiązania pomiędzy nimi

Z punktu widzenia modelowania ważniejsza jest notacja.
Z punktu widzenia generacji kodu – metamodel.

Sposoby patrzenia na modelowanie

Perspektywa przypadków użycia

Opisuje funkcjonalność, jaką powinien dostarczać system, widzianą przez jego użytkowników.

Perspektywa logiczna

Zawiera sposób realizacji funkcjonalności, strukturę systemu widzianą przez projektanta.

Perspektywa implementacyjna

Opisuje poszczególne moduły i ich interfejsy wraz z zależnościami; perspektywa ta jest przeznaczona dla programisty.

Sposoby patrzenia na modelowanie

Perspektywa przypadków użycia

Opisuje funkcjonalność, jaką powinien dostarczać system, widzianą przez jego użytkowników.

Perspektywa logiczna

Zawiera sposób realizacji funkcjonalności, strukturę systemu widzianą przez projektanta.

Perspektywa implementacyjna

Opisuje poszczególne moduły i ich interfejsy wraz z zależnościami; perspektywa ta jest przeznaczona dla programisty.

Sposoby patrzenia na modelowanie

Perspektywa przypadków użycia

Opisuje funkcjonalność, jaką powinien dostarczać system, widzianą przez jego użytkowników.

Perspektywa logiczna

Zawiera sposób realizacji funkcjonalności, strukturę systemu widzianą przez projektanta.

Perspektywa implementacyjna

Opisuje poszczególne moduły i ich interfejsy wraz z zależnościami; perspektywa ta jest przeznaczona dla programisty.

Sposoby patrzenia na modelowanie

Perspektywa procesowa

Zawiera podział systemu na procesy (czynności) i procesory (jednostki wykonawcze); opisuje właściwości pozafunkcjonalne systemu i służy przede także programistom i integratorom.

Perspektywa wdrożenia

Definiuje fizyczny podział elementów systemu i ich rozmieszczenie w infrastrukturze; perspektywa taka służy integratorom i instalatorom systemu.

Sposoby patrzenia na modelowanie

Perspektywa procesowa

Zawiera podział systemu na procesy (czynności) i procesory (jednostki wykonawcze); opisuje właściwości pozafunkcjonalne systemu i służy przede także programistom i integratorom.

Perspektywa wdrożenia

Definiuje fizyczny podział elementów systemu i ich rozmieszczenie w infrastrukturze; perspektywa taka służy integratorom i instalatorom systemu.

Rodzaje diagramów UML

- Diagram przypadków użycia
- Diagram pakietów
- Diagram klas
- Diagram strukturalny
- Diagram komponentów
- Diagram wdrożenia
- Diagram stanów
- Diagram aktywności (czynności)
- Diagramy interakcji (współpracy, przebiegu (sekwencji), komunikacji, przeglądu interakcji)
- Diagram przebiegów czasowych

Diagram przypadków użycia

Diagram przypadków użycia opisuje system z punktu widzenia użytkownika, pokazuje co robi system, a nie jak to robi. Diagram ten sam w sobie zazwyczaj nie daje nam zbyt wielu informacji, dlatego też zawsze potrzebna jest do niego dokumentacja w postaci dobrze napisanego przypadku użycia. Przypadki użycia są bardzo ważnym narzędziem zbierania wymagań. Diagramy przypadków użycia, mimo swojej prostoty, są bardzo przydatne, gdyż tworzą swojego rodzaju spis treści dla wymagań modelowanego systemu.

