SOLID Principles

Design Principles and Approaches

S	Single Responsibility
O	Open/Closed
L	Liskov substitution
I	Interface Segregation
D	Dependency Inversion

SoftUni Team

Technical Trainers

Software University

Software University

https://softuni.bg

SoftUni

Table of Contents

Questions

sli.do

#csharp-advanced

Why Clean Code Matters?

- How **clean code** (or its absence) **affects** our software?

```
"...So if you want to go fast,
 if you want to get done quickly,
 if you want your code to be easy to write,
 make it easy to read."
                          - Robert C. Martin
```

Single Responsibility

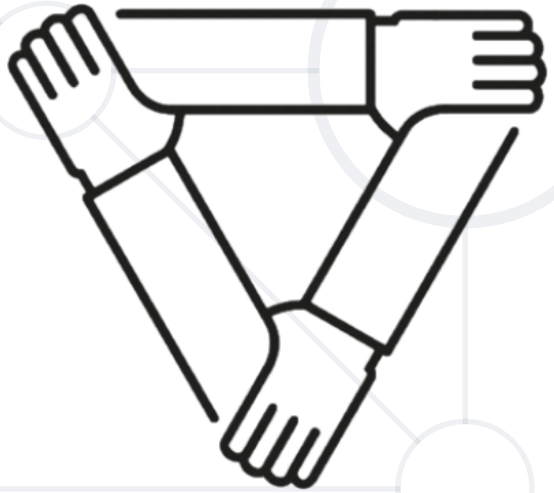
What is Single Responsibility?

- Every class should be responsible **for only a single part of the functionality** and that responsibility should be entirely **encapsulated** by the class.

```
"There should never be more than
one reason for a class to change."
     - Robert C. "Uncle Bob" Martin
```

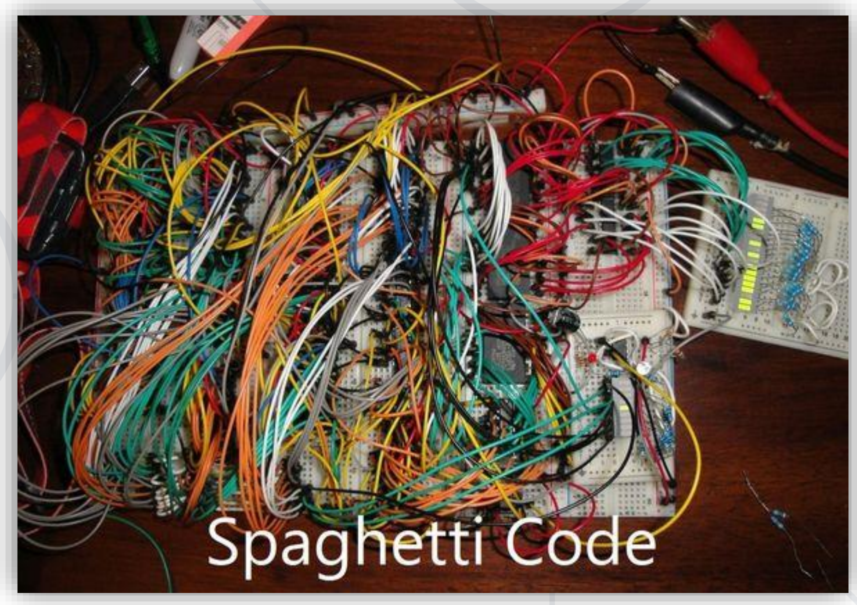
Strong Cohesion

- **Cohesion** refers to the grouping of **functionally related processes** into a particular module.
- Aim for **strong cohesion**
  - Each **task** maps a **single** code unit
  - A method should do **one operation**
  - A class should represent **one entity**



