

DESIGN PATTERNS

COURSE 2

CONTENT

❑ Fundamental principles of OOP

- ❑ Encapsulation
- ❑ Inheritance
- ❑ Abstractisation
- ❑ Polymorphism
- ❑ [Exception Handling]

❑ Fundamental Patterns

- ❑ Inheritance
- ❑ Delegation
- ❑ Interface
- ❑ Abstract superclass
- ❑ Inheritance and abstract superclass
- ❑ Immutable objects
- ❑ Marker interface

❑ OOD key principles

- ❑ SRP – Single Responsibility Principle
- ❑ OCP - Open Close Principle
- ❑ LSP – Liskov Substitution Principle
- ❑ ISP – Interface Segmentation Principle
- ❑ DIP – Dependency Inversion Principle
- ❑ DRY – Don't Repeat Yourself

FUNDAMENTALS

PRINCIPLES OF OOP

❑ Objects

- ❑ Describe characteristics (properties) and behavior (methods) of real life objects

❑ Object Oriented language

❑ Encapsulation

- ❑ hide unnecessary details and provide a clear and simple interface for working with them

❑ Inheritance

- ❑ improve code readability and enable the reuse of functionality

❑ Abstraction

- ❑ deal with objects considering their important characteristics and ignore all other details

❑ Polymorphism

- ❑ how to work in the same manner with different objects

❑ [Error handling]

- ❑ the process of responding to the occurrence, during computation, of *exceptions* – anomalous or exceptional conditions requiring special processing – often changing the normal flow of program execution.

FUNDAMENTALS PRINCIPLES OF OOP. ABSTRACTION

❑ Abstraction

- ❑ Problem -> Model

❑ Model

- ❑ Data
- ❑ Operations
- ❑ A simplification with a scope of a problem
 - ❑ Simple model => accessible code
- ❑ Model Views
 - ❑ A view for system major parts interaction
 - ❑ A view of system details
 - ❑ A view from user point of view
 - ❑ ...
- ❑ Common notation
 - ❑ Unified Modeling Model (UML)

Example

- ❑ How does a person see a computer?
 - ❑ Child: a device for gaming
 - ❑ Electronics: an assembly of circuits and transistors
 - ❑ Programmers: an environment for developing tools

FUNDAMENTALS PRINCIPLES OF OOP.

ENCAPSULATION

- ❑ Hide unnecessary the properties and behavior of objects
- ❑ Reduce the necessary knowledge about a class, in order to user it
 - ❑ In many cases the programmer does not need to know implementation details of a class, if the class offers the desired behavior

```
class WithoutEncapsulationOrInformationHide
{
    public static final int STATUS_ACTIVE = 0;
    public static final int STATUS_HALTED = 1;
    public int status = STATUS_ACTIVE;
};
```

FUNDAMENTALS PRINCIPLES OF OOP.

ENCAPSULATION

- ❑ Hide unnecessary the properties and behavior of objects
- ❑ Reduce the necessary knowledge about a class, in order to user it
 - ❑ In many cases the programmer does not need to know implementation details of a class, if the class offers the desired behavior

```
class WithoutEncapsulationOrInformationHide
{
    public static final int STATUS_ACTIVE = 0;
    public static final int STATUS_HALTED = 1;
    public int status = STATUS_ACTIVE;
};
```

```
class EncapsulationWithoutInformationHide{
    public static final int STATUS_ACTIVE = 0;
    public static final int STATUS_HALTED = 1;
    private int status = STATUS_ACTIVE;
    public int getStatus() {
        return status;
    }
};
```

FUNDAMENTALS PRINCIPLES OF OOP.

ENCAPSULATION

- ❑ Hide unnecessary the properties and behavior of objects
- ❑ Reduce the necessary knowledge about a class, in order to user it
- ❑ In many cases the programmer does not need to know if the class offers the

```
class EncapsulationAndInformationHide {  
    public static final int STATUS_ACTIVE = 0;  
    public static final int STATUS_HALTED = 1;  
    private int status = STATUS_ACTIVE;  
    private int getStatus() {  
        return status;  
    }  
    public boolean isActive() {  
        return getStatus() == STATUS_ACTIVE;  
    }  
};
```

```
class EncapsulationWithoutInformationHide{  
    public static final int STATUS_ACTIVE = 0;  
    public static final int STATUS_HALTED = 1;  
    private int status = STATUS_ACTIVE;  
    public int getStatus() {  
        return status;  
    }  
};
```

FUNDAMENTALS PRINCIPLES OF OOP.