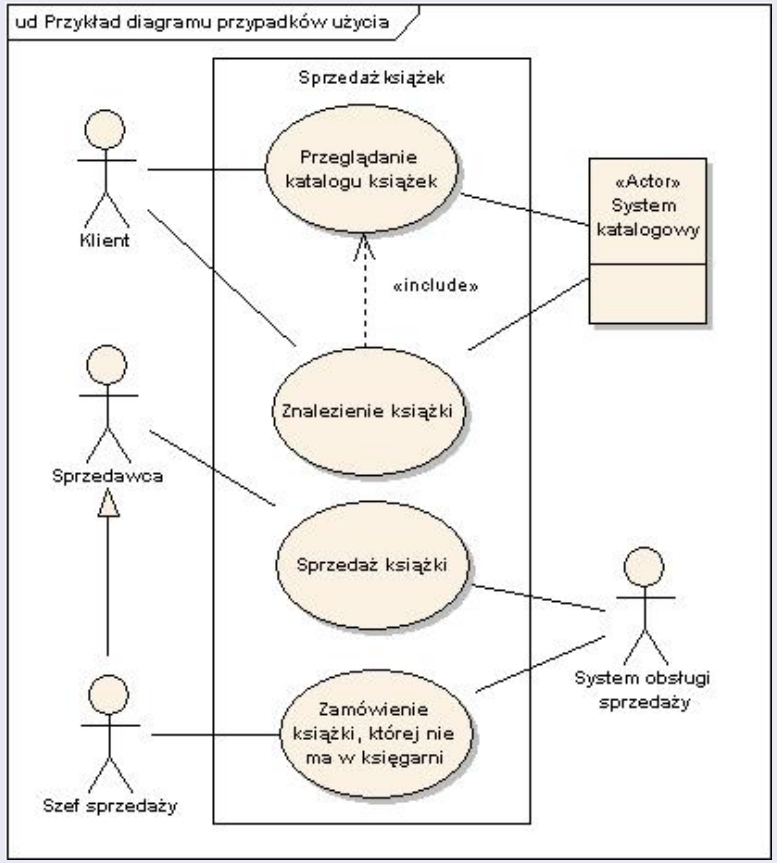


Diagram pakietów

Diagram pakietów służy do tego, by uporządkować strukturę zależności w systemie, który ma bardzo wiele klas, przypadków użycia itp. Przyjmujemy, że pakiet zawiera w sobie wiele elementów, które opisują jakieś w miarę dobrze określone zadanie. Na diagramie umieszczamy pakiety i wskazujemy na zależności między nimi. Dzięki temu dostajemy na jednym diagramie obraz całości, bądź dużego fragmentu, systemu.

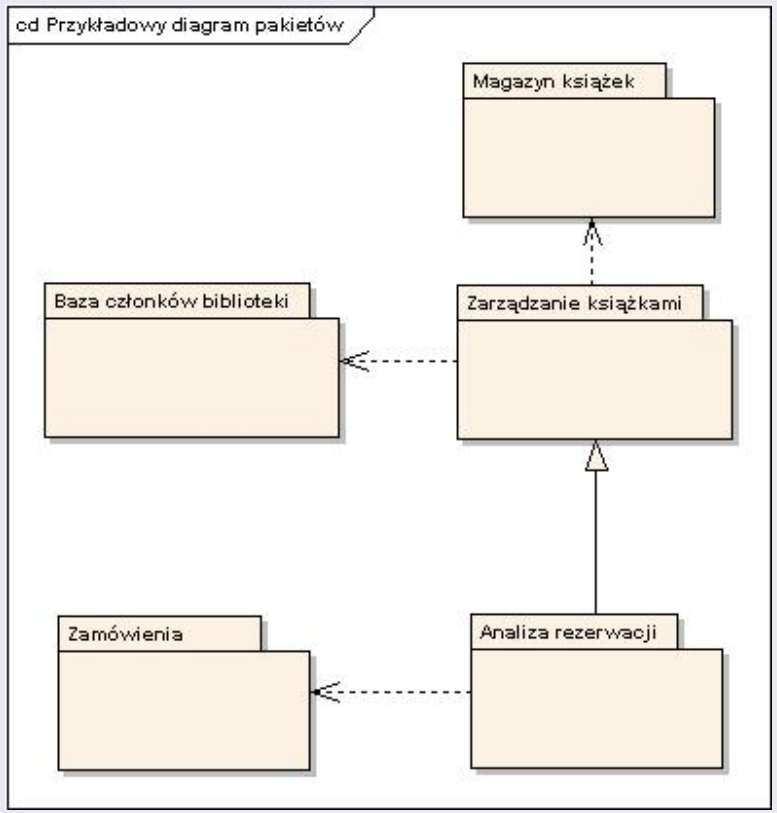
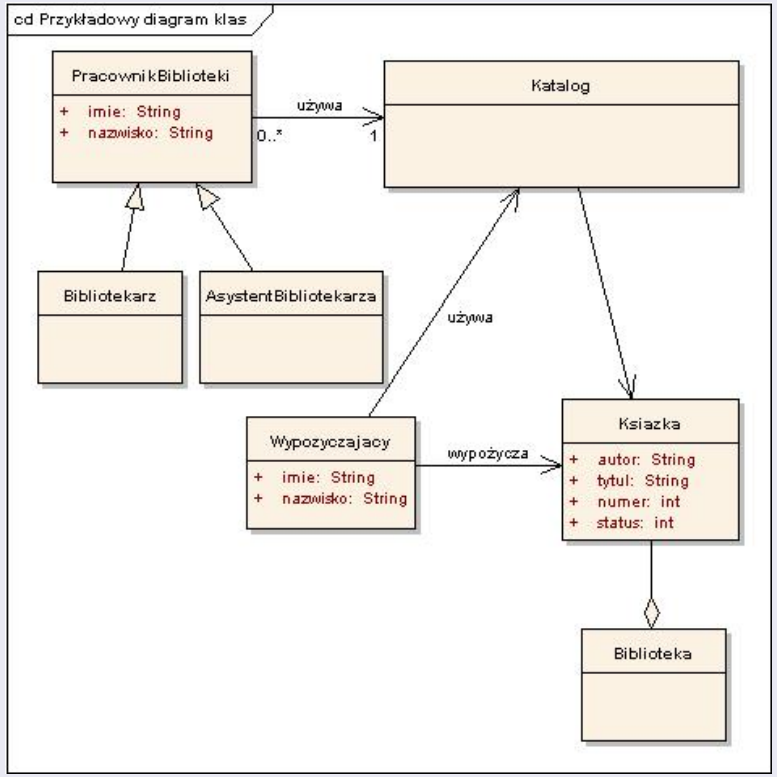


Diagram klas

Diagram klas jest ściśle powiązany z projektowaniem obiektowym systemu informatycznego lub wręcz bezpośrednio z jego implementacją w określonym języku programowania. Elementami tego diagramu są klasy, reprezentowane przez prostokąty, które mogą zawierać informację o polach i metodach klasy.



