Professional Software Engineering

Andrea Carrara and Patrick Berggold

Hritik Singh and Mohab Hassaan – Tutors

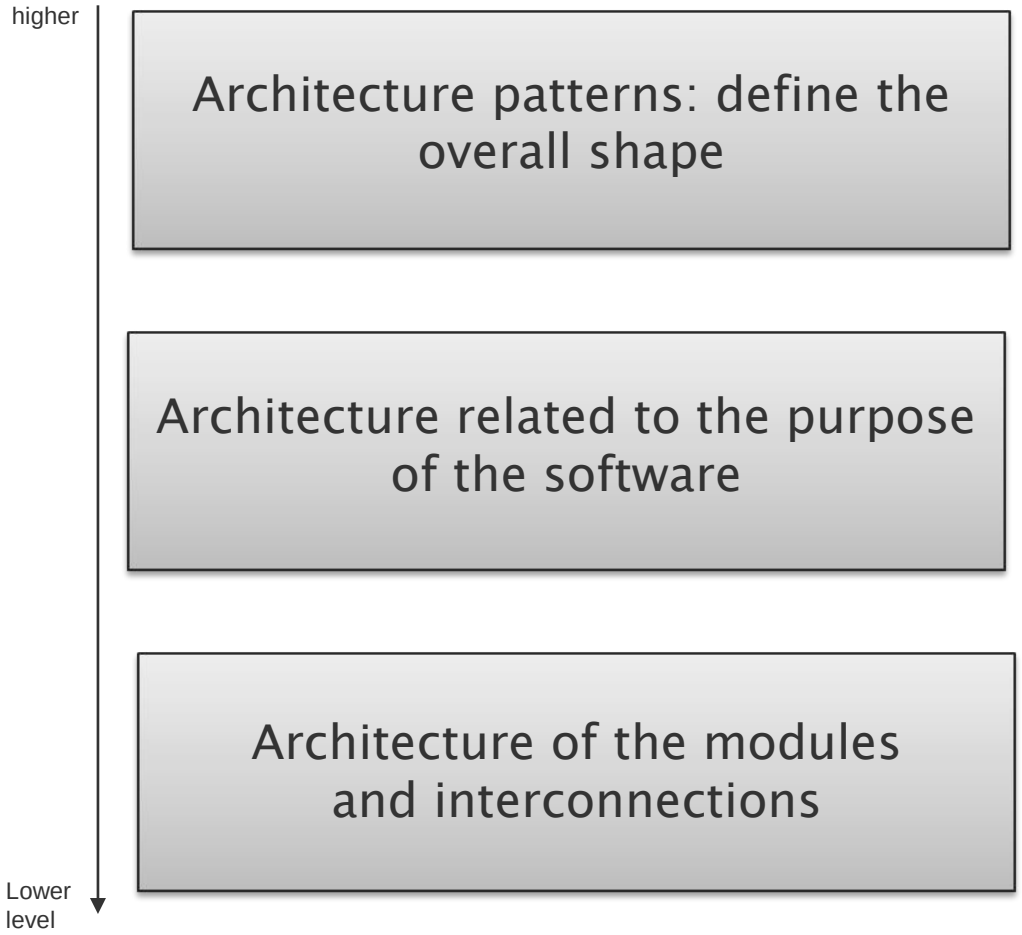
Chair of Computational Modeling and Simulation

andrea.carrara@tum.de; patrick.berggold@tum.de

ARCHITECTURE AND DEPENDENCIES

andrea.carrara@tum.de; patrick.berggold@tum.de

What is Software Architecture?



