# Paradygmaty programowania obiektowego

## Mechanizm refleksji w programowaniu obiektowym

dr inż. Radosław Idzikowski

# Czym jest refleksja?

**Definicja**

Refleksja w programowaniu obiektowym to zdolność programu do dynamicznego analizowania i manipulowania swoimi własnymi strukturami danych.

- Program może uzyskać informacje o klasach, obiektach, funkcjach, zmiennych itp.
- Pozwala na dynamiczne modyfikowanie zachowania programu.

# Zastosowanie refleksji w C++

- Inspekcja typów w czasie wykonania.
- Tworzenie uniwersalnych narzędzi i bibliotek.
- Rozszerzanie zachowania klas w czasie wykonania.
- Implementacja mechanizmów serializacji i deserializacji.
- Obsługa dynamicznego wywoływania metod.

# Ograniczenia w C++

- C++ nie dostarcza wbudowanych mechanizmów refleksji jak niektóre inne języki.
- Brak automatycznego dostępu do informacji o typach w czasie wykonania.
- Konieczność stosowania manualnych technik lub zewnętrznych bibliotek.
- Może wpływać na wydajność programu ze względu na narzut czasowy.

# Przykłady języków z wbudowanym mechanizmem refleksji

Refleksja to ważna cecha niektórych języków programowania, która umożliwia programom analizowanie i manipulowanie swoimi własnymi strukturami danych w czasie wykonania. Oto kilka przykładów języków, które posiadają wbudowany mechanizm refleksji:

- `Java` posiada rozbudowany mechanizm refleksji. `Java Reflection API` umożliwia programom dynamiczne analizowanie i manipulowanie typami, atrybutami, metodami, konstruktorami itp. w czasie wykonania. Dzięki temu można tworzyć uniwersalne narzędzia, testować i modyfikować obiekty w trakcie działania programu.
- `Python` jest językiem, który jest znany z bogatego mechanizmu refleksji. `Python Reflection API` umożliwia programom uzyskiwanie informacji o typach, atrybutach, funkcjach itp. w czasie wykonania. Można dynamicznie analizować i manipulować obiektami oraz tworzyć metaklasy, które kontrolują tworzenie klas.
- `C#` również dostarcza mechanizmy refleksji. `C# Reflection API` umożliwia programom analizowanie typów, atrybutów, metod, właściwości itp. w czasie wykonania. Refleksja w C# jest użyteczna do tworzenia narzędzi deweloperskich, takich jak refaktoryzacja kodu, generowanie dokumentacji lub testowanie jednostkowe.
- `Ruby` to język, który ma silne wsparcie dla refleksji. `Ruby Reflection API` umożliwia programom analizowanie i manipulowanie klas, modułów, metod, zmiennych itp. w czasie wykonania. Refleksja w Ruby jest szeroko wykorzystywana do tworzenia metaprogramowania i budowania dynamicznych aplikacji.
- `Groovy` jako język dynamiczny, posiada wbudowany mechanizm refleksji. `Groovy Reflection API` umożliwia programom dynamiczne analizowanie i manipulowanie typami, atrybutami, metodami, konstruktorami itp. Refleksja w Groovy jest użyteczna w przypadku tworzenia skryptów, testowania jednostkowego i tworzenia narzędzi programistycznych.

# Ograniczenia refleksji w C++

Chociaż C++ jest językiem programowania obiektowego, nie zawiera on natywnie pełnego mechanizmu refleksji, jak wiele innych języków. Istnieje kilka ograniczeń dotyczących refleksji w C++, oto niektóre z nich:

- **Brak domyślnego mechanizmu refleksji**: C++ nie dostarcza wbudowanego API refleksji takiego jak Java Reflection API czy Python Reflection API. Programista musi samodzielnie implementować mechanizmy refleksji, jeśli są one wymagane.
- **Brak pełnej informacji o typach w czasie wykonania**: C++ jest statycznie typowanym językiem, co oznacza, że większość informacji o typach jest znana w czasie kompilacji. W rezultacie, w czasie wykonania jest ograniczona ilość dostępnych informacji o typach, co utrudnia pełne wykorzystanie refleksji.
- **Brak możliwości modyfikacji struktury klas w czasie wykonania**: W odróżnieniu od niektórych języków, w których refleksja umożliwia dodawanie, usuwanie lub modyfikację klas w czasie wykonania, C++ nie zapewnia takiej elastyczności. Struktura klas musi być określona i skompilowana przed uruchomieniem programu.
- **Ograniczenia w dostępie do prywatnych elementów klas**: Refleksja w C++ ma ograniczony dostęp do prywatnych elementów klas, takich jak prywatne pola czy prywatne metody. W przypadku korzystania z refleksji, dostęp do takich prywatnych elementów może być utrudniony lub niemożliwy.
- **Wydajność**: Implementacja mechanizmów refleksji w C++ może wpływać na wydajność programu. Refleksja często wiąże się z większym narzutem czasowym i pamięciowym, ponieważ wymaga dynamicznej analizy struktury klas w czasie wykonania.

# Techniki manualnego uzyskiwania informacji o typach w C++

Ponieważ język C++ nie posiada wbudowanego mechanizmu refleksji, programiści często muszą stosować techniki manualnego uzyskiwania informacji o typach. Oto kilka takich technik:

- **Szablony typów**: C++ umożliwia programistom definiowanie i manipulowanie szablonami typów. Wykorzystanie szablonów może prowadzić do statycznego rozwiązania niektórych problemów refleksji, takich jak uzyskanie informacji o typach lub dostęp do składowych klasy.
- **Makra preprocesora**: Preprocesor C++ umożliwia programistom manipulację kodem źródłowym przed kompilacją. Makra preprocesora mogą być wykorzystane do generowania kodu na podstawie informacji o typach w czasie kompilacji.
- **Zastosowanie wzorców projektowych**: Niektóre wzorce projektowe, takie jak Wzorzec Fabryki lub Wzorzec Metoda Fabrykująca, mogą być wykorzystane do dynamicznego tworzenia obiektów na podstawie informacji o typach.
- **Ręczna implementacja refleksji**: Programiści mogą samodzielnie zaimplementować prosty mechanizm refleksji w C++. Można to zrobić poprzez utworzenie struktury danych lub klas, które przechowują informacje o typach i składowych klas.
- **Zewnętrzne narzędzia**: Istnieją zewnętrzne narzędzia, takie jak biblioteki lub frameworki, które dostarczają mechanizmy refleksji dla języka C++. Można je wykorzystać, aby uzyskać informacje o typach w czasie wykonania i manipulować nimi.

# Szablony typów

Szablon typów (ang. *type template*), znany również jako szablon generyczny, to mechanizm w języku C++, który umożliwia programistom tworzenie generycznych klas lub funkcji, które mogą działać na różnych typach danych. Szablony typów pozwalają na tworzenie kodu, który jest parametryzowany typem, co umożliwia reużywalność kodu i elastyczność w obsłudze różnych typów danych.

```
#include <iostream>
template <typename T>
T minimum(const T& a, const T& b)
{
    return a < b ? a : b;
}
int main() {
    int int_a = 5;
    int int_b = 7;
    auto min1 = minimum<int>(int_a, int_b);
    std::cout << min1 << "\n";
    float float_a = 3.5f;
    float float_b = 7.14f;
    auto min2 = minimum<float>(float_a, float_b);
    std::cout << min2 << "\n";
}
```

# Szablony typów

Zastosowanie szablonów typów do uzyskania informacji o typie:

```
#include <iostream>
class Dog {
public:
        int age;
        std::string name;
};
template <typename T>
void PrintType() {
        std::cout << typeid(T).name() << std::endl;
}
int main() {
        PrintType<int>();
        PrintType<double>();
        PrintType<Dog>();
        return 0;
}
```

# Makra preprocesora

```
#include <iostream>
#define PRINT_TYPE(T) \
  std::cout << typeid(T).name() << std::endl;
class Dog {
public:
        int age;
};
int main() {
        PRINT_TYPE(int);
        PRINT_TYPE(double);
        PRINT_TYPE(Dog);
        Dog dog;
        PRINT_TYPE(dog.age);
        return 0;
}
```

# Wzorzec projektowy Fabryka

## Definicja

**Wzorzec projektowy Fabryka (*Factory*)**

Kkreacyjny wzorzec projektowy, który zapewnia jednolity sposób tworzenia obiektów, ukrywając szczegóły implementacyjne. Fabryka jest odpowiedzialna za tworzenie obiektów i zwracanie ich jako interfejsu ogólnego typu.