UML definiuje 4 poziomy widoczności cech i metod: + publiczny, # chroniony (klasa i jej podklasy), – prywatny, ~ publiczny wewnątrz pakietu.

Diagram klas

Diagram klas jest ściśle powiązany z projektowaniem obiektowym systemu informatycznego lub wręcz bezpośrednio z jego implementacją w określonym języku programowania. Elementami tego diagramu są klasy, reprezentowane przez prostokąty, które mogą zawierać informację o polach i metodach klasy.

UML definiuje 4 poziomy widoczności cech i metod: + publiczny, # chroniony (klasa i jej podklasy), – prywatny, ~ publiczny wewnątrz pakietu.

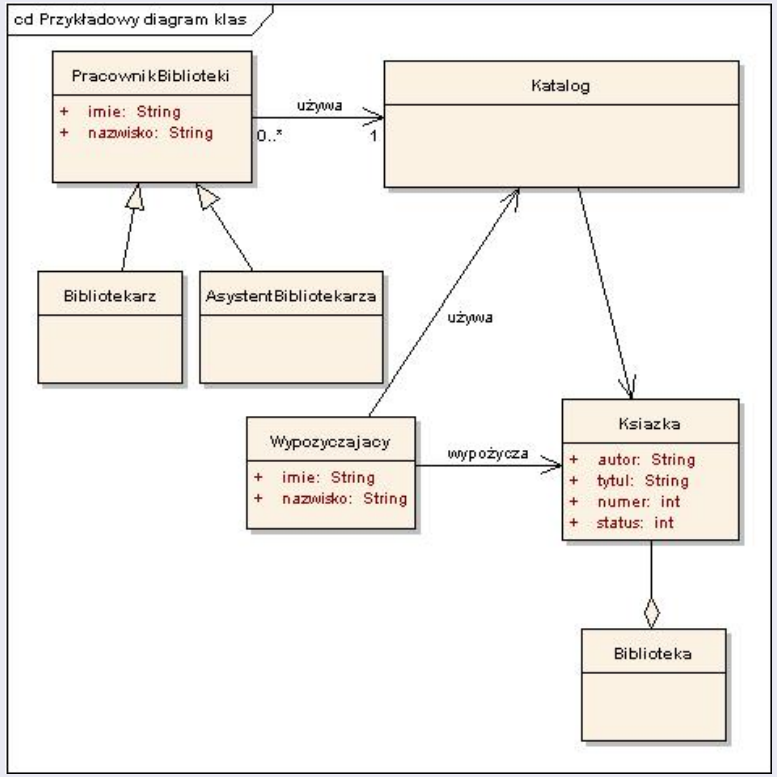


Diagram strukturalny

Jest przeznaczony do tego, by modelować współpracę klas, interfejsów, komponentów, które są zaangażowane w pewne zadanie. Diagram ten jest nieco podobny do diagramu klas, z tą różnicą, że diagram klas przedstawia statyczny obraz fragmentu systemu, a diagram strukturalny obrazuje elementy systemu wykonujące wspólne zadanie, typowe sposoby użycia elementów systemu, związki między tymi elementami, które może być trudno przedstawić na innych diagramach.

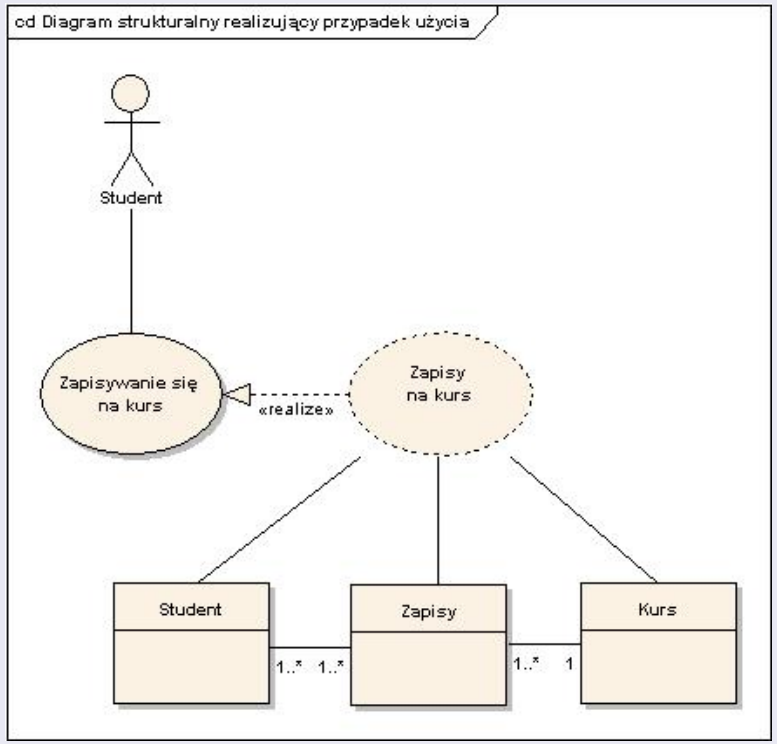


Diagram komponentów

Diagram komponentów robimy z podobnych powodów, co diagram pakietów – chcemy podzielić system na prostsze elementy i pokazać zależności między nimi. Diagram pakietów koncentrował się na podziale systemu z logicznego punktu widzenia, diagram komponentów z kolei dzieli system na fizyczne elementy oprogramowania: pliki, biblioteki, gotowe, wykonywalne programy itp.