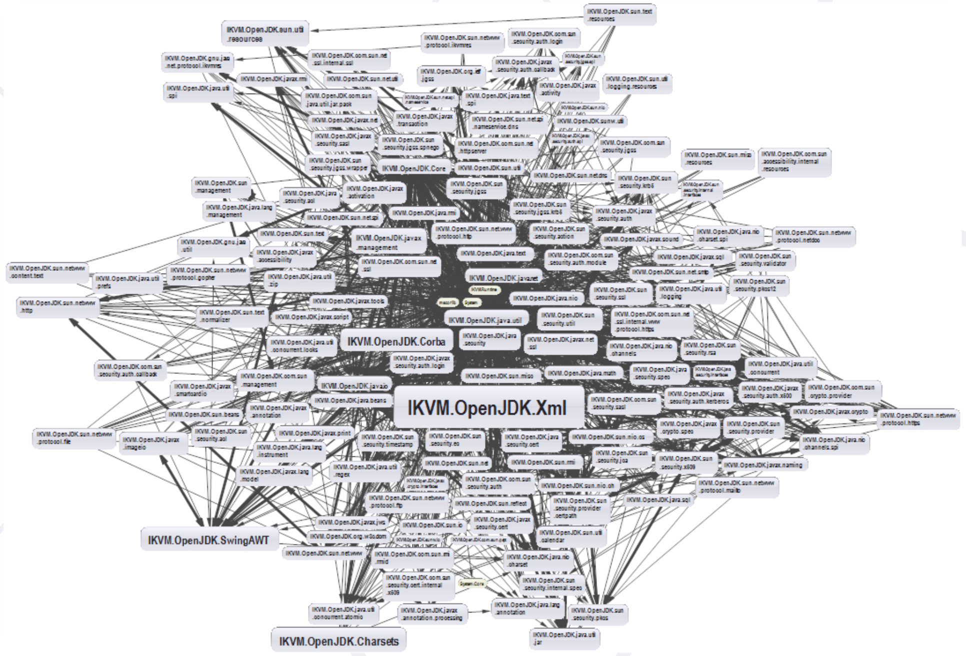
Loose Coupling

- **Coupling** - the degree of dependence between modules
  - How closely connected two modules are
  - The strength of the relationship between modules
- Aim for **loose** coupling
  - Supports **readability** and **maintainability**
  - Often a sign of good system **design**