Główne cele wzorca Fabryka to:

- Abstrakcja procesu tworzenia obiektów: Wzorzec Fabryka umożliwia odseparowanie kodu klienta od szczegółów tworzenia obiektów, dzięki czemu klient nie musi znać konkretnej implementacji i może operować na ogólnym interfejsie.
- Zapewnienie jednolitego interfejsu: Fabryka tworzy obiekty i zwraca je jako interfejsu ogólnego typu. Dzięki temu kod klienta może operować na obiektach bez konieczności zależności od konkretnych implementacji.
- Łatwa rozszerzalność: Wzorzec Fabryka umożliwia łatwe dodawanie nowych typów obiektów bez zmiany istniejącego kodu klienta. Wystarczy dodać nową klasę dziedziczącą po interfejsie ogólnego typu i zaimplementować ją w fabryce.

# Wzorzec projektowy Fabryka

## Elementy

Wzorzec Fabryka składa się z trzech głównych elementów:

- Interfejsu Fabryki (*Factory Interface*): definiuje ogólny interfejs dla fabryki, który jest używany przez klienta do tworzenia obiektów.
- Fabryki (*Concrete Factory*): implementuje interfejs fabryki i jest odpowiedzialna za tworzenie konkretnych obiektów.
- Obiektów (*Concrete Object*): są to obiekty tworzone przez fabrykę. Implementują interfejs ogólny typu i dostarczają konkretnej funkcjonalności.

# Wzorzec projektowy Fabryka

*Factory Interface*

```
class Animal {
public:
    std::string name;
    virtual void Voice() const = 0;
};
```

# Wzorzec projektowy Fabryka

*Concrete Object*

```
class Dog : public Animal {
public:
    void Voice() const override  {
        std::cout << "Woof!!!" << std::endl;
    }
};

class Cat : public Animal {
public:
    void Voice() const override {
        std::cout << "Meow!!!" << std::endl;
    }
};
```

# Wzorzec projektowy Fabryka

*Concrete Factory*

```
class Factory {
public:
    static Animal* CreateObject(int type) {
        switch (type) {
        case 1: return new Dog();
        case 2: return new Cat();
        default: return nullptr;
        }
    }
};
```

# Wzorzec projektowy Fabryka

## Wywołanie

```
int main() {
    Animal* obj1 = Factory::CreateObject(1);
    if (obj1) {
        obj1->Voice();
    }

    Animal* obj2 = Factory::CreateObject(2);
    if (obj2) {
        obj2->Voice();
    }

    return 0;
}
```

# Ręczna implementacja refleksji w C++

Można stworzyć tablicę struktur, która przechowuje informacje o klasach i ich składowych. Struktury te zawierają pola takie jak nazwa, typ, modyfikatory dostępu itp.

```
struct FieldInfo {
    std::string name;
    std::string type;
};

struct ClassInfo {
    std::string name;
    std::vector<FieldInfo> fields;
};

std::vector<ClassInfo> classMetadata;
```

# Ręczna implementacja refleksji w C++

Używamy składni typedef do zdefiniowania nowego typu o nazwie `CreateObjectFunc`. Typ ten jest wskaźnikiem do funkcji, która nie przyjmuje żadnych argumentów i zwraca wskaźnik do obiektu klasy bazowej `Animal`.

```
typedef Animal* (*CreateObjectFunc)();
std::map<std::string, CreateObjectFunc> objectRegistry;
```

Funkcja szablonowa `createObject` tworzy obiekt typu `T` i zwraca wskaźnik do obiektu klasy bazowej.

```
template<typename T>
Animal* createObject() {
    return new T();
}
```

Funkcja `registerObject` dodaje wpis do mapy `objectRegistry` z nazwą klasy i funkcją tworzącą obiekt.

```
void registerObject(const std::string& className, CreateObjectFunc createFunc) {
    objectRegistry[className] = createFunc;
}
```

Funkcja `createInstance` na podstawie podanej nazwy wyszukuje funkcję w mapie i wywołuje ją.

```
Animal* createInstance(const std::string& className) {
    if (objectRegistry.find(className) != objectRegistry.end()) {
        CreateObjectFunc createFunc = objectRegistry[className];
        return createFunc();
    }
```