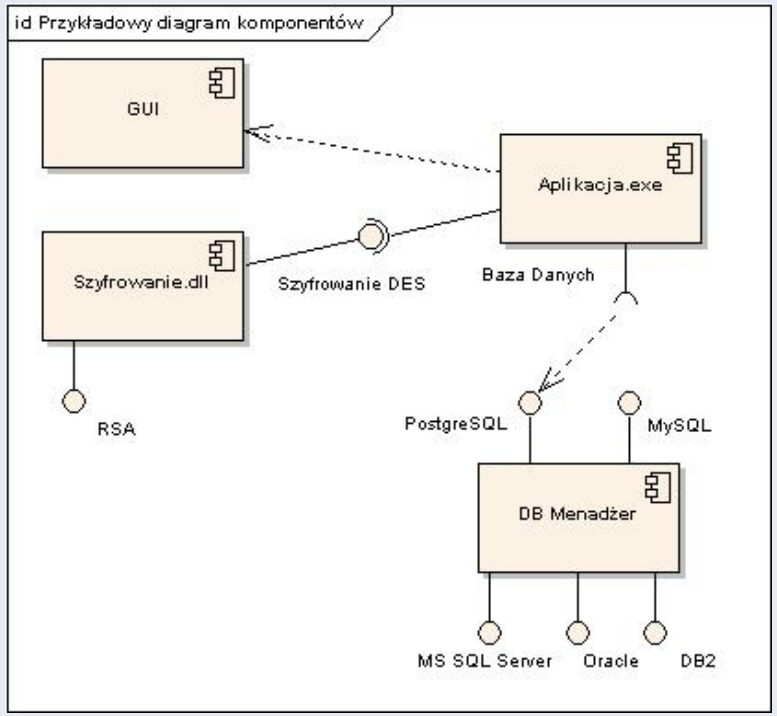


Diagram wdrożenia

Diagram wdrożenia pokazuje, jak będzie wyglądało wdrożenie i konfiguracja naszego oprogramowania.

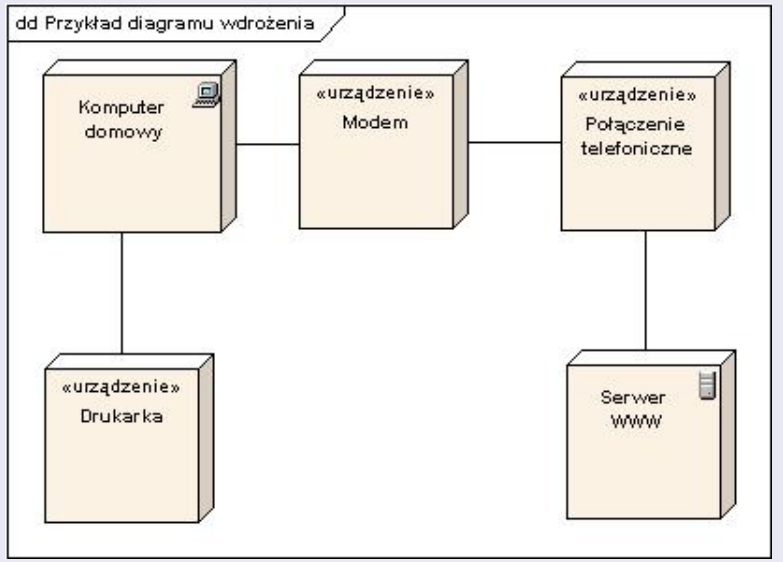


Diagram stanów

Diagram stanów z służy do tego, by pokazać w jakich stanach mogą być obiekty.