INHERITANCE

❑ Inheritance is a mechanism which allows a class A to inherit members (data and functions) of a class B. We say “A inherits from B”. Objects of class A thus have access to members of class B without the need to redefine them.

❑ Terminology

❑ Base class

❑ The class that is inherited

❑ Derived class

❑ A specialization of base class

❑ Kind-of relation

❑ Class level (Circle is a kind-of Shape)

❑ Is-a relation

❑ Object level (The object circle1 is-a shape.)

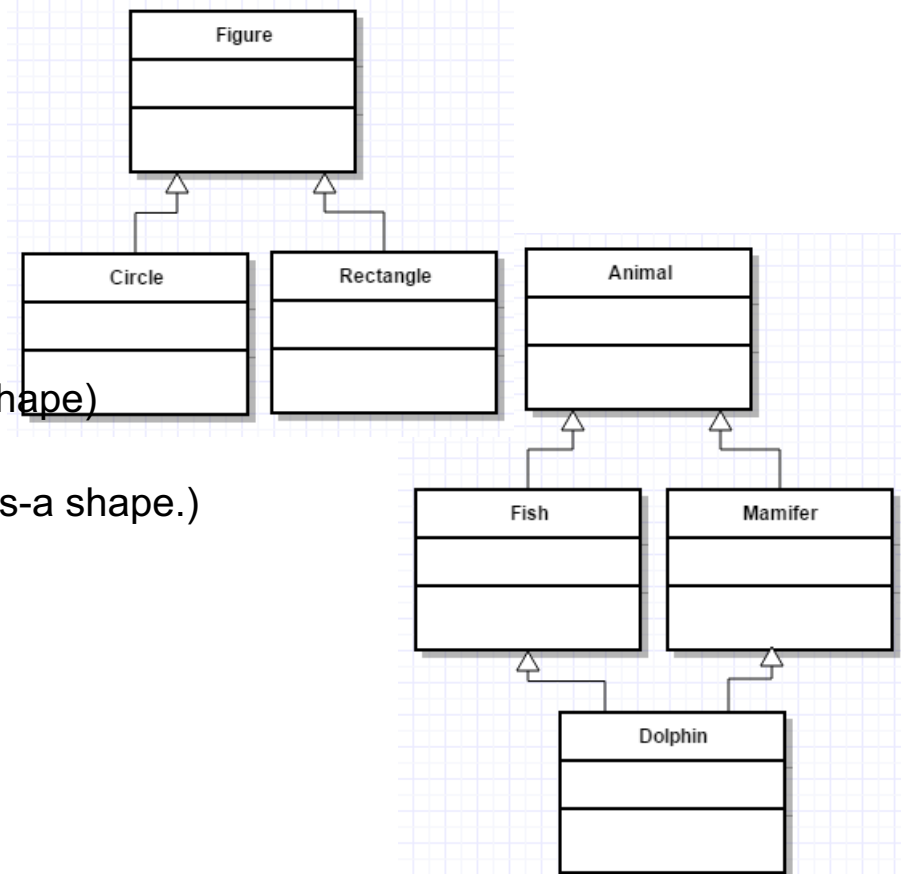
❑ Types of inheritance

❑ Simple

❑ One base class

❑ Multiple

❑ Multiple base classes



FUNDAMENTALS PRINCIPLES OF OOP.