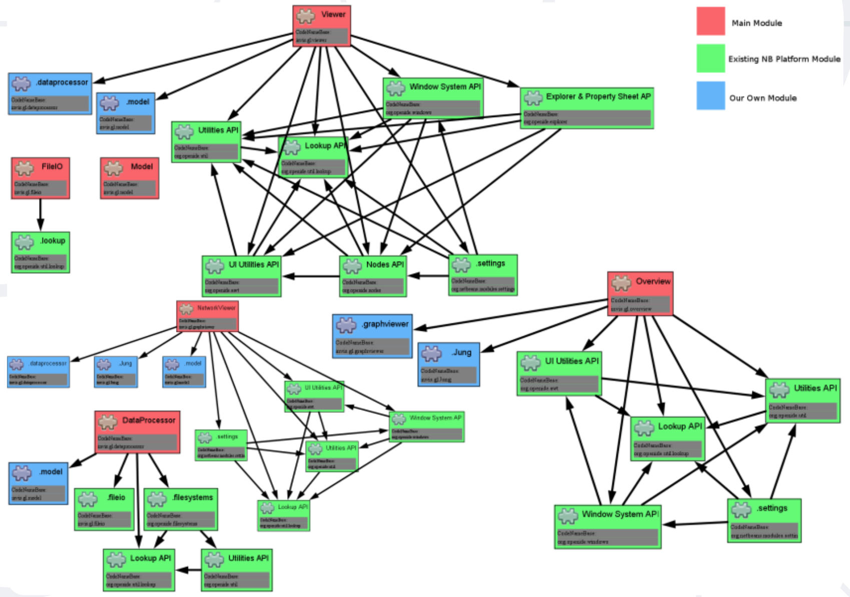


Dependencies and Coupling

- Depend directly on other modules

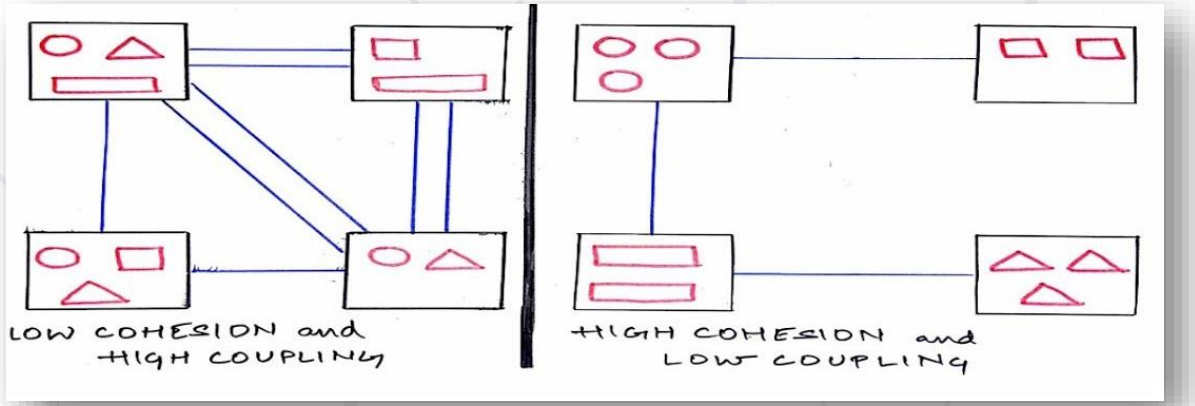


- Depend on abstractions



Cohesion and Coupling – Approaches

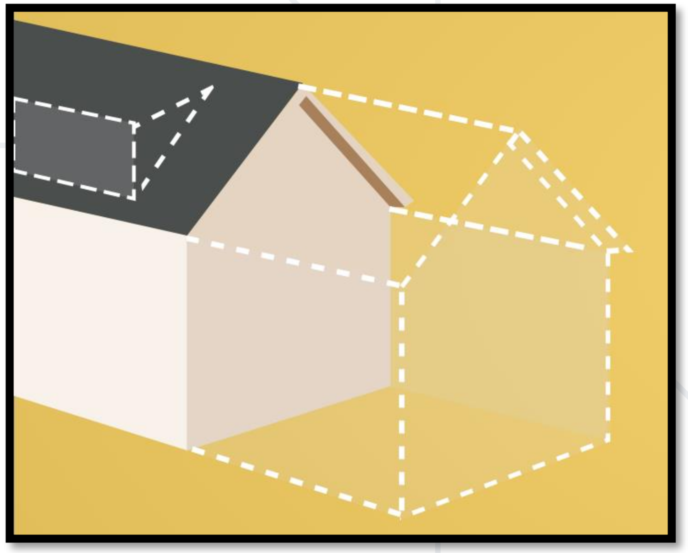
- **Small number** of instance variables inside a class
- Each method of a class should manipulate **one or more** of those variables
- Two modules should **exchange** as **little information** as possible
- Use **abstraction**
- Creating an **easily reusable** subsystem


LOW COHESION and HIGH COUPLING
HIGH COHESION and LOW COUPLING

Open/Closed

What is the Open/Closed Principle?

- Software entities like **classes**, **modules** and **functions** should be **open** for **extension**, but **closed** for **modifications**
- **Extensibility**
  - Adding a new behavior **doesn't require** changes over existing source code
- **Reusability**
  - subsystems are **suitable for reusing** in other projects - modularity