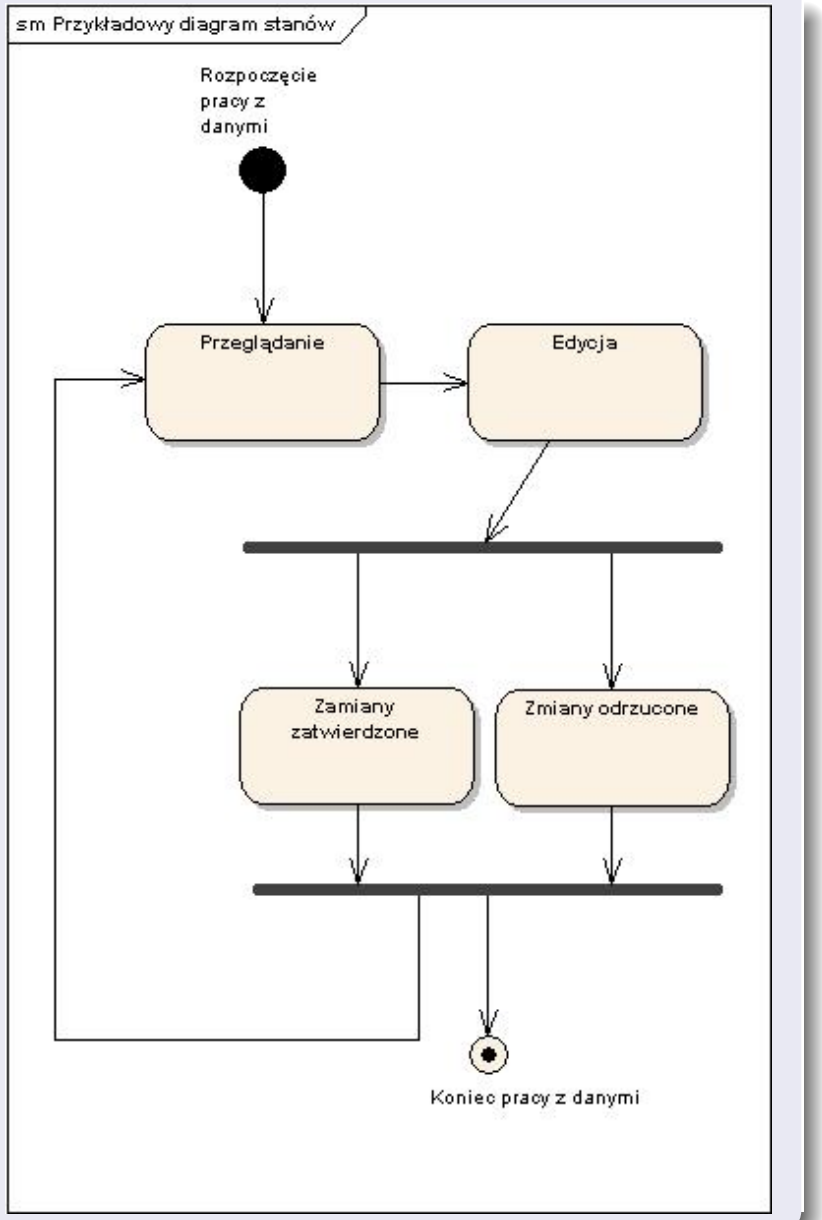


Diagram aktywności

Diagram aktywności jest pewną mutacją diagramu stanów, z tą różnicą, że diagram aktywności skupia się raczej na opisaniu jakiegoś procesu, w którym uczestniczy wiele obiektów, zaś diagram stanów pokazuje, jakie są możliwe stany konkretnego obiektu. Diagram aktywności jest bardzo dobrym narzędziem, gdy chcemy przedstawić odpowiedzialność obiektów w ramach jakiegoś procesu.

