AGH University of Krakow | Faculty of Mechanical Engineering and Robotics
| Department of Robotics and Mechatronics

MECHATRONIC ENGINEERING | I cycle | semester VI | 2025/2026

Object oriented programming and software engineering

Instruction I:

SOLID

You will learn:

How to design object-oriented programs using the SOLID principles, which help create systems that are easier to maintain, extend, and understand.

You will learn how to structure classes so that they have clear responsibilities, how to design flexible relationships between components, and how to avoid common design problems. You will also practice improving existing code by applying better design decisions.

You will be able to recognize poor design and change it into a more robust solution.

Course supervisor:
dr inż. Lucjan Miękina, miekina@agh.edu.pl

Instruction author:
mgr inż. Joanna Koszyk, jkoszyk@agh.edu.pl

1. Initial information

SOLID is made up of five following principles:

1. **Single Responsibility Principle (SRP):** A class should have only one reason to change, meaning it should only have one responsibility.
2. **Open-Close Principle (OCP):** Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification -> use inheritance and polymorphism to extend behavior without modifying existing code.
3. **Liskov Substitution Principle (LSP):** Subtypes must be substitutable with their base types. Objects of a derived class should be usable in place of objects of the base class without breaking the program.
4. **Interface Segregation Principle (ISP):** Clients should not be forced to depend on interfaces they do not use.
5. **Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules. Both should depend on abstractions (i.e. interfaces).

2. Theoretical content

Bad design usually leads to:

- classes that do too many things
- code that breaks when new features are added
- duplication and hard-to-maintain logic

Good design:

- separates responsibilities
- allows adding new features without rewriting existing code
- makes systems easier to test and extend

Example of Poor Design (Violating SRP):

```
class Sensor {
public:
    void readData(){
        // read measurements from sensor
    }
    void saveToFile(){
        // file handling
    }
    void display(){
        // display data
    }
};
```

Improved design (SRP applied):

```
class Sensor {
public:
    int readData(){
        // read measurements from sensor
        return 0;
    }
};

class FileManager {
public:
    void save(int data){
```

```
            // saving data to file
        }
    };

    class Display {
    public:
        void show(int data){
            // display data
        }
    };
```

Example of OCP:

Instead of modifying code:

```
    if(type == "temperature"){
        // read temperature measurement
    }
    else if(type == "pressure"){
        // read pressure measurement
    }
```

Use polymorphism:

```
    class Sensor{
    public:
        virtual int read() = 0;
    };

    class TemperatureSensor : public Sensor{
    public:
        int read() override{
            return 10;
        }
    };
```

Example of ISP:

```
    class Sensor {
        virtual void read() = 0;
        virtual void setParams() = 0;
    };
```

Not all devices need both -> split interfaces

Example of DIP:

Poor design (Violating DIP):

```
    class TemperatureSensor {
    public:
        int read() {
            return 0;
        }
    };

    class MeasurementSystem {
    private:
        TemperatureSensor sensor;

    public:
        void checkObstacle() {
            int value = sensor.read();
            if (value > 37) {
                // high temperature detected
            }
```

```
    }
};
```

DIP applied:

```
class TemperatureSensor : public Sensor {
public:
    int read() override {
        return 10;
    }
};

class PressureSensor : public Sensor {
public:
    int read() override {
        return 20;
    }
};

class MeasurementSystem {
private:
    Sensor* sensor;

public:
    MeasurementSystem(Sensor* s) : sensor(s){}

    void checkTemperature(){
        int value = sensor->read();
        if (value > 37) {
            // high temperature detected
        }
    }
};
```

Now we can do:

```
Sensor* s1 = new TemperatureSensor();
Sensor* s2 = new PressureSensor();

MeasurementSystem r1(s1);
MeasurementSystem r2(s2);
```

3. Tasks

TASK 1.

You are given a class:

```
class Robot {
public:
    void readDistanceSensor();
    void moveForward();
    void avoidObstacle();
    void readEncoder();
};
```

Change the system along with SRP.

TASK 2.

Add a new type of sensor (for example a camera). Change the system along with OCP (do not modify existing classes)

TASK 3.

Change the program along with DIP. Robot class should depend on abstraction instead of specific implementation.