- Model View Controller (MVC) pattern. Separates the application into three main components: the model, which holds the data; the view, which displays the data; and the controller, which handles user input and updates the model and view.
- The architecture is related to the purpose of the software. I.e. if the software application is an e-commerce platform, the architecture would include features such as a shopping cart, user account management, and payment processing.
- Packages, components, and classes are used to organize and structure the codebase for better maintainability and scalability.

andrea.carrara@tum.de; patrick.berggold@tum.de

What goes wrong with code

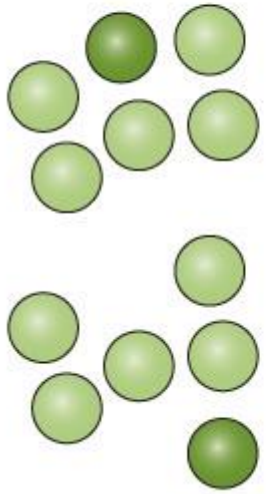
- In the first development and release the code is clean, elegant, and compelling.
- The system starts to degrade with the increase in complexity and requirement changes.
- The program becomes a mass of code that the developers find increasingly hard to maintain
- A redesign is necessary. The old system keeps changing and evolving; the new design must keep up.

andrea.carrara@tum.de; patrick.berggold@tum.de

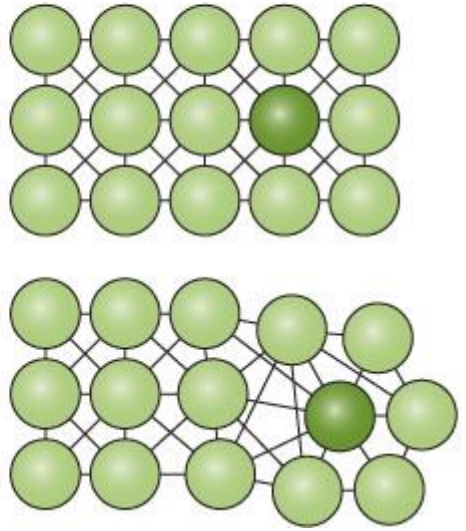
Symptoms of Degrading Design – Rigidity

- A rigid code is **difficult to change, extend or modify**
- Every change causes a **cascade of subsequent changes** in dependent modules
- Caused by **poor design, lack of modularity, tight coupling**
- Unknown time for the change of functionality
- Starts as design deficiency, ends up being adverse management policy

Loosely coupled pieces of code can move around and change their shape freely, just like the molecules in a liquid.



Tightly coupled pieces of code rely on the code around them. Changing one piece is difficult because the other pieces must move to accommodate it.



andrea.carrara@tum.de; patrick.berggold@tum.de

Example of rigidity – Design problems

```
class BankAccount
{
    public decimal Balance { get; set; }

    public void Deposit(decimal amount)
    {
        Balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        if (amount > Balance)
        {
            throw new Exception("Insufficient funds");
        }
        Balance -= amount;
    }
}
```

- The design is tightly coupled with the current requirements of the system
- In case of **requirement changes** of handling different currencies → the class is not able to support the change
- Requirement change of account overdraft protection → modification of the class

andrea.carrara@tum.de; patrick.berggold@tum.de

Example of rigidity – Design improvements

```
interface IAccount
{
    decimal Balance { get; set; }
    void Deposit(decimal amount);
    void Withdraw(decimal amount);
}

class BankAccount : IAccount
{
    public decimal Balance { get; set; }

    public void Deposit(decimal amount)
    {
        Balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        if (amount > Balance)
        {
            throw new Exception("Insufficient funds");
        }
        Balance -= amount;
    }
}
```

- Dividing the account balance and the currency into different classes
- Use interfaces to define the expected behavior and adapt to different currencies or account types
- Possibility do define OverDraftAccount without changing the BackAccount class

andrea.carrara@tum.de; patrick.berggold@tum.de

Symptoms of Degrading Design – Fragility

- Closely related to Rigidity
- Brakes of code in multiple locations when the changes are made
- Problems arise in areas of the code that are not conceptually related to the change
- Each change can break the code more than how it was before

andrea.carrara@tum.de; patrick.berggold@tum.de

Example fragility – Design problems

```
class Bank
{
    public decimal Balance { get; set; }
    public decimal InterestRate { get; set; }
    public List<Account> Accounts { get; set; }

    public void AddAccount(Account account)
    {
        Accounts.Add(account);
    }

    public void RemoveAccount(Account account)
    {
        Accounts.Remove(account);
    }

    public void ApplyInterest()
    {
        foreach (var account in Accounts)
        {
           account.Balance += account.Balance * InterestRate;
        }
    }
}
```