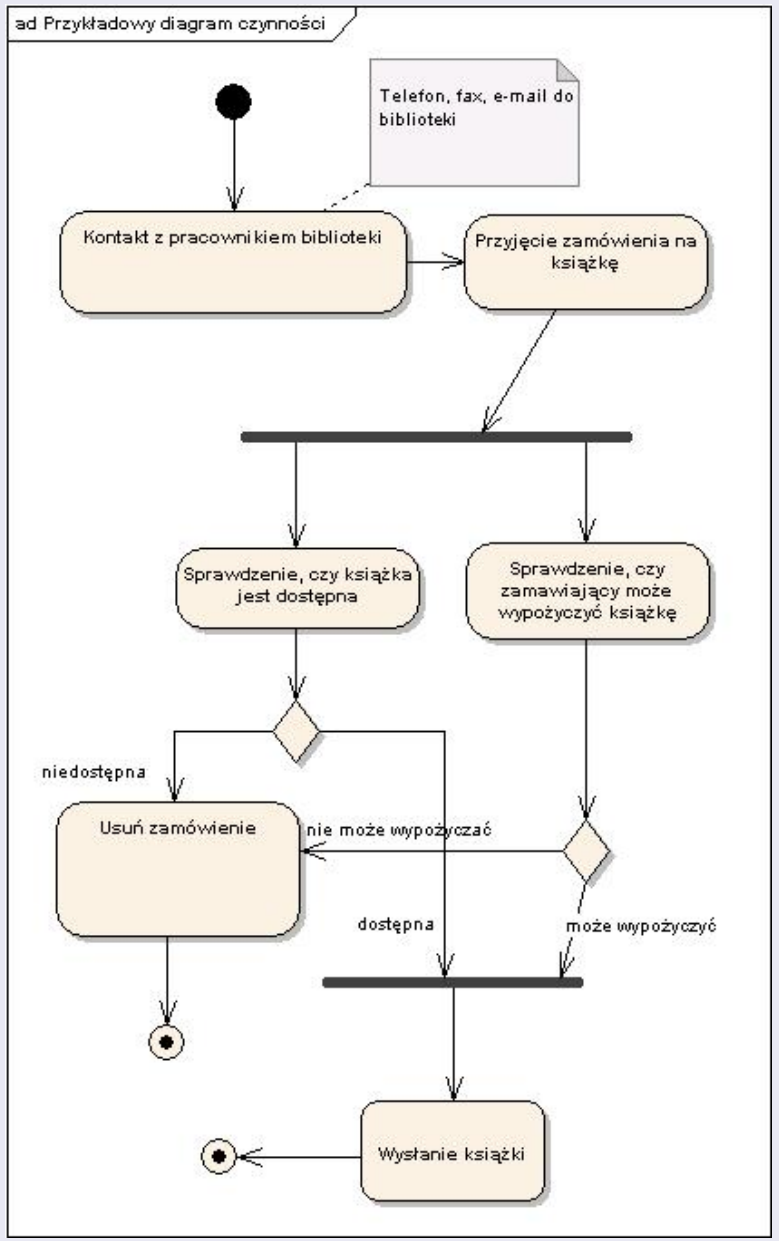


Diagram współpracy (komunikacji)

Diagram współpracy jest jednym z diagramów interakcji. Używamy go po to, żeby zobrazować dynamikę systemu – wzajemne oddziaływanie na siebie obiektów oraz komunikaty, jakie między sobą przesyłają.

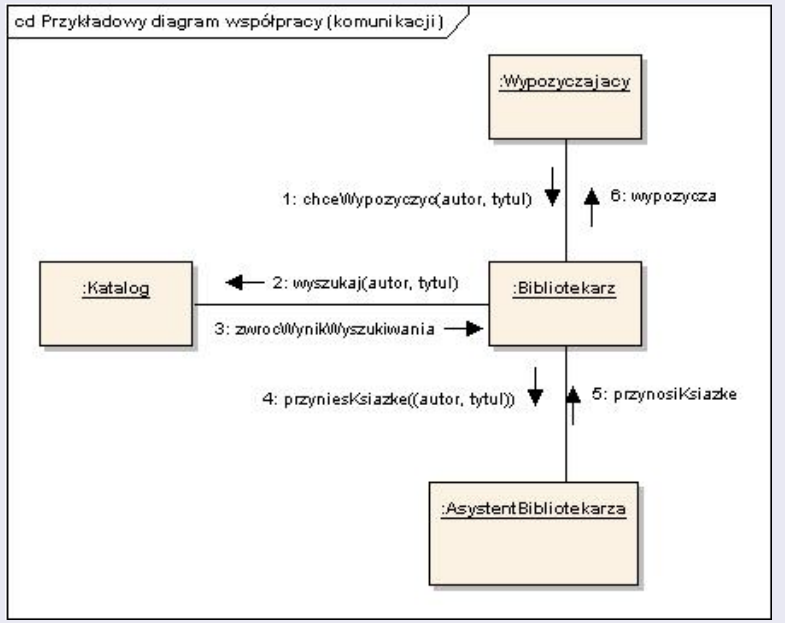


Diagram przebiegu (sekwencji)

Analogiczną informację do diagramu komunikacji zawiera drugi z diagramów interakcji, diagram przebiegu. Diagram komunikacji koncentrował się na zobrazowaniu współpracy między obiektami, teraz chcemy pokazać kolejność przesyłania komunikatów i czas istnienia obiektów.

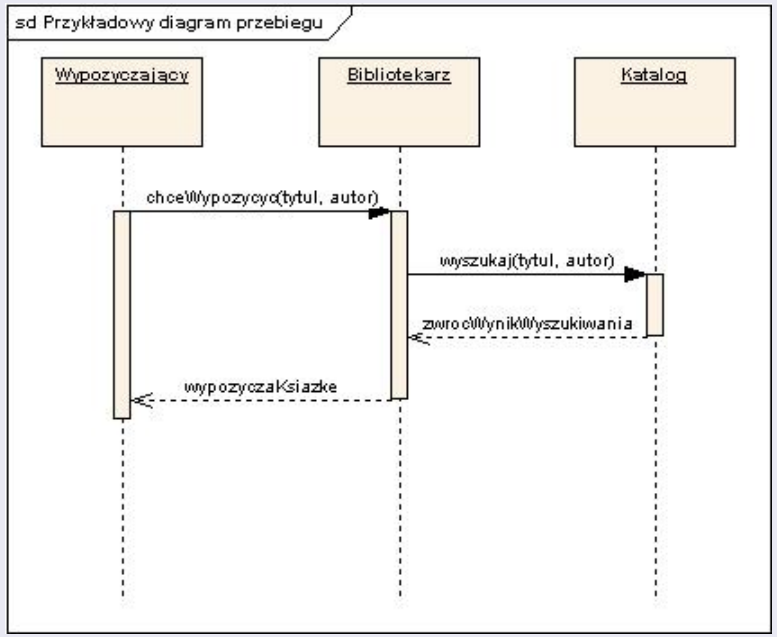
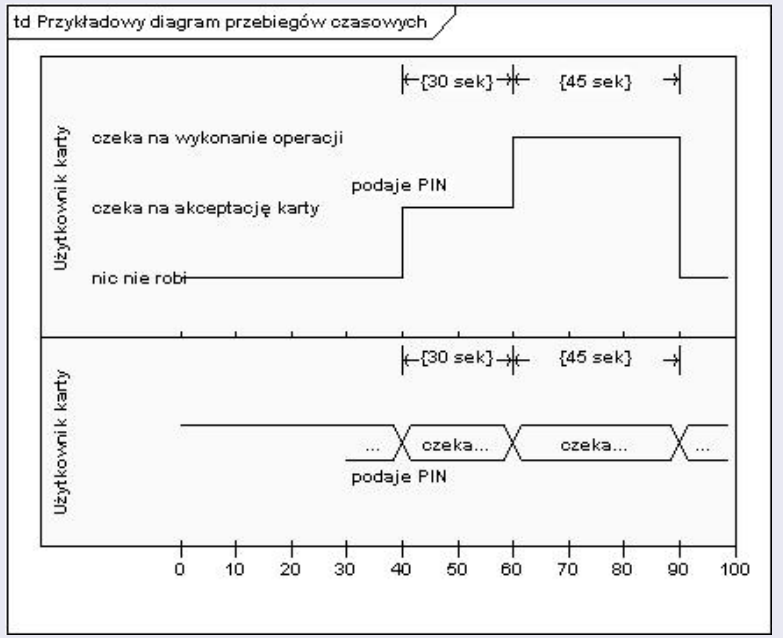


