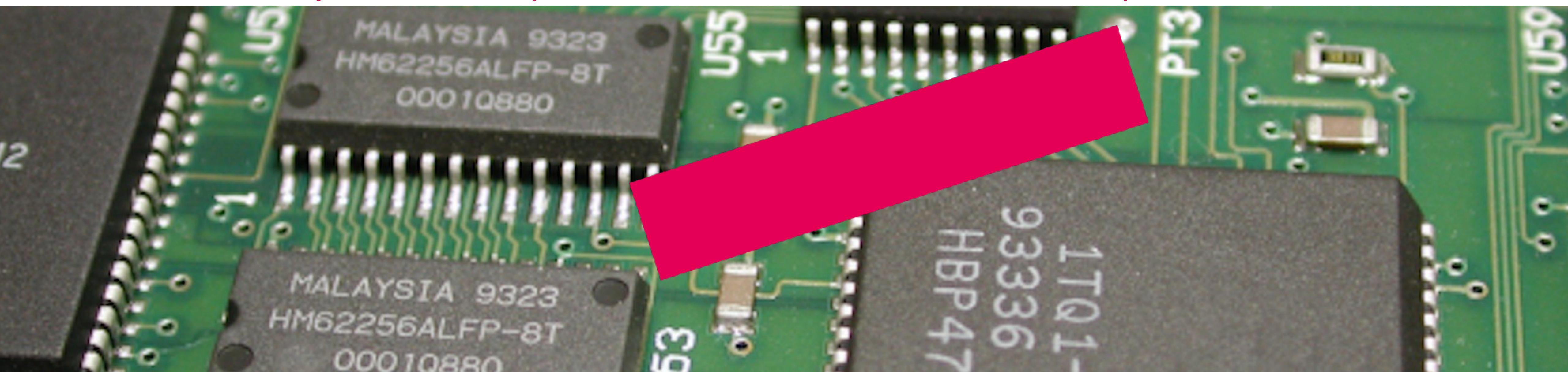


Embedded Systems Development - 2. Constructors, Lambdas and Open-Closed



Electrical Engineering / Embedded Systems
Faculty of Engineering

Johan.Korten@han.nl

V1.3 2024

To begin with...

Attendance

Schedule (exact info see #00 and roster at insite.han.nl)

		C++	UML
Step 1	Single responsibility	Scope, namespaces, string	class diagram
Step 2	Open-Closed Principle	Constructors, iterators, lambdas	Inheritance / Generalization
Step 3	Liskov Substitution Principle	lists, inline functions, default params	
Step 4	Interface Segregation Principle	interfaces and abstract classes	
Step 5	Dependency Inversion Principle	threads, callback	
Step 6	Coupling and cohesion	polymorphism	
Step 7	n/a	n/a	n/a

Note: subject to changes as we go...

Constructors

- Constructors have the same name as the class itself
- Constructors don't have a return type
- A constructor is automatically called when the object is created.
- If no constructor is defined, C++ compiler generates a default constructor:
 - with no parameters
 - with an empty body

C++ Language: Constructors

- default constructor
- copy constructor
- overloading instructors
- destructors

C++ Language: Copy Constructor

In Rectangle.cpp:

```
Rectangle::Rectangle(const Rectangle& rect) { cout << "Copy Constructor Called\n"; }
```

In Rectangle.h:

```
Rectangle(const Rectangle&);
```

```
int main()
{
    Rectangle r1;
    Rectangle r2{1.0, 3.0}; // Uniform initialization: { .... }
    r1 = Rectangle(r2); // calls copy constructor
    return 0;
}
```

C++ Language: Destructor

In Rectangle.cpp:

```
Rectangle::~Rectangle() { cout << "Destructor Called\n"; }
```

In Rectangle.h:

```
~Rectangle(); // destructor
```

// Note: check output of your program to see when the destructor is called ...