- ApplyInterest method assumes all the accounts in the bank are of the **same type**, they all have a **Balance**, and are affected all by the same interest rate

- In case of **requirement changes** to support different types of accounts (savings or checking account) with different interest rates → ApplyInterest method brakes and need a modification

andrea.carrara@tum.de; patrick.berggold@tum.de

Example fragility - Design improvements

```
interface IAccount
{
    decimal Balance { get; set; }
    void ApplyInterest();
}

class SavingsAccount : IAccount
{
    public decimal Balance { get; set; }
    public decimal InterestRate { get; set; }
    public void ApplyInterest()
    {
        Balance += Balance * InterestRate;
    }
}

class CheckingAccount : IAccount
{
    public decimal Balance { get; set; }
    public decimal InterestRate { get; set; }
    public void ApplyInterest()
    {
        Balance += Balance * InterestRate;
    }
}
```

```
class Bank
{
    public List<IAccount> Accounts { get; set; }
    public void AddAccount(IAccount account)
    {
        Accounts.Add(account);
    }

    public void RemoveAccount(IAccount account)
    {
        Accounts.Remove(account);
    }

    public void ApplyInterest()
    {
        foreach (var account in Accounts)
        {
            account.ApplyInterest();
        }
    }
}
```

- Use an **interface** to define the expected behavior of an account, and then use **polymorphism** to handle different types of accounts.
- More robust to changes

andrea.carrara@tum.de; patrick.berggold@tum.de

Symptoms of Degrading Design – Immobility

- Inability to **reuse software** from other projects or parts of the same.
- The code might have too many baggage dependencies.
- Even though the code is very similar to what you need, the time to clean up the code is more than building it from zero.

andrea.carrara@tum.de; patrick.berggold@tum.de

Symptoms of Degrading Design – Viscosity

- Comes in two forms: viscosity of the design, and viscosity of the environment

- In case of requirement changes there are multiple solutions. **When the design preserving methods are harder to employ than the "hacks"** (shortcuts), then the viscosity of the design is high. It is easy to do the wrong thing, but hard to do the right thing.

- Viscosity of the environment comes about when the **development environment is slow and inefficient.** If compile times are very long, engineers will be tempted to make changes that don't force large recompiles, even though those changes are not optimal from a design point of view

andrea.carrara@tum.de; patrick.berggold@tum.de

Example Design Viscosity – Preserving design method

```
class Shape
{
    public virtual void Draw()
    {
        // code to draw the shape
    }
}

class Circle : Shape
{
    public override void Draw()
    {
        // code to draw a circle
    }
}

class Square : Shape
{
    public override void Draw()
    {
        // code to draw a square
    }
}
```

» Add a new property called "Color" to the Shape class and then update the Draw() method to use this property to set the color of the shape when it is drawn.

» This method **preserves the design** and follows good OOP practices.

```
class Program
{
    // correct way of adding color property
    class ShapeWithColor : Shape
    {
        public Color color { get; set; }
        public override void Draw()
        {
            // code to draw the shape with color
        }
    }
}
```


andrea.carrara@tum.de; patrick.berggold@tum.de

Example Design Viscosity – "Hack" method

```
class Shape
{
    public virtual void Draw()
    {
        // code to draw the shape
    }
}

class Circle : Shape
{
    public override void Draw()
    {
        // code to draw a circle
    }
}

class Square : Shape
{
    public override void Draw()
    {
        // code to draw a square
    }
}
```