Diagram przebiegów czasowych

Diagram przebiegów czasowych obrazuje zachowanie obiektu z naciskiem na dokładne określenie czasu, w którym obiekt jest poddawany jakimś zmianom lub sam wykonuje jakieś działanie.



Podsumowanie

- Diagramy komponentów i wdrożenia przedstawiają logiczną i fizyczną strukturę podsystemów.
- Diagramy interakcji służą do opisu komunikacji pomiędzy obiektami
- Diagramy czynności definiują algorytmy realizacji funkcji, a diagramy stanu – zmianę zachowania obiektów.

Bardziej szczegółowe informacje na kursie *Technologia Programowania* (III semestr).

Podsumowanie

- Diagramy komponentów i wdrożenia przedstawiają logiczną i fizyczną strukturę podsystemów.
- Diagramy interakcji służą do opisu komunikacji pomiędzy obiektami
- Diagramy czynności definiują algorytmy realizacji funkcji, a diagramy stanu – zmianę zachowania obiektów.

Bardziej szczegółowe informacje na kursie *Technologia Programowania* (III semestr).

Java vs cpp - podobieństwa

- Podobny sposób definiowania klas.
- Występowanie typów podstawowych: **boolean**, **char**, **byte** , **short**, **int**, **long**, **float**, **double**.
- Podobna zasada definiowania konstruktorów.
- Podobna zasada przeładowywania funkcji.
- Podobna zasada definiowania dziedziczenia. Drobne różnice w składni.

Java vs cpp - różnice

Java

- Wszystko musi być zdefiniowane w klasie. Nie ma funkcji i atrybutów globalnych. Można wykorzystać słowo **static**. Nie ma struktur oraz unii.
- Obiekty typów innych niż podstawowe muszą być definiowane za pomocą **new**. Nie można definiować na stosie tak jak w c++. Wszystkie typy podstawowe mogą być definiowane tylko na stosie (bez użycia **new**). Jednak każdy typ podstawowy posiada swoją klasę opakowującą.
- Używanie pakietów (**packages**) zamiast przestrzeni nazw (**namespaces**)
- Referencje do obiektów są inicjalizowane wartością pustą. Typy podstawowe wartościami 0 lub równoważnymi.
- Nie ma wskaźników takich jak w C.
- Brak destruktorów. Używanie mechanizmu **Garbage Collector**. Można jednak nadpisać metodę **finalize**.
- Brak domyślnych argumentów metod.

Java vs cpp - różnice

Java

- Brak instrukcji **goto**. Mozna uzywać **break** i **continue**.
- Wszystkie obiekty dziedziczą po klasie **Object**.
- Brak wielodziedziczenia. Klasa może dziedziczyć po jednej klasie, ale po wielu interfejsach.
- Interfejsy.
- Brak słowa kluczowego **virtual**. Wszystkie niestatyczne metody używają łączenia dynamicznego.
- Brak przeładowania operatorów, ale możliwe jest przeładowanie metod.

CSharp vs Java - podobieństwa

- Obiekty są referencjami. Tworzenie za pomocą słowa kluczowego **new**.
- **Garbage Collector**.
- Języki czysto obiektowe. Każda klasa dziedziczy po **Object**.
- Mozliwość dziedziczenia po tylko po jednej klasie, ale po wielu interfejsach.
- Wątki i synchronizacja.
- Obsługa wyjątków.