C++ Language: Constructors - Default Constructor

Either has:

- no parameters (arguments)

or:

- all parameters have default values

E.g. (in Rectangle.h):

```
Rectangle(double width, double height) : width_{width = 1.0}, height_{height = 1.0} {}
```

```
Rectangle(double width, double height) : width_{width}, height_{height} {}
```

```
Rectangle();
```

In Rectangle.cpp

```
Rectangle::Rectangle()  
    : Rectangle{1.0, 1.0} // Constructor delegation  
{}
```

C++ Language: Iterators

CONTAINER	TYPES OF ITERATOR SUPPORTED
Vector	Random-Access
List	Bidirectional
Deque	Random-Access
Map	Bidirectional
Multimap	Bidirectional
Set	Bidirectional
Multiset	Bidirectional
Stack	No iterator Supported
Queue	No iterator Supported
Priority-Queue	No iterator Supported

C++ Language: Iterators

```
// for_each example
#include <iostream>          // std::cout
#include <algorithm>         // std::for_each
#include <vector>             // std::vector

void myfunction (int i) { // function:
    std::cout << ' ' << i;
}

struct myclass {           // function object type:
    void operator() (int i) {std::cout << ' ' << i;}
} myobject;

int main () {
    std::vector<int> myvector;
    myvector.push_back(10);
    myvector.push_back(20);
    myvector.push_back(30);

    std::cout << "myvector contains:";
    for_each (myvector.begin(), myvector.end(), myfunction);
    std::cout << '\n';

    // or:
    std::cout << "myvector contains:";
    for_each (myvector.begin(), myvector.end(), myobject);
    std::cout << '\n';

    return 0;
}
```

C++ Language: Iterators

<https://www.geeksforgeeks.org/introduction-iterators-c/>

Note: in C++ 20 changes were made in iterator implementation
(Version 20 was released Feb. 2020)

Tip: it might be interesting to also see changes for C++ 2023

<https://en.wikipedia.org/wiki/C%2B%2B23>

C++ Language: Lambda functions

Aka: anonymous function (function literal, lambda abstraction, or lambda expression)

In other languages: closure

[capture clause] (parameters) -> return-type

```
{  
    definition of method  
}
```

C++ Language: Lambda function to variable

```
#include <functional>
#include <iostream>

int main()
{
    // Assign lambda expression that adds two numbers to an auto variable.
    auto f1 = [] (int x, int y) { return x + y; };

    std::cout << f1(2, 3) << std::endl;
}
```

C++ Language: Lambda function to variable

```
#include <functional>
#include <iostream>

int main()
{
    // Assign the same lambda expression to a function object.
    function<int(int, int)> f2 = [](int x, int y) { return x + y; };

    std::cout << f2(3, 4) << std::endl;
}
```

C++ Language: Lambda function to variable

```
#include <functional>
#include <iostream>

int main()
{
    // Assign the same lambda expression to a function object.
    auto sum = [] (int a, int b) {
        return a + b;
    };
    std::cout << sum(2, 4) << std::endl;
}
```

Why is it an anonymous function?

C++ Language: Lambda function to variable

```
#include <functional>
#include <iostream>

int main()
{
    // Assign the same lambda expression to a function object.
    auto sum = [] (int a, int b) {
        return a + b;
    };
    std::cout << sum(2, 4) << std::endl;
}
```

Why is it an anonymous function?

There is no function name in the implementation just the Variable you assign the function to (“auto sum” in this case)...

C++ Language: More on Lambda functions

<https://docs.microsoft.com/en-us/cpp/cpp/examples-of-lambda-expressions?view=vs-2019>

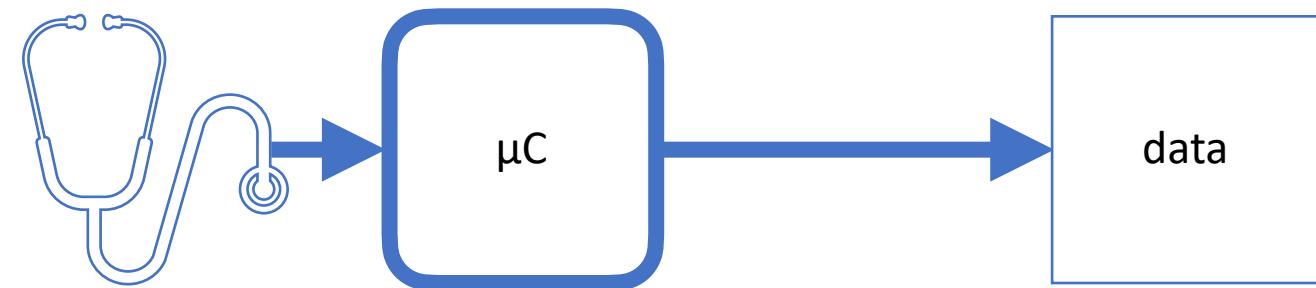
SOLID: Open-Closed Principle

“A software artifact should be open for extension but closed for modification.” - Meyer 1988

(From: Clean Architecture, R.C. Martin, 2018)

SOLID: Open-Closed Principle

Embedded Software Context:



Imagine your application needs to send sensor data to an SD-card...