- A hackish approach would be to add a global variable called "shapeColor" and then update the Draw() method to use this variable to set the color of the shape when it is drawn.
- This method **does not preserve the design** and is considered a hack.

```
class Program
{
    public static Color shapeColor;
    static void Main(string[] args)
    {
        shapeColor = Color.Red;
        Shape shape = new Circle();
        shape.Draw();
    }
}
```


andrea.carrara@tum.de; patrick.berggold@tum.de

Causes of Degrading Design

Changing requirements

- Requirements changed in a way the initial design did not anticipate.
- The changes often need to be done quickly → the original design changes and the violations accumulate.
- Design needs to be **resilient to the changes**

Dependency management

- Four symptoms are either directly, or indirectly caused by improper dependencies between the modules of the software
- The dependency architecture is degrading → the ability of the software to be maintained degrades

andrea.carrara@tum.de; patrick.berggold@tum.de

SOLID PRINCIPLES

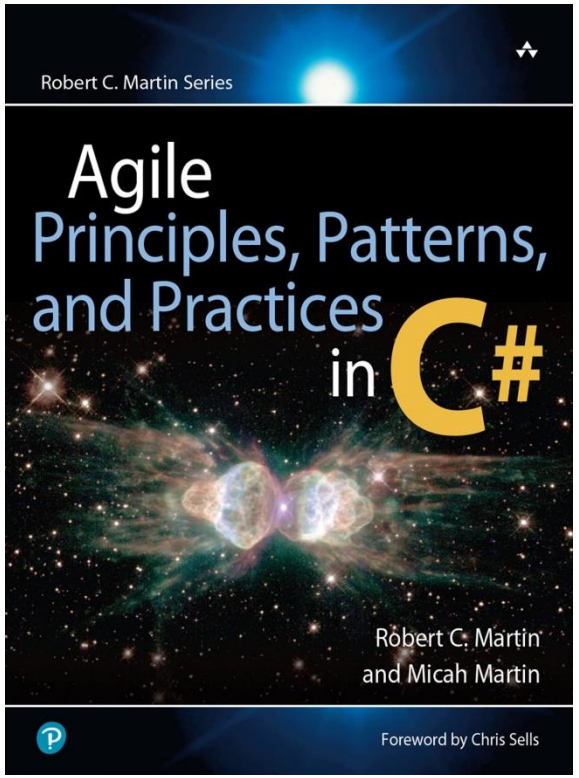
andrea.carrara@tum.de; patrick.berggold@tum.de

History of SOLID

Robert C. Martin is the main person behind these ideas (some individual ideas predate him though)

Single Responsibility Principle (SRP)

- First appeared as a news group posting in 1995.
- Full treatment given in Martin and Martin, Agile Principles, Patterns, and Practices in C#, Prentice Hall, 2006



andrea.carrara@tum.de; patrick.berggold@tum.de

SOLID PRINCIPLES

Five important OO design principles:

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface-Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

andrea.carrara@tum.de; patrick.berggold@tum.de

Single Responsibility Principle (SRP)

A class should have only one responsibility and not multiple.

- Every object, class, and method needs to have a single responsibility

- Risk of ending up with spaghetti code

andrea.carrara@tum.de; patrick.berggold@tum.de

Example – SRP violation

```
// Violation of SRP
class Customer
{
    public string Name { get; set; }
    public string Address { get; set; }
    public string Phone { get; set; }
    public void Save() { /* code to save customer to
database */ }
    public void SendInvoice() { /* code to send invoice
to customer */ }
    public void SendReminder() { /* code to send
reminder to customer */ }
}
```

- The class customer has three responsibilities:
  - Holding Customer data
  - Saving the customer data to a database
  - Sending invoices and reminders to the customer

andrea.carrara@tum.de; patrick.berggold@tum.de

Example – Code Refactoring

```
// Adhering to SRP
class Customer
{
    public string Name { get; set; }
    public string Address { get; set; }
    public string Phone { get; set; }
}

class CustomerRepository
{
    public void Save(Customer customer) { /* code to
save customer to database */ }
}

class InvoiceService
{
    public void SendInvoice(Customer customer) { /* code
to send invoice to customer */ }
    public void SendReminder(Customer customer) { /*
code to send reminder to customer */ }
}
```