CSharp vs Java

Base <--> super

```
1  /* CSharp */
2  public MyClass(string s) : base(s)
3  {
4  }
5  public MyClass() : base()
6  {
7  }
```

```
1  /* Java */
2  Public MyClass(String s)
3  {
4  super(s);
5  }
6  public MyClass()
7  {
8  super();
9  }}
```

CSharp vs Java

Is <--> instanceof

```
1  /* CSharp */
2  MyClass myClass = new MyClass();
3  if (myClass is MyClass)
4  {
5  //executed
6  }
```

```
1  /* Java */
2  MyClass myClass = new MyClass();
3  if (myClass instanceof MyClass)
4  {
5  //executed
6  }
```

CSharp vs Java

lock <--> synchronized

```
1  /* CSharp */
2  MyClass myClass = new MyClass();
3  lock (myClass)
4  {
5  //myClass is
6  //locked
7  }
8  //myClass is
9  //unlocked
```

```
1  /* Java */
2  MyClass myClass = new MyClass();
3  synchronized (myClass)
4  {
5  //myClass is
6  //locked
7  }
8  //myClass is
9  //unlocked
```

CSharp vs Java

namespace <--> package

```
1  /* CSharp */
2  namespace MySpace
3  {
4  }
```

```
1  /* Java */
2  //package must be first keyword in class file
3  package MySpace;
4  public class MyClass
5  {
6  }
```

CSharp vs Java

readonly <--> const

```
1  /* CSharp */
2  //legal initialization
3  readonly int constInt = 5;
4  //illegal attempt to
5  //side-effect variable
6  constInt = 6;
```

```
1  /* Java */
2  //legal initialization
3  final int constInt = 5;
4  //illegal attempt to
5  //side-effect variable
6  constInt = 6;
```

CSharp vs Java

sealed <--> final

```
1  /* CSharp */
2  //legal definition
3  public sealed class A
4  {
5  }
6  //illegal attempt to subclass - A is sealed
7  public class B: A
8  {
9  }
```

```
1  /* Java */
2  //legal definition
3  public final class A
4  {
5  }
6  //illegal attempt to subclass - A is sealed
7  public class B extends A
8  {
9  }
```

CSharp vs Java

using <--> import

```
1 /* CSharp */
2 using System;
```

```
1 /* Java */
2 import System;
```

CSharp vs Java

internal <--> private

```
1  /* CSharp */
2  namespace Hidden
3  {
4  internal class A
5  {
6  }
7  }
8  //another library
9  using Hidden;
10 //attempt to illegally use a Hidden class
11 A a = new A();
```

```
1  /* Java */
2  package Hidden;
3  private class A
4  {
5  }
6  //another library
7  import Hidden;
8  //attempt to illegally use a Hidden class
9  A a = new A();
```

CSharp vs Java

extends

```
1  /* CSharp */
2  //A is a subclass of
3  //B
4  public class A : B
5  {
6  }
```

```
1  /* Java */
2  //A is a subclass of
3  //B
4  public class A extends B
5  {
6  }
```

CSharp vs Java

implements

```
1  /* CSharp */
2  //A implements I
3  public class A : I
4  {
5  }
```

```
1  /* Java */
2  //A implements I
3  public class A implements I
4  {
5  }
```

Tylko w CSharp

as

```
1  /* CSharp */
2  Object o = new string();
3  string s = o as string;
4  if (null != s)
5  {
6  //executed
7  Console.writeln(s);
8  }
```

```
1  /* Java */
2  Object o = new String();
3  string s = null;
4  if (o instanceof String)
5  {
6  s = (String) o;
7  }
8  if (null != s)
9  {
10 //executed
11 System.Out.Writeln(s);
12 }
```

Tylko w CSharp

get

```
1  /* CSharp */
2  class MyClass
3  {
4  private int m_int;
5  public int MyInt
6  {
7  get
8  {return m_int;}
9  }
10 MyClass m = new MyClass();
11 int n = m.MyInt;
```

```
1  /* Java */
2  class MyClass
3  {
4  private int m_int;
5  public int getInt()
6  {return (m_int);}
7  }
8  MyClass m = new MyClass();
9  int n = m.getInt();
```

Tylko w CSharp

set

```
1  /* CSharp */
2  public class MyClass
3  {
4  private int m_int;
5  public int MyInt
6  {
7  set
8  {m_int = value;}
9  }
10 }
11 MyClass m = new MyClass();
12 m.MyInt = 3;
```

```
1  /* Java */
2  public class MyClass
3  {
4  private int m_int;
5  public void set(int i)
6  {m_int = i;}
7  }
8  MyClass m = new MyClass();
9  m.set(3);
```

Tylko w CSharp

operator

```
1  /* CSharp */
2  public class Vector3D
3  {
4  public static Vector3D operator + (Vector3D v)
5  {
6  return (new Vector3D(x+v.x,y+v.y,z+v.z));
7  }
8  }
```

```
1  /* Java */
2  public class Vector3D
3  {
4  public Vector3D add(Vector3d two)
5  {
6  //add implementation
7  }
8  }
```

Tylko w CSharp

override

```
1  /* CSharp */
2  public class A
3  {
4  public virtual int Test()
5  {
6  return 0;
7  }
8  }
9  public class B : A
10 {
11 public override int Test()
12 {
13 return 1;
14 }
15 }
16 A a = new B();
17 int I = a.Test(); //1 is returned
```

Tylko w CSharp

override

```
1  /* Java */
2  public class A
3  {
4  public int Test()
5  {
6  return 0;
7  }
8  }
9  public class B extends A
10 {
11 public int Test()
12 {
13 return 1;
14 }
15 }
16 A a = new B();
17 int I = a.Test();
18 //1 is returned. All methods
19 //in Java are virtual
```