How hard will it be to react to changing requirements? E.g. UART (maybe Bluetooth) / SPI / i2c...

SOLID: Open-Closed Principle

You'd better: “*Design for change!*”.

Open-Closed Principle: Problem

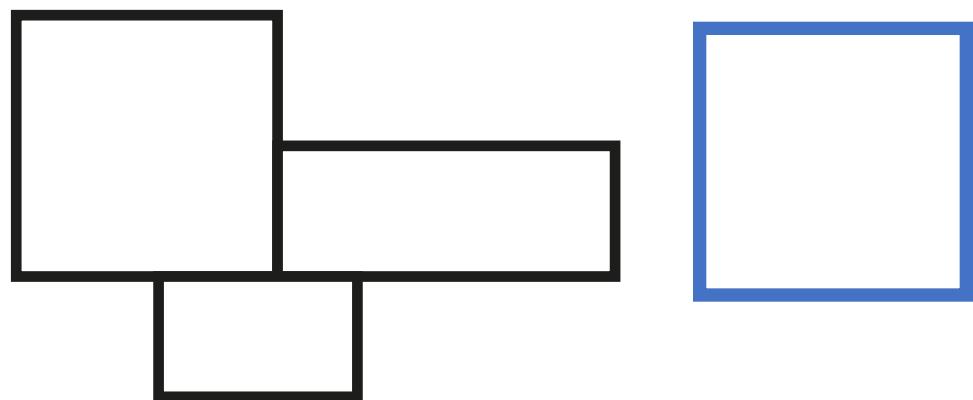
```
// Rectangle.cpp
#include "Rectangle.h"

Rectangle::Rectangle(double width, double height)
{
    this->width = width;
    this->height = height;
}

double Rectangle::getSurfaceArea() const
{
    return width * height;
}

void Rectangle::setWidth(double width)
{
    this->width = width;
}

void Rectangle::setHeight(double height)
{
    this->height = height;
}
```



```
// Rectangle.h
#include <iostream>

class Rectangle
{
public:
    Rectangle(double width, double height);
    double getSurfaceArea() const;
    void setHeight(double height);
    void setWidth(double height);

private:
    double width;
    double height;
};
```