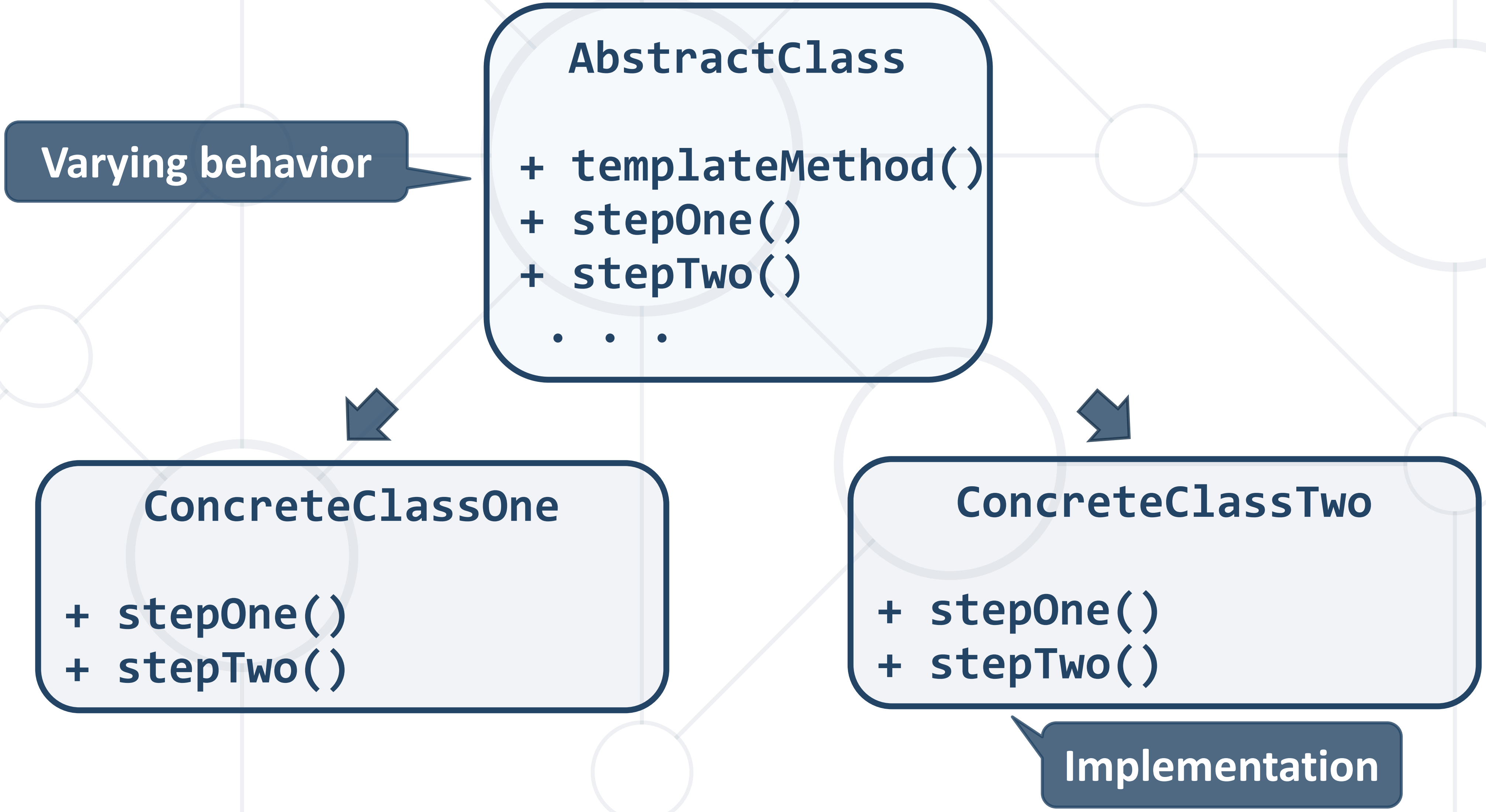
OCP – Approaches

- Parameters
  - Control behavior specifics via a **parameter** or a **delegate**
- Rely on abstraction, **not implementation**
  - Inheritance / Template Method Pattern
- Strategy Pattern
  - Plug in model (insert a new implementation of the interface)

OCP – When to Apply

- By experience - know the problem domain and if a **change** is very **likely** to **recur**
- New domain problem - implement the **most simple** way
  - Changes once - **modify**, second time - **refactor**
- TANSTAAFL - There Ain't No Such Thing As A Free Lunch
  - OCP adds **complexity** to design
  - No design can be **closed against all changes**
- Need to **retest (recheck functionality)** after changes

Template Method Pattern (1)