POLYMORPHISM

❑ **Polymorphism the ability to use a thing in different ways**

❑ Run-time

❑ Inheritance

❑ Virtual functions (C++)

❑ Generics (Java)

❑ Compile-time

❑ Templates (C++)

❑ Ad-hoc

❑ Operator overloading (in C++)

❑ Parametric

❑ Casting

FUNDAMENTALS PRINCIPLES OF OOP. ERROR HANDLING

☐ **An exception is an error that appears at run-time.**

☐ **Examples**

☐ Out of memory

☐ File already opened

☐ Null pointer exception

☐ **Variants to resolve such situations**

☐ Custom mechanism

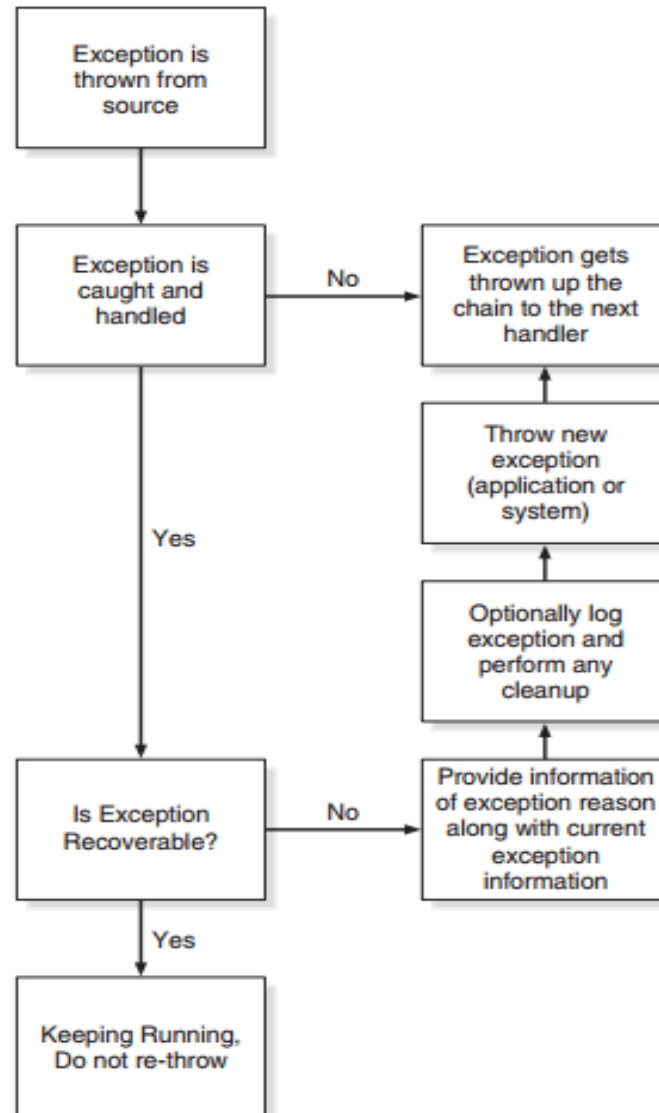
☐ Program stops -> unacceptable solution

☐ Return of an error code -> the state of the program has to test the error code returned

☐ A function that is called each time an error occurs -> no control over the caller

☐ Using language mechanism of handling exceptions

FUNDAMENTALS PRINCIPLES OF OOP. ERROR HANDLING



FUNDAMENTAL PATTERNS

☐ Fundamentals patterns

- ☐ Patterns already found permanent in modern programming languages
- ☐ Not classified in other categories

☐ Fundamentals patterns types

- ☐ Inheritance
- ☐ Delegation
- ☐ Interface
- ☐ Abstract superclass
- ☐ Inheritance and abstract superclass
- ☐ Immutable objects
- ☐ Marker interface

FUNDAMENTALS PATTERNS.

DELEGATE

❑ Name: **delegation**

❑ Intent

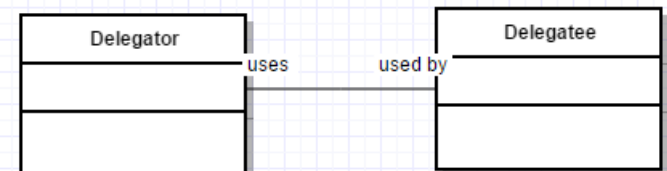
- ❑ Delegation allows objects to share behavior without using inheritance and without duplicating code

❑ Solution

- ❑ Delegation is a way of reusing and extending the behavior of a class. It works writing a new class that incorporates the functionality of the original class by using an instance of the original class and calling its methods.