Open-Closed Principle: Problem

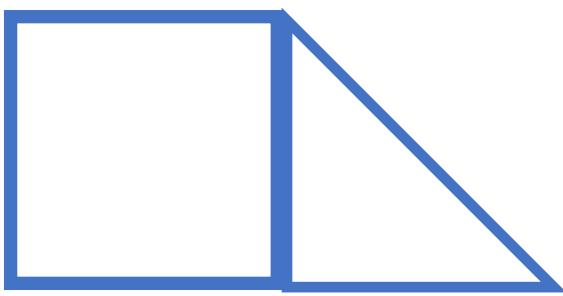
```
// Triangle.cpp
#include "Triangle.h"

Triangle::Triangle(double width, double height)
{
    this->width = width;
    this->height = height;
}

double Triangle::getSurfaceArea() const
{
    return (width * height) / 2;
}

void Triangle::setWidth(double width)
{
    this->width = width;
}

void Triangle::setHeight(double height)
{
    this->height = height;
}
```



```
// Triangle.h
#include <iostream>

class Triangle
{
public:
    Triangle(double width, double height);
    double getSurfaceArea() const;
    void setHeight(double height);
    void setWidth(double height);

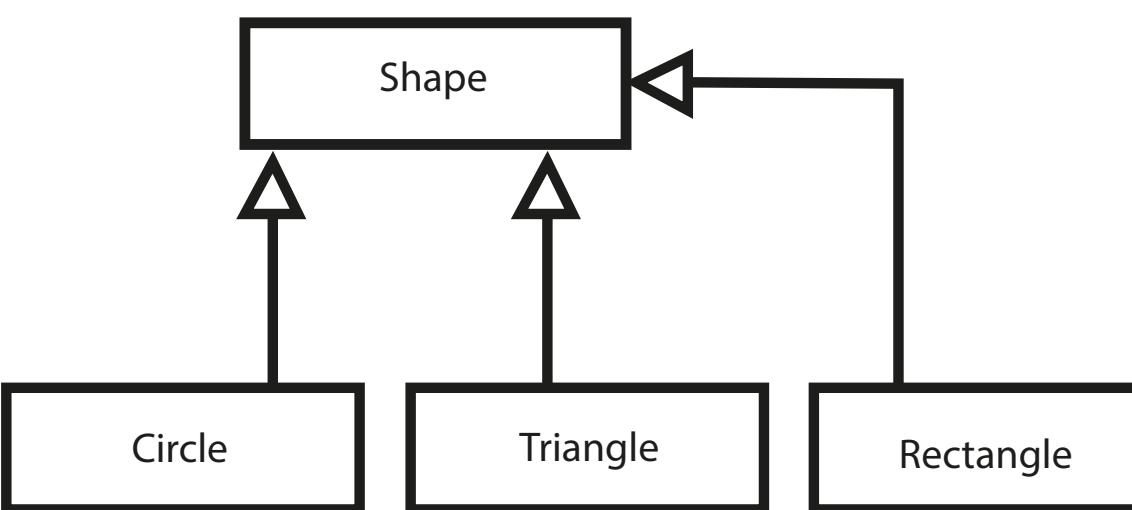
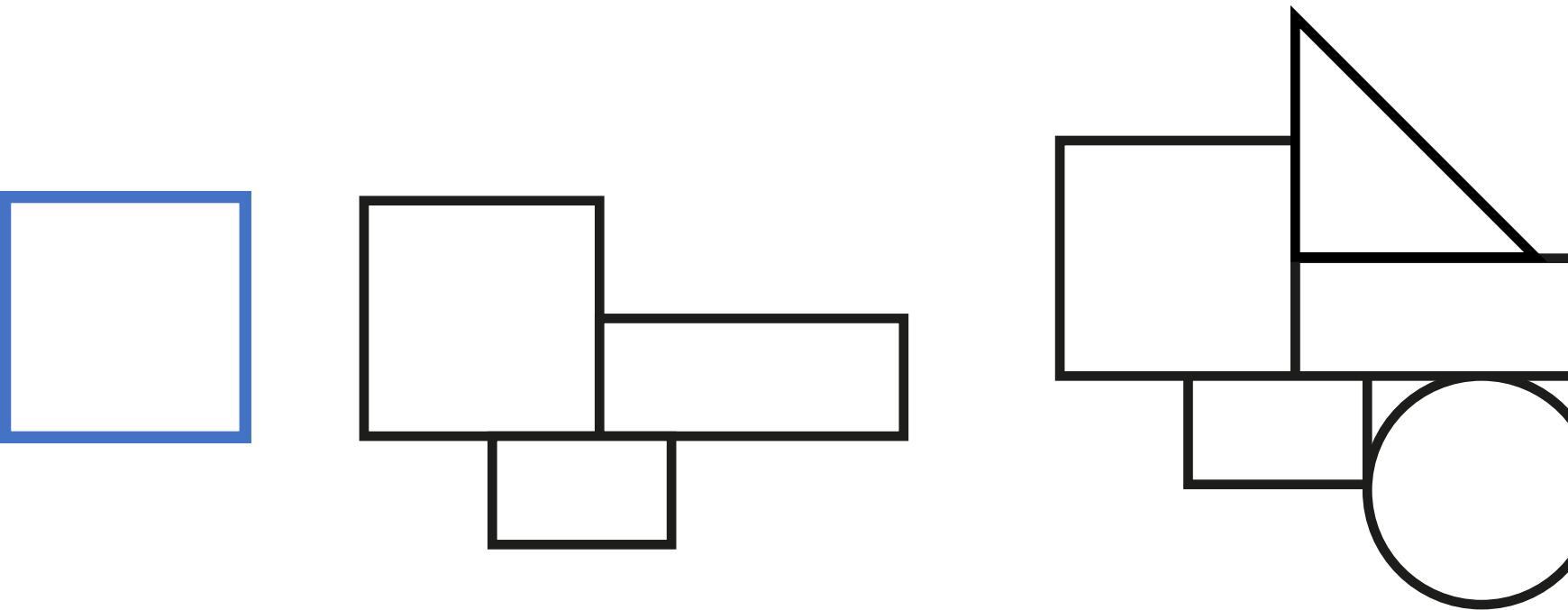
private:
    double width;
    double height;
};
```