Template Method Pattern (2)

```
public abstract class CrossCompiler
{
  public void CrossCompile()
  {
    // some functionality…
    this.CollectSource();
    // some functionality…
    this.CompileToTarget();
    // some functionality…
  }

  protected abstract void CollectSource();
  protected abstract void CompileToTarget();
}
```

Template method

Template Method Pattern (3)

```
public class IPhoneCompiler : CrossCompiler
{
  protected override void CollectSource()
  protected override void CompileToTarget()
  { // IPhone specific compilation }
}
```

```
public class AndroidCompiler : CrossCompiler
{
  protected override void CollectSource()
  protected override void CompileToTarget()
  { // Android specific compilation }
}
```

Liskov Substitution

LSP – Substitutability

- Derived types must be completely **substitutable** for their base types
- Derived classes
  - only **extend** functionalities of the base class
  - must **not** remove **base** class **behavior**

```
Student IS-SUBSTITUTED-FOR Person
```

Design Smell – Violations

- Type Checking
- Overridden methods say "*I am not implemented*"
- Base class depends on its subtypes

LSP – Approaches

- Tell Don't Ask
  - If you need to check what is the object - move the behavior **inside the object**
- New Base Class - if **two classes** share a common behavior, but are not substitutable, create a third, from which **both derive**
- There **shouldn't** be any **virtual methods** in constructors

Interface Segregation

What is Interface Segregation?

- Segregate interfaces
  - Prefer **small**, **cohesive** (lean and focused) interfaces
  - Divide "**fat**" interfaces into "**role**" interfaces

```
"Clients should not be forced to depend
on methods they do not use."
- Agile Principles, Patterns and Practices in C#
```

Fat Interfaces

- Classes whose interfaces are not cohesive have "fat" interfaces

```
public interface IWorker
{
  void Work();
  void Sleep();
}
```

```
public class Robot : IWorker
{
  void Work() { … }
  void Sleep()
    { throw new NotImplementedException() }
}
```

Design Smells – Violations

- **Not implemented** methods
- A Client references a class, but only uses a **small portion** of it

```
"Abstraction is elimination
of the irrelevant and
amplification of the essential."
                  - Robert C. Martin
```

ISP – Approaches

- What does the client **see** and **use**?
- The "**fat**" interfaces implement a **number of small** interfaces with just what you need
- All **public members** of a class divided in **separate classes** - again, could be thought of as an interface
- Let the **client define interfaces** - "**role**" interfaces

Cohesive Interfaces

- Small and Cohesive "**Role**" Interfaces

```
public interface IWorker
{
  void Work();
}
public interface ISleeper
{
  void Sleep();
}
public class Robot : IWorker
{
  void Work() { // Do some work… }
}
```

Adapter Pattern

- Problem that the **Adapter pattern** solves
  - **Reusing** classes that do not have an **interface** that a client requires
  - Making classes with **incompatible** interfaces work together
  - Providing **an alternative** interface for a class

Adapter Pattern (1)

- Convert the **incompatible** interface of a class Adaptee into another interface - Target, that clients require