❑ Consequences

- ❑ Behavior can be changed at run-time (comparing to inheritance that is static)
- ❑ The 'delegate' is hidden to delegator's clients
- ❑ More difficult to implement comparing to inheritance



FUNDAMENTAL PATTERNS.

INTERFACE

❑ Name: **Interface**

❑ Intent

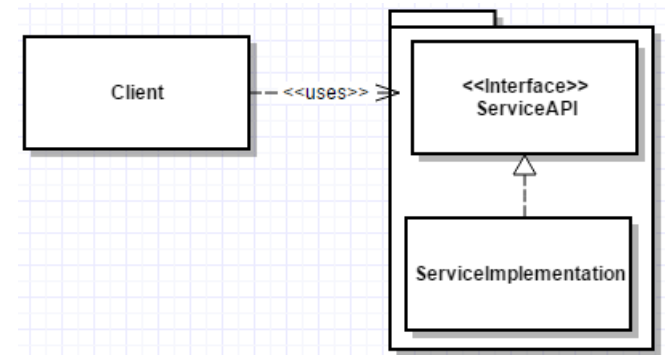
- ❑ Classes change messages between them
- ❑ The implementation must be switched at run time
- ❑ At design-time when the implementation used at compile time is not known

❑ Definition

- ❑ Decouples the service from its clients

❑ Consequences

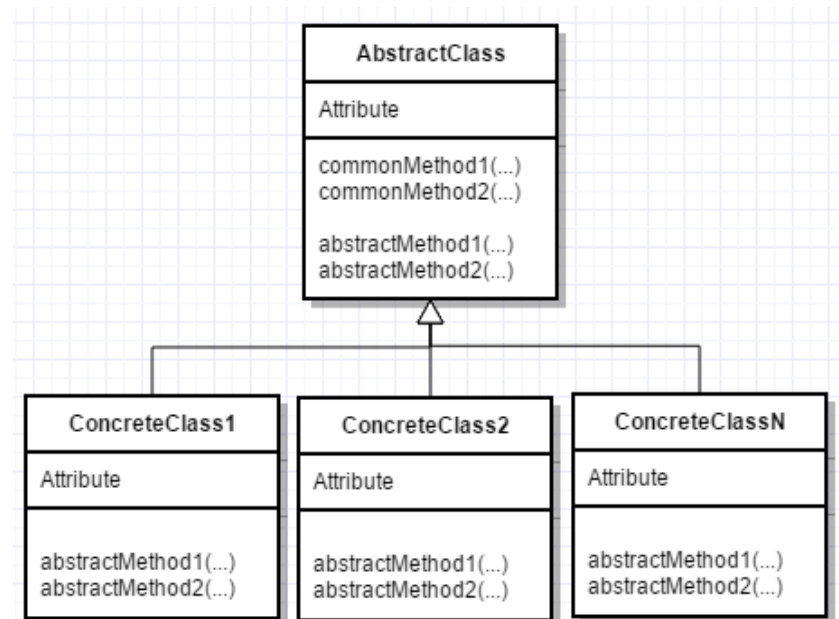
- ❑ Programming to abstraction
- ❑ Easy change the service provider
- ❑ Transparency for client



FUNDAMENTALS PATTERNS.