Open-Closed Principle: Solution

```
// Base class
class Shape {
public:
    // pure virtual function providing
    // interface framework.
    virtual int getArea() = 0;
    void setWidth(int w) {
        width = w;
    }

    void setHeight(int h) {
        height = h;
    }

protected:
    int width;
    int height;
};
```



```

#include <math.h>

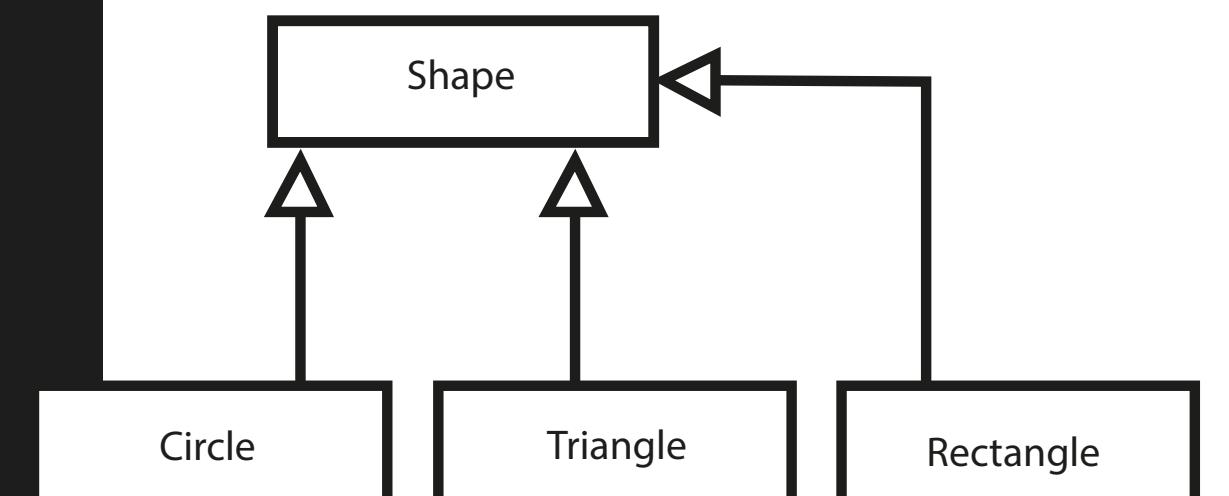
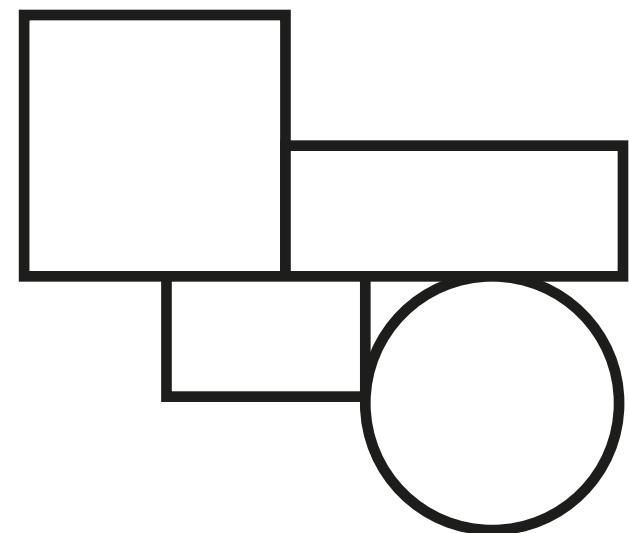
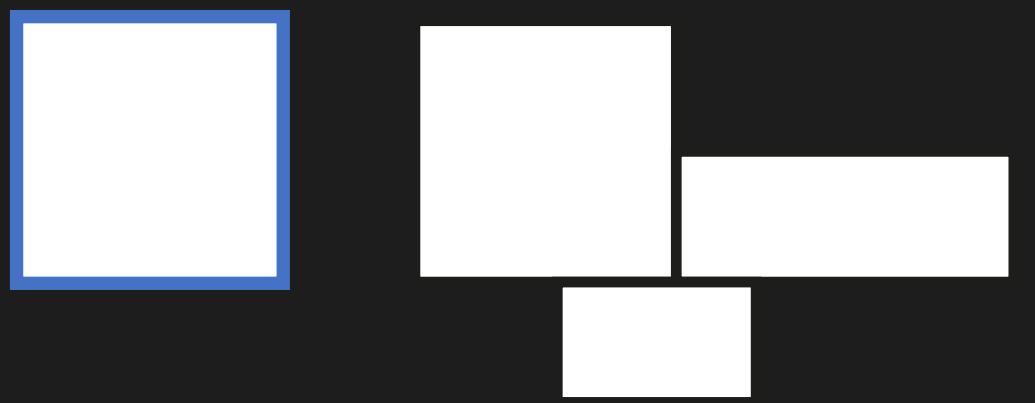
// Derived classes
class Rectangle: public Shape {
public:
    int getArea() {
        return (width * height);
    }
};

class Triangle: public Shape {
public:
    int getArea() {
        return (width * height)/2;
    }
};

class Circle: public Shape {
public:
    double radius;
    void setRadius(int r) {
        width = r*2;
        height = r*2;
        radius = r;
    }

    int getArea() {
        return (radius * radius * M_PI);
    }
};