- Customer class has only one responsibility: holding customer data.
- The CustomerRepository class has the responsibility of saving customer data to a database
- The InvoiceService class has the responsibility of sending invoices and reminders to the customer.

andrea.carrara@tum.de; patrick.berggold@tum.de

Open Closed Principle (OCP)

- **A module should be open for extension but closed for modification.**
- Modules should be written so that they can be extended, without requiring them to be modified
- Achieved using **abstraction.**
- Proper abstractions allow for features to be added by adding new code and not changing the original codebase.
- The chances are lower to break code if you do not modify it.

andrea.carrara@tum.de; patrick.berggold@tum.de

Example Open-Closed Principle (OCP)

```
// Class that represents a shape
public abstract class Shape{ public abstract double
Area();}

// Class that represents a rectangle
public class Rectangle : Shape
{
    private double width;
    private double height;

    public Rectangle(double width, double height)
    {   this.width = width;
        this.height = height; }
    public override double Area()
    { return width * height; }
}

// Class that represents a circle
public class Circle : Shape
{
    private double radius;
    public Circle(double radius)
    {   this.radius = radius; }

    public override double Area()
    {return Math.PI * radius * radius; }
}
```

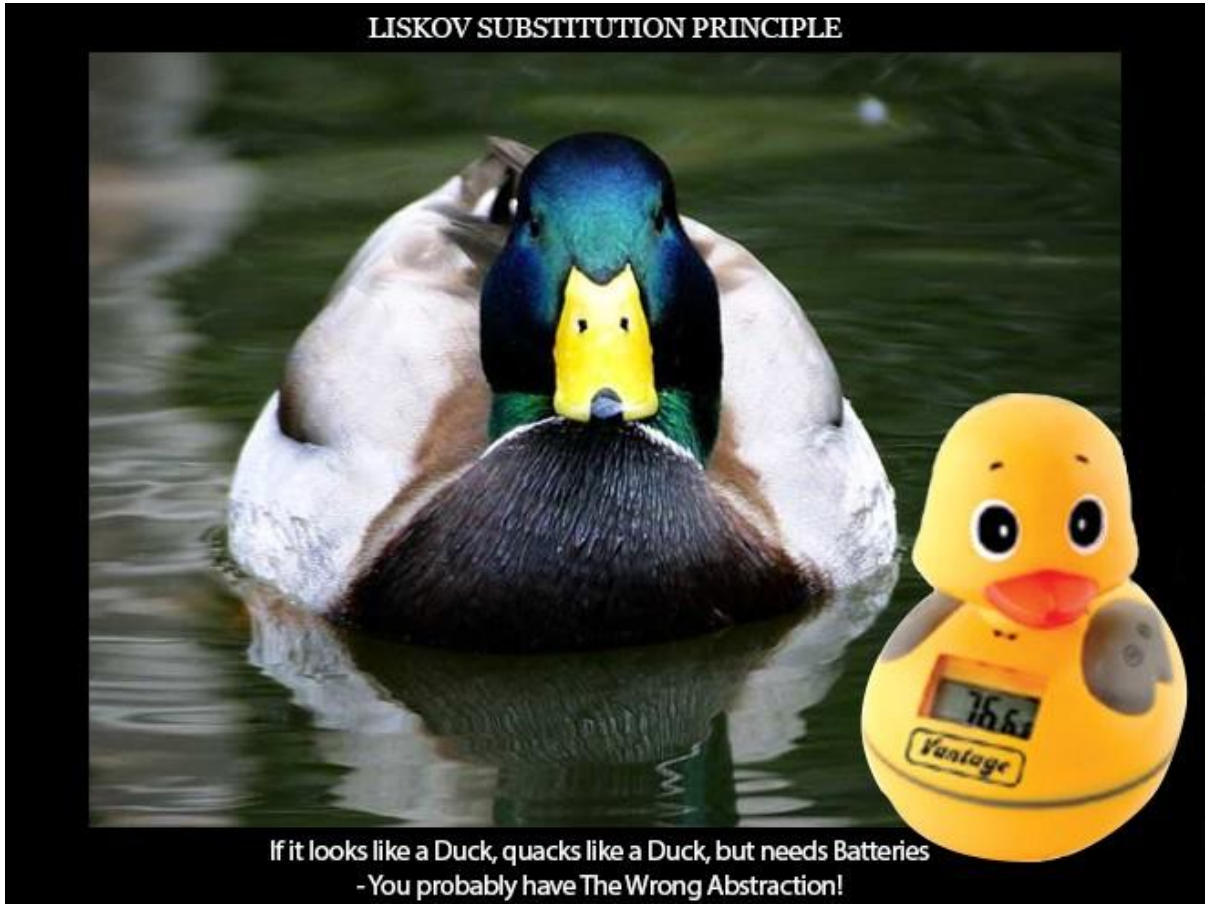
```
public class AreaCalculator
{
    private List<Shape> shapes;

    public AreaCalculator(List<Shape> shapes)
    {
        this.shapes = shapes;
    }

    public double TotalArea()
    {
        double total = 0;
        foreach (Shape shape in shapes)
        {
            total += shape.Area();
        }
        return total;
    }
}
```