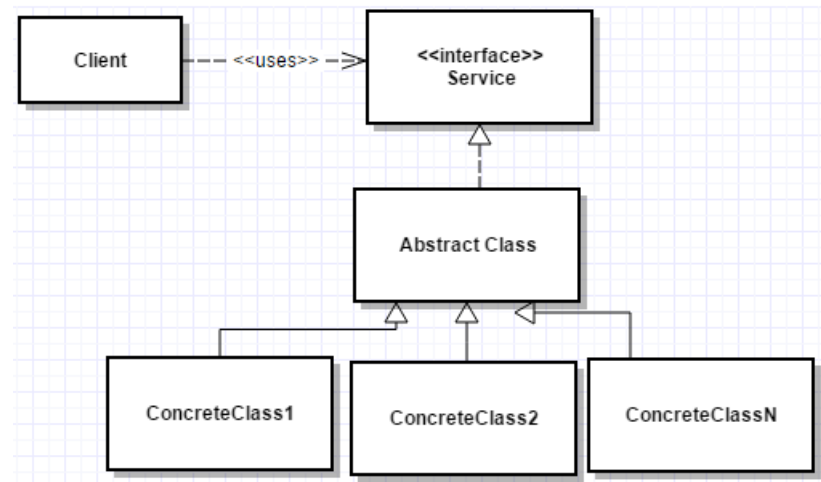
ABSTRACT SUPERCLASS

- ❑ Abstract superclass – ensures consistent behavior for its subclasses
- ❑ Consequences
 - ❑ Common behavior is consistent over subclasses
 - ❑ Clients are using the abstract superclass



FUNDAMENTALS PATTERNS. INTERFACE AND ABSTRACT SUPERCLASS

- ❑ Combines Interface and Abstract
- ❑ Superclass patterns
- ❑ Consequences
 - ❑ Combines the advantages of both patterns
 - ❑ May provide a default behavior for the entire, or just a subset, of the Service interface via AbstractService class



FUNDAMENTALS PATTERNS.

IMMUTABLE OBJECT

- ❑ **Immutable object** – the internal state of the object doesn't change after its creation
- ❑ **Consequences**
 - ❑ Only constructors can change object's state
 - ❑ All member functions are constant functions (in C++)
 - ❑ Any member function that need to change the state will create a new instance
 - ❑ Increase design's robustness and maintainability
- ❑ **Example**
 - ❑ String class in JDK

FUNDAMENTALS PATTERNS.

IMMUTABLE OBJECT

```
final class Position {  
    private int x;  
    private int y;  
    public Position(int x, int y) {  
        this.x = x;  
        this.y = y;  
    } // Position(int, int)  
    public int getX() { return x; }  
    public int getY() { return y; }  
    public Position offset(int xOffset, int yOffset) {  
        return new Position(x+xOffset, y+yOffset);  
    } // offset(int, int)  
} // class Position
```

FUNDAMENTALS PATTERNS.

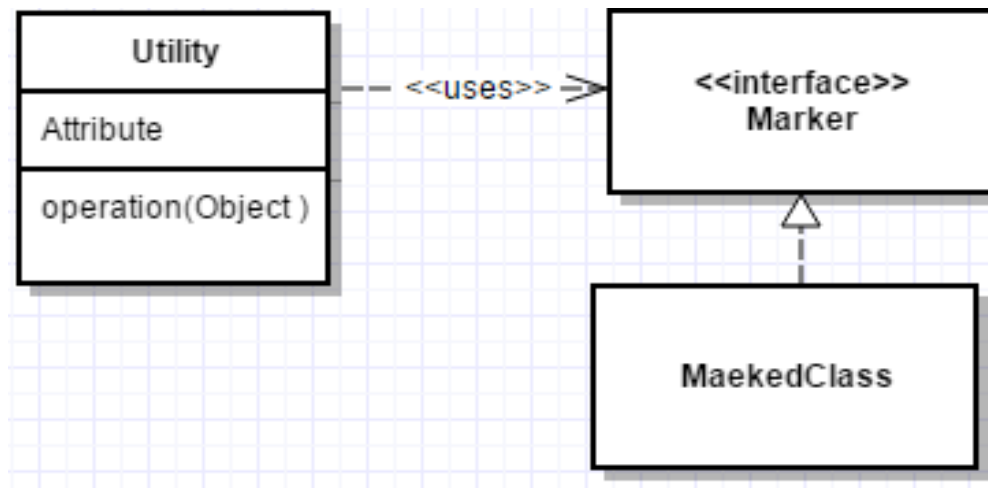
MARKER INTERFACE

- ❑ **A class implements a marker interface in order to support a semantic attribute of the system**
- ❑ **Motivation**
 - ❑ Unrelated concepts do have something in common
 - ❑ However, how to use this information is context-dependent
- ❑ **Consequences**
 - ❑ Used by utility classes that need a specific behavior from their elements, without requesting a common base class
- ❑ **Example**
 - ❑ Cloneable, Serializable, Remote in JDK

FUNDAMENTALS PATTERNS.