# Ręczna implementacja refleksji w C++

W funkcji `main` demonstrujemy tworzenie obiektów klas `Dog` i `Cat` poprzez wywołanie funkcji `createInstance` z odpowiednimi nazwami klas. Następnie wywołujemy funkcję *Voice* dla każdego utworzonego obiektu.

```
int main() {
    registerObject("Dog", &createObject<Dog>);
    registerObject("Cat", &createObject<Cat>);
    Animal* obj1 = createInstance("Dog");
    if (obj1 != nullptr) {
        obj1->Voice();
    }
    Animal* obj2 = createInstance("Cat");
    if (obj2 != nullptr) {
        obj2->Voice();
    }
    delete obj1;
    delete obj2;
    return 0;
}
```

Przykład dobrze ilustruje, jak można dynamicznie tworzyć obiekty różnych klas na podstawie nazwy klasy, wykorzystując ręczną implementację mechanizmu refleksji w C++.

# Biblioteka Boost.Reflection

Rozszerzenie biblioteki Boost, które dostarcza mechanizmy refleksji w języku C++. Oferuje ona zestaw narzędzi i klas, które umożliwiają dynamiczną analizę i manipulację strukturą klas i obiektów w czasie wykonywania programu. Oto kilka kluczowych elementów i funkcji oferowanych przez bibliotekę Boost.Reflection.

- Klasa `boost::reflection::object`: Jest to główna klasa biblioteki Boost.Reflection, reprezentująca obiekty, które mogą być analizowane i manipulowane w czasie wykonywania programu. Przez obiekt można przechodzić, odczytywać i zmieniać wartości pól oraz wywoływać metody.
- Klasa `boost::reflection::class_`: Reprezentuje klasę w refleksji. Przez obiekt tej klasy można uzyskać informacje o polach, metodach, konstruktorach i innych elementach klasy. Można również tworzyć nowe obiekty klasy i wywoływać metody na nich.
- Klasa `boost::reflection::property`: Reprezentuje pole w refleksji. Przez obiekt tej klasy można odczytywać i zmieniać wartość pola obiektu.
- Klasa `boost::reflection::method`: Reprezentuje metodę w refleksji. Przez obiekt tej klasy można wywoływać metody na obiekcie.
- Klasa `boost::reflection::constructor`: Reprezentuje konstruktor w refleksji. Przez obiekt tej klasy można tworzyć nowe obiekty klasy.

# Biblioteka `Boost.Reflection`

Biblioteka `Boost.Reflection` oferuje również wiele innych funkcji, takich jak:

- Tworzenie dynamicznych obiektów na podstawie nazwy klasy.
- Pobieranie listy pól, metod i konstruktorów klasy.
- Wywoływanie metod na obiektach dynamicznie.
- Analizowanie informacji o typach, takich jak nazwa, rozmiar, modyfikatory itp.
- Dostęp do dziedziczenia i hierarchii klas.
- Rejestracja własnych klas w refleksji.

Dzięki bibliotece `Boost.Reflection` można w prosty sposób uzyskać dostęp do informacji o klasach i obiektach, a także manipulować nimi dynamicznie. Umożliwia to tworzenie bardziej elastycznych i konfigurowalnych aplikacji, które mogą dostosowywać się do zmieniających się wymagań w czasie wykonywania programu.

# Biblioteka Boost.Reflection

Tworzenie obiektu na podstawie nazwy klasy

```
#include <boost/reflection.hpp>

using namespace boost::reflection;
int main() {
  object obj = create_object("Dog");
  return 0;
}
```

# Biblioteka Boost.Reflection

Odczytywanie wartości pola obiektu

```
#include <boost/reflection.hpp>

using namespace boost::reflection;
int main() {
  object obj = create_object("Dog");

  property prop = obj.get_property("Age");
  int value = prop.get<int>();
  return 0;
}
```

# Biblioteka Boost.Reflection

Wywoływanie metody na obiekcie

```
#include <boost/reflection.hpp>

using namespace boost::reflection;
int main() {
  object obj = create_object("Dog");
  method func = obj.get_method("bark");
  func.invoke();
  return 0;
}
```

# Wpływu refleksji na wydajność programu

Refleksja w programowaniu obiektowym ma pewien wpływ na wydajność programu. Oto kilka aspektów, które należy wziąć pod uwagę:

- Narzut czasowy: Mechanizmy refleksji, takie jak dynamiczne wywoływanie metod i dostęp do informacji o typach w czasie wykonania, mogą wprowadzać pewien narzut czasowy. Wywołanie metody dynamicznie zamiast statycznie może być wolniejsze, ponieważ wymaga dodatkowych operacji rozpoznania i interpretacji typów w czasie wykonania.

