

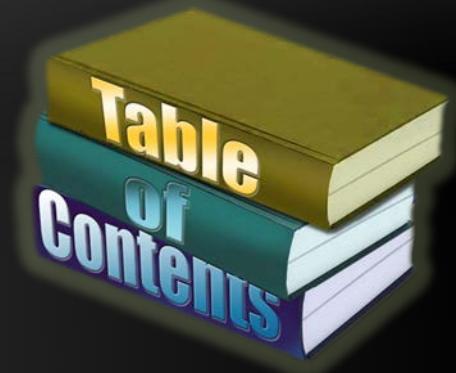


High-Quality Classes and Class Hierarchies

Best Practices in the Object-Oriented Design

Telerik Software Academy
Learning & Development
<http://academy.telerik.com>





1. Basic Principles

- Cohesion, Coupling, Inheritance, Polymorphism

2. High-Quality Classes

- Good Abstraction, Correct Encapsulation, Correct Inheritance
- Class Methods, Constructors, Data
- Good Reasons to Create a Class

3. Typical Mistakes to Avoid in OO Design



Basic Principles

Cohesion, Coupling,
Inheritance and Polymorphism

- ◆ Cohesion measures how closely are all the routines in a class/module
 - ◆ Cohesion must be strong
 - ◆ Classes must contain strongly related functionality and aim for single purpose
- ◆ Strong cohesion is a useful tool for managing complexity
 - ◆ Well-defined abstractions keep cohesion strong
 - ◆ Bad abstractions have weak cohesion

Good and Bad Cohesion

- ◆ Good: hard disk, CD-ROM, floppy



- ◆ Bad: spaghetti code



- ◆ Strong cohesion example

- ◆ Class System.Math

- ◆ Sin(), Cos(), Asin()
 - ◆ Sqrt(), Pow(), Exp()
 - ◆ Math.PI, Math.E

```
double sideA = 40, sideB = 69;  
double angleAB = Math.PI / 3;  
double sideC =  
    Math.Pow(sideA, 2) + Math.Pow(sideB, 2)  
    - 2 * sideA * sideB * Math.Cos(angleAB);  
double sidesSqrtSum = Math.Sqrt(sideA) +  
    Math.Sqrt(sideB) + Math.Sqrt(sideC);
```



- ◆ Coupling describes how tightly a class or routine is related to other classes or routines
- ◆ Coupling must be kept loose
 - ◆ Modules must depend little on each other
 - ◆ All classes and routines must have
 - ◆ Small, direct, visible, and flexible relationships to other classes and routines
 - ◆ One module must be easily used by other modules, without complex dependencies

Loose and Tight Coupling

- ◆ Loose Coupling

- ◆ Easily replace old HDD
- ◆ Easily place this HDD to another motherboard

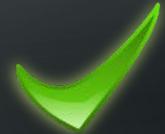


- ◆ Tight Coupling

- ◆ Where is the video adapter?
- ◆ Can you change the video controller on this MB?



Loose Coupling – Example



```
class Report
{
    public bool LoadFromFile(string fileName) {...}
    public bool SaveToFile(string fileName) {...}
}

class Printer
{
    public static int Print(Report report) {...}
}

class Program
{
    static void Main()
    {
        Report myReport = new Report();
        myReport.LoadFromFile("C:\\DailyReport.rep");
        Printer.Print(myReport);
    }
}
```

Tight Coupling – Example



```
class MathParams
{
    public static double operand;
    public static double result;
}

class MathUtil
{
    public static void Sqrt()
    {
        MathParams.result = CalcSqrt(MathParamsoperand);
    }
}

MathParamsoperand = 64;
MathUtil.Sqrt();
Console.WriteLine(MathParams.result);
```

- ◆ Inheritance is the ability of a class to implicitly gain all members from another class
 - ◆ Inheritance is principal concept in OOP
 - ◆ The class whose methods are inherited is called base (parent) class
 - ◆ The class that gains new functionality is called derived (child) class
- ◆ Use inheritance to:
 - ◆ Reuse repeating code: data and program logic
 - ◆ Simplify code maintenance

Inheritance in C# and JS

- ◆ In C# all class members are inherited
 - ◆ Fields, methods, properties, etc.
 - ◆ Structures cannot be inherited, only classes
 - ◆ No multiple inheritance is supported
 - ◆ Only multiple interface implementations
- ◆ In JS inheritance is supported indirectly
 - ◆ Several ways to implement inheritance
 - ◆ Multiple inheritance is not supported
 - ◆ There are no interfaces (JS is typeless language)