MARKER INTERFACE

- ❑ Empty interfaces
- ❑ Are differences between marker interface in Java are annotations? (homework for next course)



OBJECT ORIENTED DESIGN PRINCIPLES

❑ OOD key principles

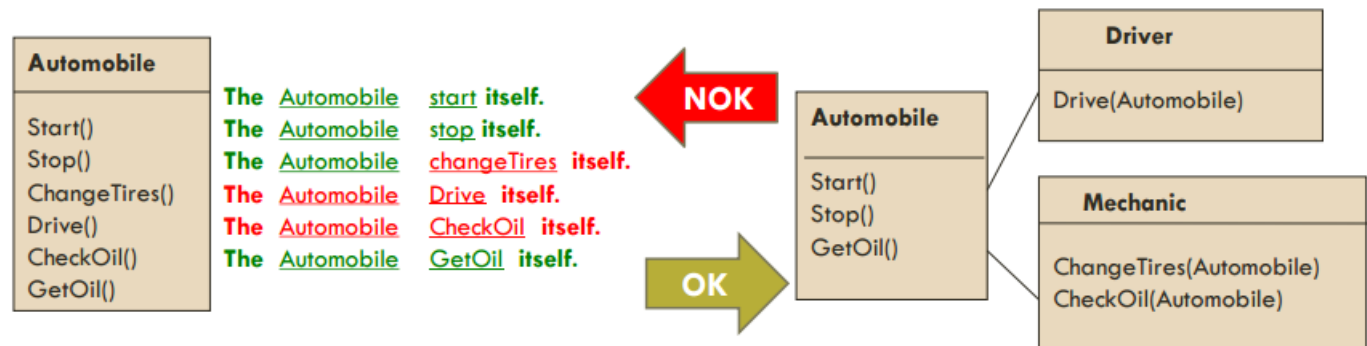
- ❑ collection of best practice, object-oriented design principles which can be applied to design, allowing you to accomplish various desirable goals such as loose-coupling, higher maintainability, intuitive location of interesting code, e.t.c.

❑ Types

- ❑ SRP – Single Responsibility Principle
- ❑ OCP - Open Close Principle
- ❑ LSP – Liskov Substitution Principle
- ❑ ISP – Interface Segmentation Principle
- ❑ DIP – Dependency Inversion Principle
- ❑ DRY – Don't Repeat Yourself

SINGLE RESPONSABILITY PRINCIPLE

- ❑ SRP: Every object in your system should have a single responsibility, and all the object's services should be focused in carrying out that single responsibility.
- ❑ ONLY one reason to change something!
- ❑ Code will be simpler and easier to maintain.
- ❑ Example: Container and Iterator (Container manages objects; Iterator traverses the container)
- ❑ How to spot multiple responsibilities? Forming sentences ending in itself.



OPEN CLOSE PRINCIPLE

❑ OCP – Classes should be open for extension and closed for modification

❑ Allowing change, but without modifying existing code. It offers flexibility.

❑ Use inheritance to extend/change existing working code and don't touch working code.

❑ OCP can also be achieved using composition.

```
class Shape {  
    int type;  
    void drawPolygon () { /* ... */ }  
    void drawPoint () { /* ... */ }  
public:  
    void draw();  
};  
void Shape::draw() {  
    switch(type) {  
        case POLYGON:  
            drawPolygon (); break;  
        case POINT:  
            drawPoint (); break;  
    }  
}
```