- Narzut pamięciowy: Mechanizmy refleksji mogą również wymagać dodatkowej pamięci do przechowywania informacji o typach i strukturach danych w czasie wykonania. To może prowadzić do większego zużycia pamięci przez program.

- Złożoność kodu: Używanie refleksji może zwiększyć złożoność kodu i utrudnić jego zrozumienie i utrzymanie. Mechanizmy refleksji wprowadzają dodatkowe abstrakcje i skomplikowane operacje, które mogą utrudniać debugowanie i rozwijanie kodu.

- Optymalizacja: Mechanizmy refleksji mogą ograniczać możliwość optymalizacji kodu przez kompilator, ponieważ niektóre operacje refleksji są dynamiczne i nieznane statycznie, kompilator może mieć trudności z dokładnym określeniem i zoptymalizowaniem tych fragmentów kodu.

Warto jednak zauważyć, że wpływ refleksji na wydajność może być zróżnicowany w zależności od konkretnych implementacji i narzędzi używanych w danym języku programowania. Niektóre języki programowania i biblioteki, takie jak C++ z biblioteką Boost.Reflection, mogą oferować bardziej zoptymalizowane mechanizmy refleksji, które minimalizują wpływ na wydajność.

# Zastosowanie

- Tworzenie wtyczek i rozszerzeń: Refleksja umożliwia dynamiczne ładowanie modułów i wtyczek do aplikacji w czasie wykonania. Można użyć refleksji do dynamicznego odnajdywania i inicjalizacji tych modułów, co daje dużą elastyczność w rozszerzaniu funkcjonalności aplikacji.

- Serializacja danych: Refleksja może być wykorzystana do automatycznego serializowania i deserializowania obiektów. Na podstawie informacji o typach, można odczytać i zapisywać dane w odpowiednich formatach, takich jak `XML`, `JSON`, lub binarne.

- Automatyczne mapowanie obiektowo-relacyjne (ORM): Refleksja może ułatwić mapowanie obiektów na relacyjne bazy danych i odwrotnie. Przy użyciu informacji o typach można dynamicznie generować zapytania `SQL` i mapować wyniki na obiekty.

- Testowanie i debugowanie: Refleksja może być przydatna podczas testowania i debugowania aplikacji. Umożliwia dynamiczne wywoływanie metod i manipulację obiektami w czasie wykonania, co ułatwia tworzenie skomplikowanych scenariuszy testowych i analizowanie działania aplikacji.

- Generowanie kodu: Refleksja może być używana do generowania kodu źródłowego na podstawie informacji o typach. Można automatycznie tworzyć interfejsy, struktury danych, adaptery i wiele innych, oszczędzając czas i zapewniając spójność kodu.

- Introspekcja aplikacji: Refleksja może być stosowana do analizy struktury aplikacji i odkrywania dostępnych klas, metod, pól itp. Może to być przydatne w przypadku generowania dokumentacji, tworzenia narzędzi do analizy statycznej lub budowania systemów interaktywnych, które dynamicznie odkrywają i manipulują

# Niejawna serializacja w C#

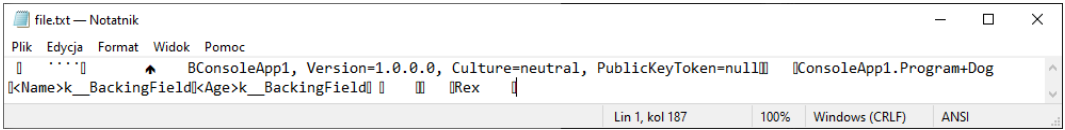
Przekształcenie obiegu do formy umożliwiającym jego zapisanie. Niejawną (*explicit*) seralizację – automatyczną uzyskuję się przez dołączenie atrybutu jako meta-dane do definicji klasy.

```
[Serializable]
class Animal
{
    string Name;
}
```

W przypadku jawniej (*implicit*) serializacji należy napisać kod do jej obsługi.