Polymorphism

- ◆ Polymorphism is a principal concept in OOP
- ◆ The ability to handle the objects of a specific class as instances of its parent class
 - To call abstract functionality
- ◆ Polymorphism allows to create hierarchies with more valuable logical structure
- ◆ Polymorphism is a tool to enable code reuse
 - Common logic is taken to the base class
 - Specific logic is implemented in the derived class in a overridden method

Polymorphism in C# and JS

- ◆ In C# polymorphism is implemented through:
 - Virtual methods (`virtual`)
 - Abstract methods (`abstract`)
 - Interfaces methods (`interface`)
- ◆ In C# `override` overrides a virtual method
- ◆ In JavaScript all methods are virtual
 - The child class can just "override" any method from any object
 - There are no interfaces (JS is typeless language)

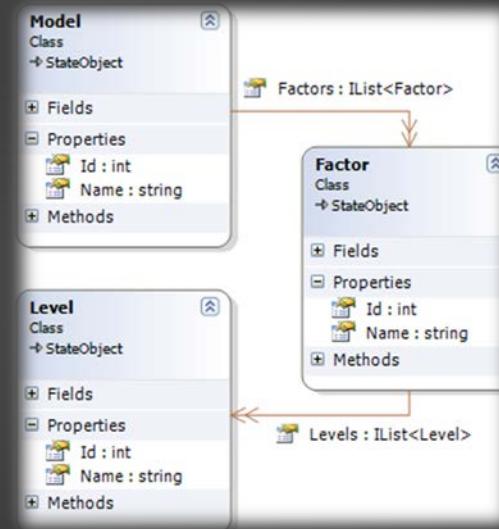
Polymorphism – Example

```
class Person
{
    public virtual void PrintName()
    {
        Console.WriteLine("I am a person.");
    }
}

class Trainer : Person
{
    public override void PrintName()
    {
        Console.WriteLine(
            "I am a trainer. " + base.PrintName());
    }
}

class Student : Person
{
    public override void PrintName()
    {
        Console.WriteLine("I am a student.");
    }
}
```





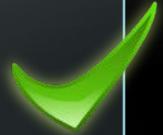
High-Quality Classes

How to Design High-Quality Classes? Abstraction, Cohesion and Coupling

High-Quality Classes: Abstraction

- ◆ Present a consistent level of abstraction in the class contract (publicly visible members)
 - What abstraction the class is implementing?
 - Does it represent only one thing?
 - Does the class name well describe its purpose?
 - Does the class define clear and easy to understand public interface?
 - Does the class hide all its implementation details?

Good Abstraction – Example



```
public class Font
{
    public string Name { get; set; }
    public float SizeInPoints { get; set; }
    public FontStyle Style { get; set; }
    public Font(
        string name, float sizeInPoints, FontStyle style)
    {
        this.Name = name;
        this.SizeInPoints = sizeInPoints;
        this.Style = style;
    }
    public void DrawString(DrawingSurface surface,
        string str, int x, int y) { ... }
    public Size MeasureString(string str) { ... }
}
```

Bad Abstraction – Example

```
public class Program  
{  
    public string title;  
    public int size;  
    public Color color;  
    public void InitializeCommandStack();  
    public void PushCommand(Command command);  
    public Command PopCommand();  
    public void ShutdownCommandStack();  
    public void InitializeReportFormatting();  
    public void FormatReport(Report report);  
    public void PrintReport(Report report);  
    public void InitializeGlobalData();  
    public void ShutdownGlobalData();  
}
```

Does this class really
represents a "program"?
Is this name good?



Does this class
really have a
single purpose?

Establishing Good Abstraction

- ◆ Define operations along with their opposites, e.g.
 - ◆ Open() and Close()
- ◆ Move unrelated methods in another class, e.g.
 - ◆ In class Employee if you need to calculate Age by given DateOfBirth
 - ◆ Create a static method CalcAgeByBirthDate(...) in a separate class DateUtils
- ◆ Group related methods into a single class
- ◆ Does the class name correspond to the class content?