```



Open-Closed Principle: Solution

```
// Some main to test your code.

int main(void) {
    Rectangle rectangle;
    Triangle triangle;

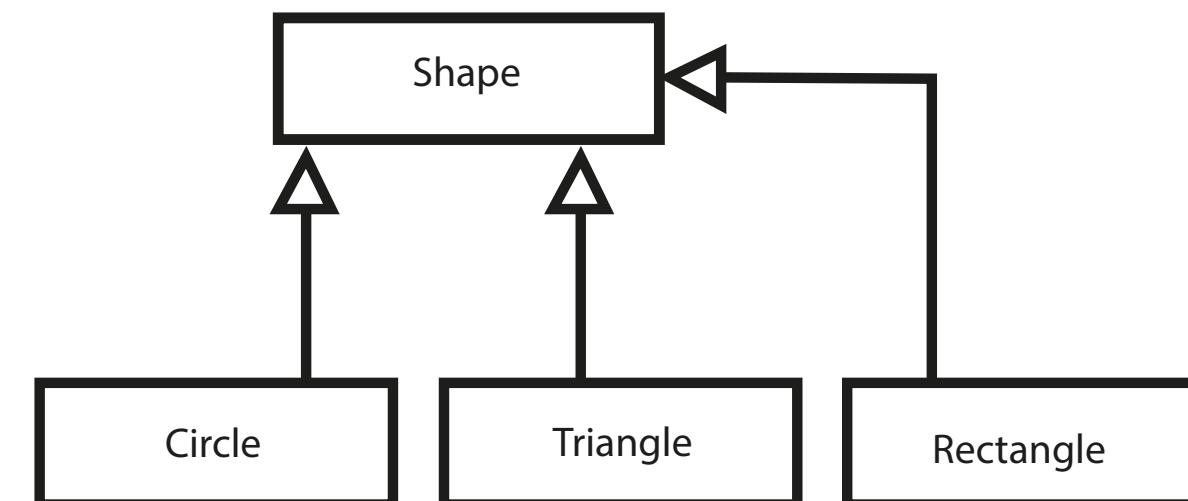
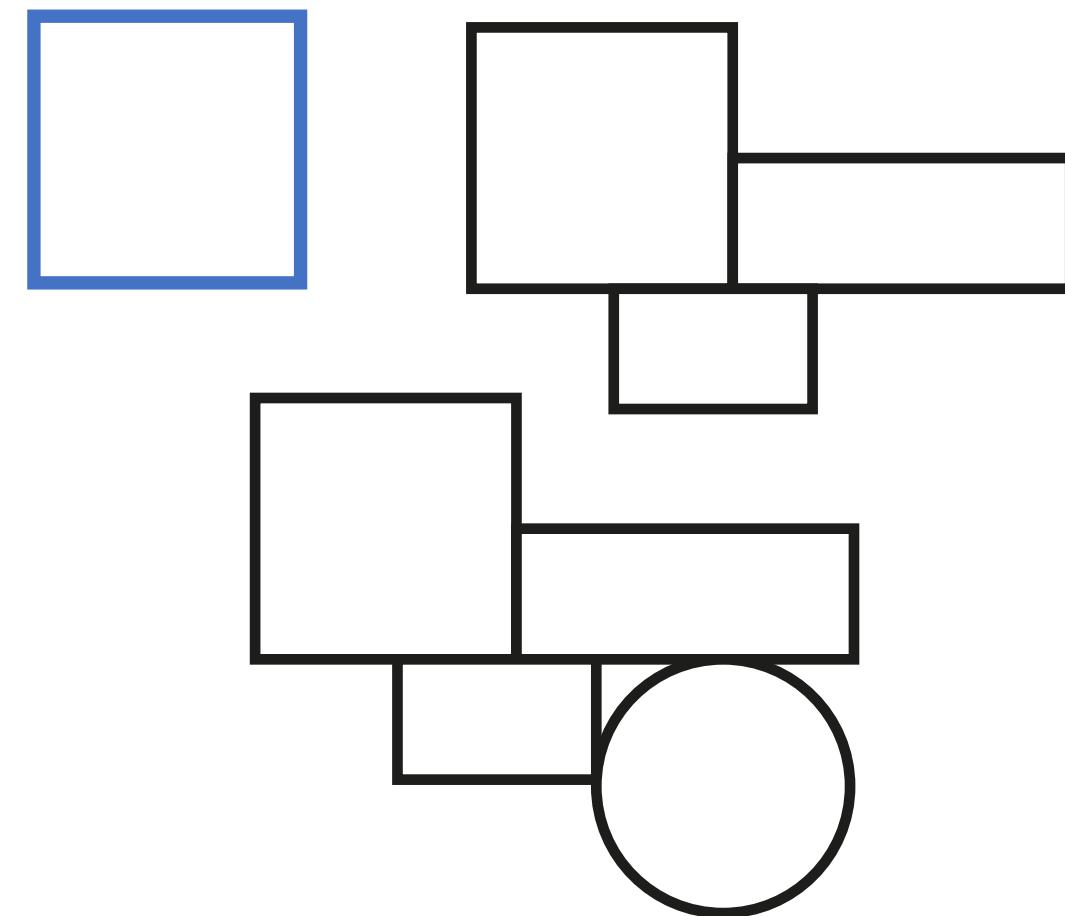
    rectangle.setWidth(5);
    rectangle.setHeight(7);

    // Print the area of the object.
    cout << "Total Rectangle area: " << rectangle.getArea() << endl;

    triangle.setWidth(5);
    triangle.setHeight(7);

    // Print the area of the object.
    cout << "Total Triangle area: " << triangle.getArea() << endl;

    return 0;
}
```