- AreaCalculator **is open for extension** because new shapes can be added (such as a square or a triangle) without modifying the AreaCalculator class
- AreaCalculator class **is also closed for modification** because the code was not modified

andrea.carrara@tum.de; patrick.berggold@tum.de

Liskov Substitution Principle (LSP)

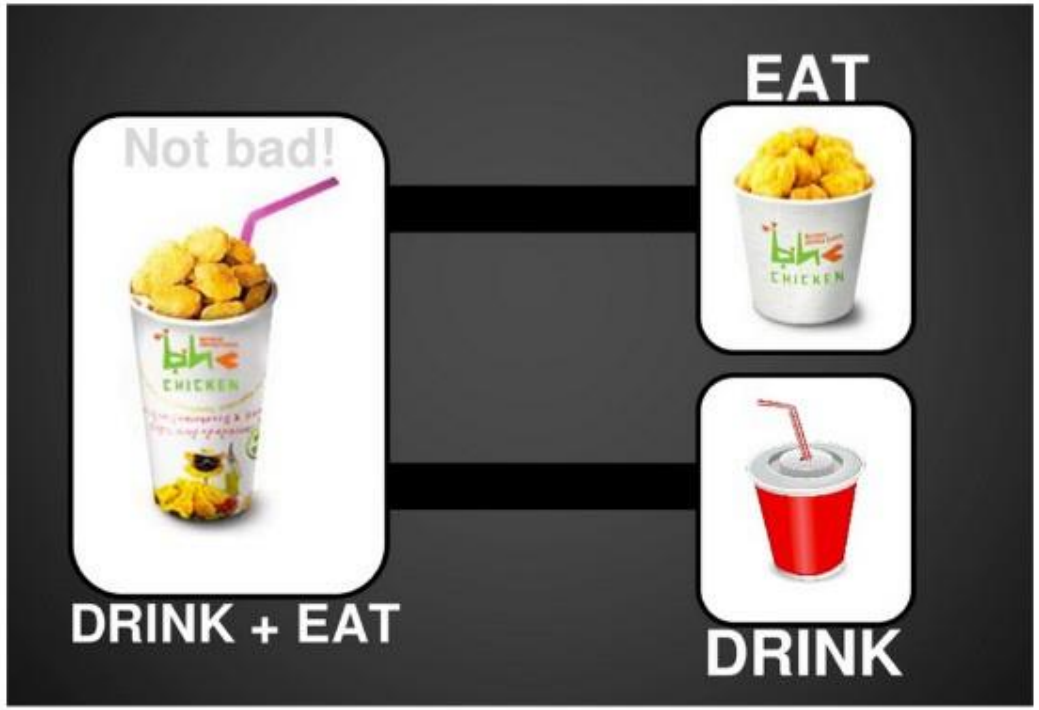
- **Subclasses should be substitutable for their base classes.**
- Subclasses should add to a base class's behavior, not replace it.
- Parent instances should be able to replace their child instances without creating any unexpected or mysterious behavior.