NOK

OK

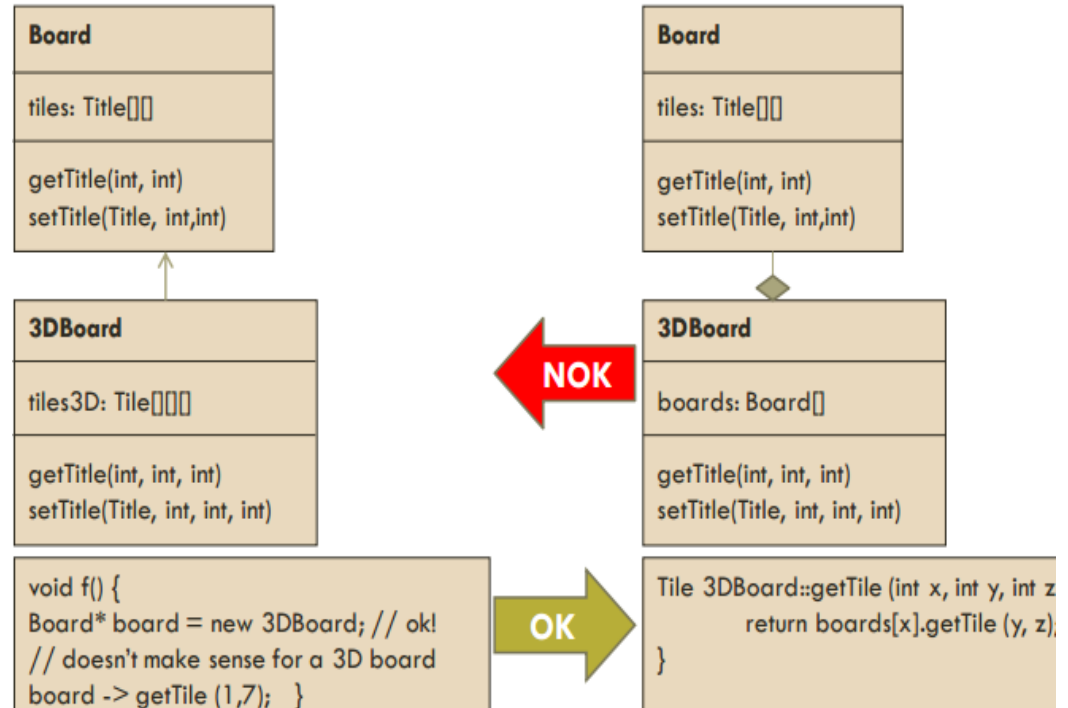
```
class Shape {  
public:  
    virtual void draw() = 0;  
};  
class Polygon : public Shape {  
public:  
    void draw();  
};  
class Point : public Shape {  
public:  
    void draw();  
};  
void Polygon::draw() { /* ... */ }  
void Point::draw() { /* ... */ }
```

LSKOV SUBSTITUTION PRINCIPLE

❑ LSP: Subtypes must be substitutable for their base types.

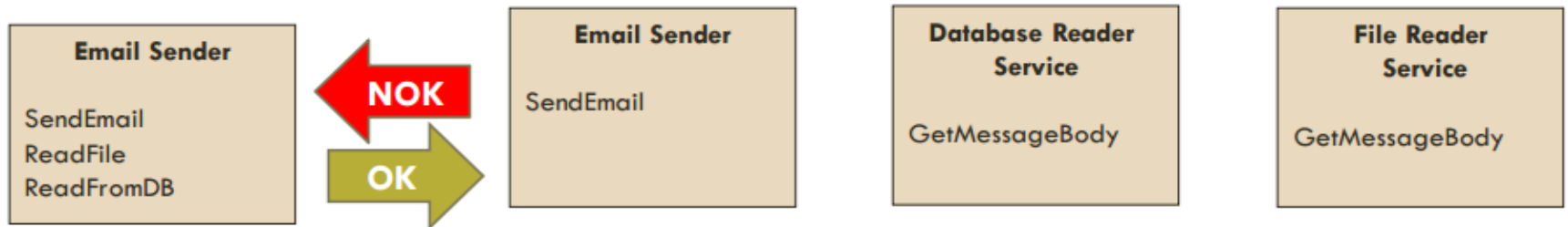
❑ Well-designed class hierarchies

❑ Subtypes must be substitutable for their base class without things going wrong.