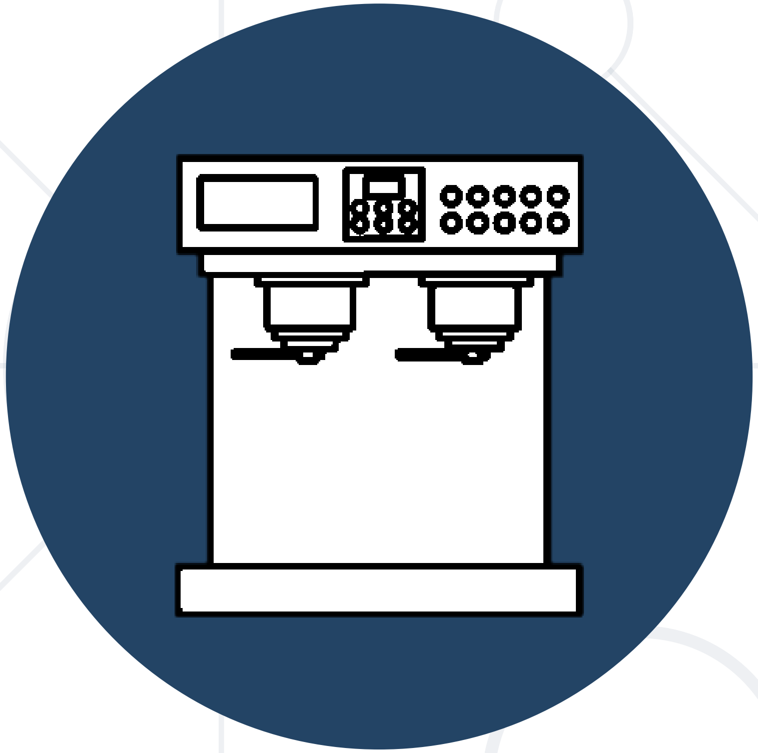
```
class Adaptee
{
  public void SpecificRequest()
  {
    Console.Write
        ("Called SpecificRequest()");
  }
}
```

```
interface ITarget
{
  void Request();
}
```

Adapter Pattern (2)

- Define a separate class - Adapter, that does the job

```
class Adapter : ITarget
{
  private Adaptee adaptee = new Adaptee();
  public void Request()
  {
    // Possibly do some other work
    adaptee.SpecificRequest();
  }
}
```