- ◆ Beware of breaking the interface abstraction due to evolution
 - ◆ Don't add public members inconsistent with abstraction
 - ◆ Example: in class called Employee at some time we add method for accessing the DB with SQL

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    ...
    public SqlCommand FindByPrimaryKeySqlCommand(int id);
}
```



- ◆ Minimize visibility of classes and members
 - ◆ In C# start from **private** and move to **internal**, **protected** and **public** if required
- ◆ Classes should hide their implementation details
 - ◆ A principle called **encapsulation** in OOP
 - ◆ Anything which is not part of the class interface should be declared **private**
 - ◆ Classes with good encapsulated classes are: less complex, easier to maintain, more loosely coupled
- ◆ Classes should keep their state clean → throw an exception if invalid data is being assigned

Encapsulation (2)

- ◆ Never declare fields public (except constants)
 - ◆ Use properties / methods to access the fields
- ◆ Don't put private implementation details in the public interface
 - ◆ All public members should be consistent with the abstraction represented by the class
- ◆ Don't make a method public just because it calls only public methods
- ◆ Don't make assumptions about how the class will be used or will not be used

Encapsulation (3)

- ◆ Don't violate encapsulation semantically!
 - ◆ Don't rely on non-documented internal behavior or side effects
 - ◆ Wrong example:
 - ◆ Skip calling `ConnectToDB()` because you just called `FindEmployeeById()` which should open connection
 - ◆ Another wrong example:
 - ◆ Use `String.Empty` instead of `Titles.NoTitle` because you know both values are the same



Inheritance or Containment?

- ◆ Containment is "has a" relationship
 - ◆ Example: Keyboard has a set of Keys
- ◆ Inheritance is "is a" relationship
 - ◆ Design for inheritance: make the class abstract
 - ◆ Disallow inheritance: make the class sealed
 - ◆ Subclasses must be usable through the base class interface
 - ◆ Without the need for the user to know the difference

- ◆ Don't hide methods in a subclass
 - ◆ Example: if the class Timer has private method Start(), don't define Start() in AtomTimer
- ◆ Move common interfaces, data, and behavior as high as possible in the inheritance tree
 - ◆ This maximizes the code reuse
- ◆ Be suspicious of base classes of which there is only one derived class
 - ◆ Do you really need this additional level of inheritance?

- ◆ Be suspicious of classes that override a routine and do nothing inside
 - ◆ Is the overridden routine used correctly?
- ◆ Avoid deep inheritance trees
 - ◆ Don't create more than 6 levels of inheritance
- ◆ Avoid using a base class's protected data fields in a derived class
 - ◆ Provide protected accessor methods or properties instead

- ◆ Prefer inheritance to extensive type checking:

```
switch (shape.Type)
{
    case Shape.Circle:
        shape.DrawCircle();
        break;
    case Shape.Square:
        shape.DrawSquare();
        break;
    ...
}
```



- ◆ Consider inheriting Circle and Square from Shape and override the abstract action Draw()

Class Methods and Data

- ◆ Keep the number of methods in a class as small as possible → reduce complexity
- ◆ Minimize direct methods calls to other classes
 - ◆ Minimize indirect methods calls to other classes
 - ◆ Less external method calls == less coupling
 - ◆ Also known as "fan-out"
- ◆ Minimize the extent to which a class collaborates with other classes
 - ◆ Reduce the coupling between classes

- ◆ Initialize all member data in all constructors, if possible
 - ◆ Uninitialized data is error prone
 - ◆ Partially initialized data is even more evil
 - ◆ Incorrect example: assign FirstName in class Person but leave LastName empty
- ◆ Initialize data members in the same order in which they are declared
- ◆ Prefer deep copies to shallow copies (ICloneable should make deep copy)

Use Design Patterns

- ◆ Use private constructor to prohibit direct class instantiation
- ◆ Use design patterns for common design situations
 - ◆ Creational patterns like Singleton, Factory Method, Abstract Factory
 - ◆ Structural patterns like Adapter, Bridge, Composite, Decorator, Façade
 - ◆ Behavioral patterns like Command, Iterator, Observer, Strategy, Template Method

Top Reasons to Create Class

- ◆ Model real-world objects with OOP classes
- ◆ Model abstract objects, processes, etc.
- ◆ Reduce complexity
 - ◆ Work at higher level
- ◆ Isolate complexity
 - ◆ Hide it in a class
- ◆ Hide implementation details → encapsulation
- ◆ Limit effects of changes
 - ◆ Changes affect only their class

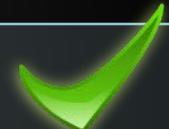
Top Reasons to Create Class (2)

- ◆ Hide global data
 - ◆ Work through methods
- ◆ Group variables that are used together
- ◆ Make central points of control
 - ◆ Single task should be done at single place
 - ◆ Avoid duplicating code
- ◆ Facilitate code reuse
 - ◆ Use class hierarchies and virtual methods
- ◆ Package related operations together

- ◆ Group related classes together in namespaces
- ◆ Follow consistent naming convention

```
namespace Utils
```

```
{  
    class MathUtils { ... }  
    class StringUtils { ... }  
}
```



```
namespace DataAccessLayer
```

```
{  
    class GenericDAO<Key, Entity> { ... }  
    class EmployeeDAO<int, Employee> { ... }  
    class AddressDAO<int, Address> { ... }  
}
```



Typical Mistakes to Avoid

Lessons Learned from the
OOP Exam at Telerik Software Academy

Plural Used for a Class Name

- ◆ Never use plural in class names unless they hold some kind of collection!
- ◆ Bad example:

Singular: Teacher (a single teacher, not several).