Patterns: Delegation

Jos put in his example “constructor delegation”

What is delegation:

- you delegate a task to some other method or even class

We might touch delegation later on, in protocol oriented languages delegation is key.

Delegation also helps to sustain the Single Responsibility principle.

Typical Sensor Library

<https://www.arduino.cc/reference/en/libraries/category/sensors/>

Issues with libraries:

https://github.com/DFRobot/DFRobot_VL6180X/blob/master/DFRobot_VL6180X.cpp

Can you spot the violation of Single Responsibility?

What about this one?

<https://github.com/pololu/lis3mdl-arduino/blob/master/LIS3MDL.cpp>

Typical Sensor Library

So:

- Wire is the i2c helper object for Arduino
- SAMDx1 e.g. uses custom / multiple wires, is that possible with previous libraries?
- What if you use multiple sensors of the same or other type on the same bus?

Solution:

You could inject the Wire-object of choice using dependency injection.

Also:

Beware of side-effects, who is in control of the i2c bus, your sensor library or your main program (or even something in between, maybe some helper class).

Patterns: Dependency Injection

- UML is not an annoying extra: it is a way to keep your thinking structured and document your software in a visual way also for non-programmers.
- Even though Design Patterns are sometimes abstract really abstract (and initially maybe somewhat confusing) they help you to keep your software structured.
- Uncle Bob has excellent views on Software Craftsmanship so please listen to his advices.

Links

https://www.tutorialspoint.com/compile_cpp_online.php

<https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/>

https://github.com/ksvbka/design_pattern_for_embedded_system/blob/master/design-patterns-for-embedded-systems-in-c-an-embedded-software-engineering-toolkit.pdf

Visual Studio Code Terminals Configuration:

<https://www.youtube.com/watch?v=E9C3M0XIndM>

“

Any questions?