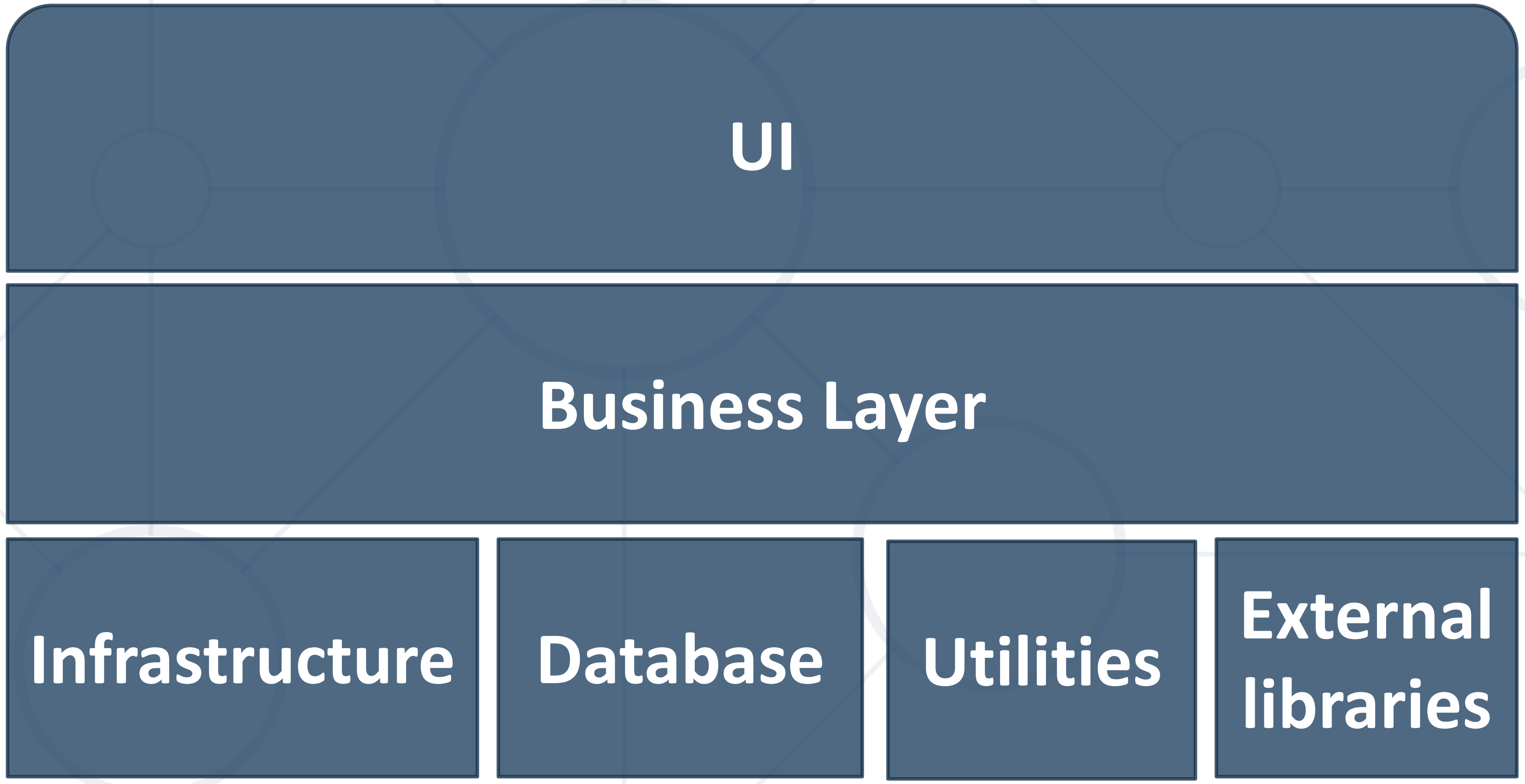


Dependency Inversion

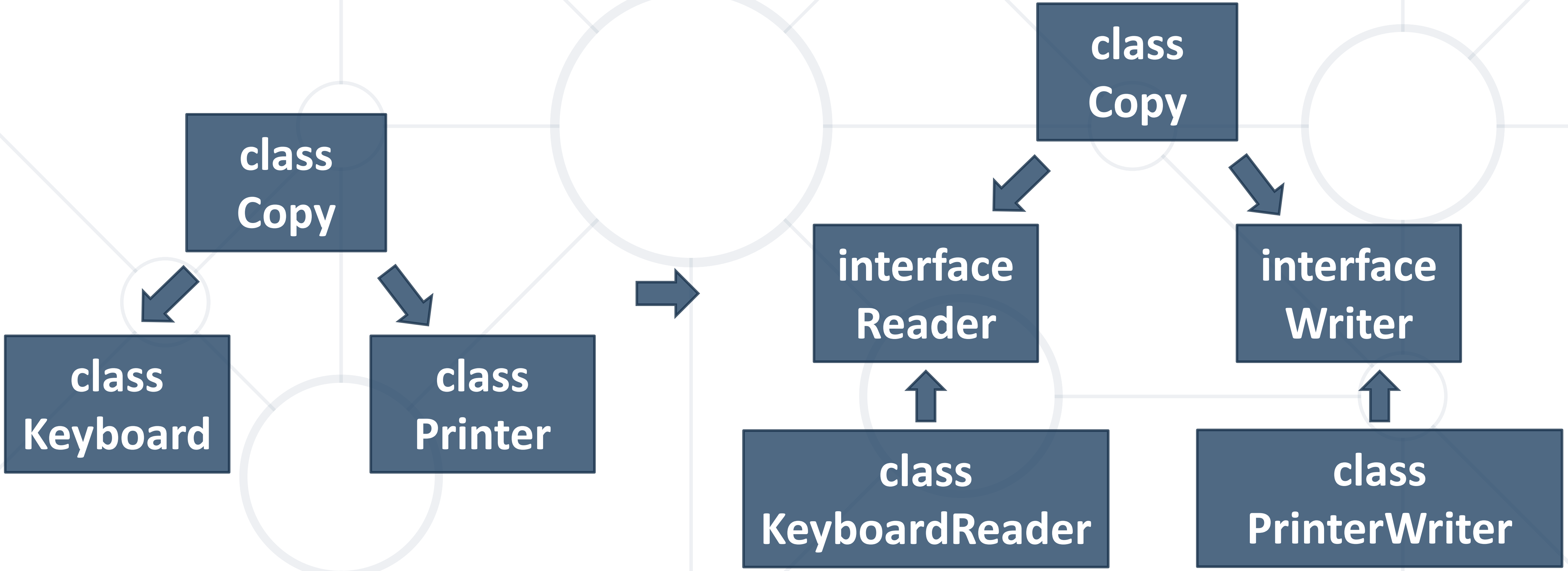
Dependency Examples

- A **dependency** is any external component / system:
  - Framework
  - 3rd party library
  - Database
  - File system
  - Email
  - Web service
  - System resource (e.g. clock)
  - Configuration
  - The **`new`** keyword
  - Static method
  - Global function
  - Random generator
  - Console