# Niejawna serializacja w C#

```
[Serializable]
public class Dog
{
    public string Name { get; set; }
    public int Age { get; set; }
    public override string ToString() { return Name;}
}
static void Main(string[] args)
{
    Dog dog = new Dog() {Name = "Rex", Age = 2};
    Stream writer = new FileStream("file.txt", FileMode.Create);
    BinaryFormatter b = new BinaryFormatter();
    b.Serialize(writer, dog);
    writer.Close();
    Console.ReadLine();
}
```



# Niejawna serializacja do XML w C#

```
static void Main(string[] args)
{
    Dog dog = new Dog() {Name = "Rex", Age = 2};
    XmlSerializer x = new XmlSerializer(dog.GetType());
    x.Serialize(Console.Out, dog);
    Console.ReadLine();
}
```

```
<?xml version="1.0" encoding="ibm852"?>
<Dog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchem
  <Name>Rex</Name>
  <Age>2</Age>
</Dog>
```

# Serializacja z użyciem refleksji w C#

```
public class Dog
{
    public string Name { get; set; }
    public string Breed { get; set; }
    public int Age { get; set; }
    public  string Serialise()
    {
        string str = "";
        Type type = this.GetType();
        PropertyInfo[] properties = type.GetProperties();
        foreach (PropertyInfo property in properties)
            str += String.Format("{0}: {1}\t",
            property.Name, property.GetValue(this));
        return str;
    }
}
static void Main(string[] args)
{
    Dog dog = new Dog() { Name = "Rex",Breed = "Wolf", Age = 2};
    Console.WriteLine(dog.Serialise());
}
```

# Prosta refleksja w C#

```
namespace DogApp
{
    public class Program
    {
        public class Dog
        {
            public string Name { get; set; }
            public int Age { get; set; }
            public void Voice(int repeat)
            {
                for (int i = 0; i < repeat; i++)
                    Console.WriteLine("Woof!");
            }
            public override string ToString() { return Name;}
        }
        static void Main(string[] args)
        {
            var type = Type.GetType("DogApp.Program+Dog");
            Object obj = Activator.CreateInstance(type);
            MethodInfo method = obj.GetType().GetMethod("Voice");
            method.Invoke(obj, new object[] {5});
            Console.ReadLine();
```

# Mechanizm refleksji w C#

## Działanie

Główną klasą dla refleksji jest klasa `System.Type`. Przy użyciu tej klasy można przeszukać oraz przeanalizować odpowiednie metadane w celu podejrzenia:

- pól,
- własności,
- metod,
- wydarzeń.

# Mechanizm refleksji w C#

## Elementy

- `Module` – uzyskanie wszystkich metod zdefiniowanych.
- `MethodInfo` – dostęp do parametrów, nazwy, typu zwracanego, modyfikatorów dostępu i szczegółów implementacji.
- `EventInfo` – dostęp do typu danych obsługi zdarzeń, nazwy, typu deklarowania i atrybutów niestandardowych.
- `ConstructorInfo` – dostęp do danych dotyczących parametrów, modyfikatorów dostępu i szczegółów implementacji konstruktora.
- `Assembly` – załadowanie modułów wymienionych w manifeście zestawu.
- `PropertyInfo` – dostęp do typu deklarującego, typu odzwierciedlonego, typu danych, nazwy i statusu z możliwością zapisu, odczytania lub ustawienia własności.

# Podsumowanie

- Omówiliśmy mechanizm refleksji w programowaniu obiektowym C++
- Przeanalizowaliśmy różne techniki uzyskiwania informacji o typach
- Przedstawiliśmy bibliotekę Boost.Reflection jako zaawansowane narzędzie refleksji
- Zidentyfikowaliśmy ograniczenia refleksji i wpływ na wydajność programu
- Przedstawiliśmy przykłady sytuacji, w których refleksja może być przydatna

# Wnioski

- Refleksja jest potężnym narzędziem w programowaniu obiektowym
- Otwiera wiele możliwości i rozszerza elastyczność aplikacji
- Wymaga uwagi na ograniczenia i wpływ na wydajność
- Wybór odpowiednich narzędzi i technik zależy od wymagań projektu