andrea.carrara@tum.de; patrick.berggold@tum.de

Interface Segregation Principle (ISP)

- **Depend upon Abstractions. Do not depend upon concretions.**
- It is better to break down main classes into smaller more specific classes rather than try to maintain one large generalized class.
- the class' clients aren't relying on methods that they don't use.
- The result of implementing this principle, generally speaking, is to have a lot of small, focused interfaces that define only what is needed by their implementations.



andrea.carrara@tum.de; patrick.berggold@tum.de

Example – Interface Segregation Principle

```
public interface IProduct
{
    int ID { get; set; }
    double Weight { get; set; }
    int Stock { get; set; }
    int LegSize { get; set; }
    int WaistSize { get; set; }
}

public class Jeans : IProduct
{
    public int ID { get; set; }
    public double Weight { get; set; }
    public int Stock { get; set; }
    public int LegSize { get; set; }
    public int WaistSize { get; set; }
}
```

Change in requirements →

```
public class BaseballCap : IProduct
{
    public int ID { get; set; }
    public double Weight { get; set; }
    public int Stock { get; set; }
    public int LegSize { get; set; }
    public int WaistSize { get; set; }
    public int HatSize { get; set; }
}
```

Why BaseballCap have LegSize and WaistSize? Solution **Refactor**!

andrea.carrara@tum.de; patrick.berggold@tum.de

Example – Interface Segregation Principle

```
public interface IProduct{
    int ID { get; set; }
    double Weight { get; set; }
    int Stock { get; set; }
}
public interface IPants{
    public int LegSize { get; set; }
    public int WaistSize { get; set; }
}
public interface IHat{
    public int HatSize { get; set; }
}
public class Jeans : IProduct, IPants{
    public int ID { get; set; }
    public double Weight { get; set; }
    public int Stock { get; set; }
    public int Inseam { get; set; }
    public int WaistSize { get; set; }
}
public class BaseballCap : IProduct, IHat{
    public int ID { get; set; }
    public double Weight { get; set; }
    public int Stock { get; set; }
    public int HatSize { get; set; }
}
```

- Each class now has only properties that they need
- ISP can potentially result in a lot of additional interfaces.
- Adhering to this principle allows for much more flexible and modifiable code
- Many interfaces are added that could be used only once

andrea.carrara@tum.de; patrick.berggold@tum.de

DRY – DON’T REPEAT YOURSELF

andrea.carrara@tum.de; patrick.berggold@tum.de

SOLID PRINCIPLES

Five important OO design principles:

- Don‘t Repeat Yourself
- **Every piece of knowledge must have a single, unambiguous, authoritative representation within a system**
- significantly decreases maintenance overhead
- significantly decreases the opportunity for bugs
- considerably decreases code overhead
- may even increase performance (a little bit)

andrea.carrara@tum.de; patrick.berggold@tum.de

Summary Object-Oriented Design

Five important OO design principles:

- The SOLID-Principle is a “checklist” to reflect whether your code is well structured and identifies where you should refactor it
  - **To minimize dependencies (SRP)**
  - **To ensure extensibility (OCP)**
  - **To confirm compatibility of operations along the OOP-Hierarchy (LSP)**
  - **To have manageable and separated Interfaces (ISP)**
  - **To have unidirectional dependencies between Objects (DIP)**

- The DRY-Principle: Well-organized code avoids repetitions

andrea.carrara@tum.de; patrick.berggold@tum.de