Dependencies in Traditional Programming



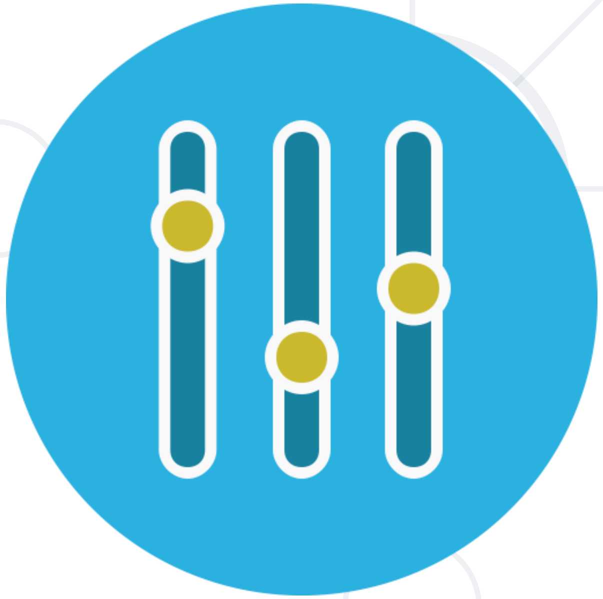
Depend On Abstractions



Types of Dependency Inversion

Constructor injection

Property injection



Parameter injection

Constructor Inversion – Pros and Cons

- Pros
  - Class' requirements are self-documenting
  - We don't have to worry about state validation
- Cons
  - Too many parameters
  - Sometimes, the functionality doesn't need all of the dependencies

Constructor Inversion – Example

```
class Copy
{
    private IReader reader;
    private IWriter writer;
    public Copy(IReader reader, IWriter writer)
    {
        this.reader = reader;
        this.writer = writer;
    }
    // Read/Write data through the reader/writer
}
var copy = new Copy(new ConsoleReader(),
                    new FileWriter("out.txt"));
```

Property Inversion – Pros and Cons

- Pros
  - Functionality can be changed at any time
  - That makes the code very flexible
- Cons
  - State can be invalid
  - Less intuitive to use

Property Inversion – Example

```
class Copy
{
    public IReader Reader { get; set; }
    public IWriter Writer { get; set; }
    public void CopyAllChars()
    {
        // Read/Write data through the reader/writer
    }
}
Copy copy = new Copy();
copy.Reader = new ConsoleReader();
copy.Writer = new FileWriter("output.txt");
copy.CopyAllChars();
```

Parameter Inversion – Pros and Cons

- Pros
  - Changes are only localized to the method

- Cons
  - Too many parameters
  - Breaks the method signature

Parameter Inversion – Example

```
class Copy
{
  public CopyAllChars(IReader reader, IWriter writer)
  {
    // Read/Write data through the Reader/Writer
  }
}
Copy copy = new Copy();
var reader = new ConsoleReader();
var writer = new FileWriter("output.txt");
copy.CopyAllChars(reader, writer);
```

DIP Violations

- Classic DIP Violations:
  - Using the **`new`** keyword
  - Using **`static`** methods / properties
- How to fix code, that violates the DIP:
  - **Extract interfaces** + use **constructor injection**
  - Set up an Inversion of Control (**IoC**) container

Summary

- SOLID principle make software more:
  - Understandable
  - Flexible
  - Maintainable

Software University

Questions?

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

Flutter International

INDEAVR Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

BOSCH

Postbank Решения за твоето утре

PHAR VISION

SmartIT

DXC TECHNOLOGY

createX

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License