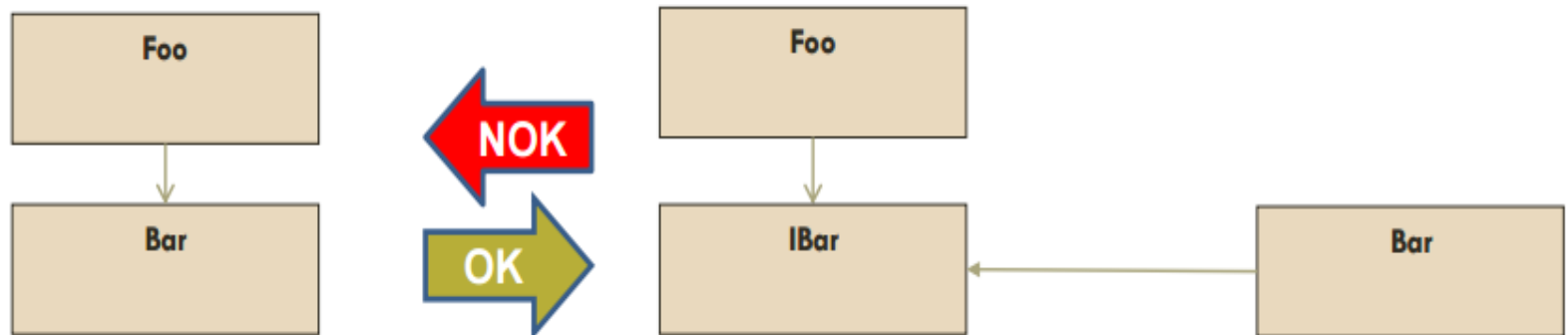
INTERFACE SEGMENTATION PRINCIPLE

- ❑ **ISP: Clients should not be forced to depend on methods they do not use**
- ❑ **Keep interfaces small, cohesive, and focused**
- ❑ **Whenever possible, let the client define the interface**



DEPENDENCY INVERSION PRINCIPLE

- ❑ High-level modules should not depend on low-level modules. Both should depend on abstractions
- ❑ Abstractions should not depend on details. Details should depend upon abstractions
- ❑ Detail should be dependent on Policy. This means that you should have the Policy define and own the abstraction that the detail implements.



DEPENDENCY INVERSION PRINCIPLE

// **Bad example**

```
class Worker {  
    public void work() {  
        // ....working  
    }  
}  
  
class Manager {  
    Worker worker;  
  
    public void setWorker(Worker w) {  
        worker = w;  
    }  
  
    public void manage() {  
        worker.work();  
    }  
}  
  
class SuperWorker {  
    public void work() {  
        //.... working much more  
    }  
}
```

// **Good example**

```
interface IWorker {  
    public void work();  
}  
  
class Worker implements IWorker{  
    public void work() {  
        // ....working  
    }  
}  
  
class SuperWorker implements IWorker{  
    public void work() {  
        //.... working much more  
    }  
}  
  
class Manager {  
    IWorker worker;  
  
    public void setWorker(IWorker w) {  
        worker = w;  
    }  
  
    public void manage() {  
        worker.work();  
    }  
}
```

DON'T REPEAT YOURSELF

- ❑ DRY: Avoid duplicate code by abstracting out things that are common and placing those things in a single location.
- ❑ No duplicate code => ONE requirement n ONE place!
- ❑ This principle can and should be applied everywhere (e.g. in Analysis phase –don't duplicate requirements or features!)
- ❑ Code is easier and safer to maintain because we have to change only one place.

```
String::String(const char* pch) {  
    if(pch!=NULL) {  
        str = new char[(sz=strlen(pch))+1];  
        strcpy(str, pch);  
    } else {  
        str = NULL;  
        sz = 0;  
    }  
}  
  
void String::set(const char* pch) {  
    if(str!=NULL) delete [] str;  
    if(pch!=NULL) {  
        str = new char[(sz=strlen(pch))+1];  
        strcpy(str, pch);  
    } else {  
        str = NULL;  
        sz = 0;  
    }  
}
```

NOK

OK

```
/*private*/ void String::init(const char* pch) {  
    if(pch!=NULL) {  
        str = new char[(sz=strlen(pch))+1];  
        strcpy(str, pch);  
    }  
    else {  
        str = NULL;  
        sz = 0;  
    }  
}  
  
String::String(const char* pch) {  
    init(pch);  
}  
  
void String::set(const char* pch) {  
    if(str!=NULL) delete [] str;  
    init(pch);  
}
```