```
public class Teachers : ITeacher
{
    public string Name { get; set; }
    public List<ICourse> Courses { get; set; }
}
```



- ◆ Good example:

```
public class GameFieldConstants
{
    public const int MIN_X = 100;
    public const int MAX_X = 700;
}
```



Throwing an Exception without Parameters

- ◆ Don't throw exception without parameters:

```
public ICourse CreateCourse(string name, string town)
{
    if (name == null)
    {
        throw new ArgumentNullException();
    }
    if (town == null)
    {
        throw new ArgumentNullException();
    }
    return new Course(name, town);
}
```



Which parameter is null here?

- ◆ Check for invalid data in the setters and constructors, not in the getter!

```
public string Town
{
    get
    {
        if (string.IsNullOrWhiteSpace(this.town))
            throw new ArgumentNullException();
        return this.town;
    }
    set
    {
        this.town = value;
    }
}
```

Put this check
in the setter!



Missing this for Local Members

- Always use `this.XXX` instead of `XXX` to access members within the class:

```
public class Course
{
    public string Name { get; set; }

    public Course(string name)
    {
        Name = name;
    }
}
```



Use `this.Name`

Empty String for Missing Values

- ◆ Use null when a value is missing, not 0 or ""
 - ◆ Make a field / property nullable to access null values or just disallow missing values
- ◆ Bad example:

```
Teacher teacher = new Teacher("");
```

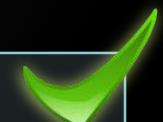


Empty name is very
bad idea! Use null.

```
Teacher teacher = new Teacher();
```



```
Teacher teacher = new Teacher(null);
```



Magic Numbers in the Classes

- ◆ Don't use "magic" numbers
 - ◆ Especially when the class has members related to those numbers:

```
public class Wolf : Animal
{
    ...
    bool TryEatAnimal(Animal animal)
    {
        if (animal.Size <= 4)
        {
            return true;
        }
    }
}
```



This if statement is very wrong.
4 is the size of the Wolf, which
has a Size property inherited
from Animal. Why not use
`this.Size` instead of 4?

Base Constructor Not Called

- ◆ Call the base constructor to reuse the object's state initialization code:

```
public class Course
{
    public string Name { get; set; }
    public Course(string name)
    { this.Name = name; }
}

public class LocalCourse
{
    public string Lab { get; set; }
    public Course(string name, string lab)
    {
        this.Name = name;
        this.Lab = lab;
    }
}
```



: base (name)

Call the base
constructor instead!

Repeating Code in the Base and Child Classes

- Never copy-paste the code of the base in the inherited class



```
public class Course
{
    public string Name { get; set; }
    public ITeacher Teacher { get; set; }
}

public class LocalCourse
{
    public string Name { get; set; }
    public ITeacher Teacher { get; set; }
    public string Lab { get; set; }
}
```

Why are these fields duplicated and not inherited?

Broken Encapsulation through a Parameterless Constructor

- ◆ Be careful to keep fields well encapsulated

```
public class Course
{
    public string Name { get; private set; }
    public ITeacher Teacher { get; private set; }

    public Course(string name, ITeacher teacher)
    {
        if (name == null)
            throw ArgumentNullException("name");
        if (teacher == null)
            throw ArgumentNullException("teacher");
        this.Name = name;
        this.Teacher = teacher;
    }

    public Course() { }
```



Breaks encapsulation:
Name & Teacher can
be left null.

Coupling the Base Class with Its Child Classes

- ◆ Base class should never know about its children!

```
public class Course
{
    public override string ToString()
    {
        StringBuilder result = new StringBuilder();
        ...
        if (this is ILocalCourse)
        {
            result.Append("Lab = " + ((ILocalCourse)this).Lab);
        }
        if (this is IOffsiteCourse)
        {
            result.Append("Town = " + ((IOffsiteCourse)this).Town);
        }
        return result.ToString();
    }
}
```



Hidden Interpretation of Base Class as Its Specific Child Class

- ◆ Don't define `IEnumerable<T>` fields and use them as `List<T>` (broken abstraction)

```
public class Container<T>
{
    public IEnumerable<T> Items { get; }

    public Container()
    {
        this.Items = new List<T>();
    }

    public void AddItem (T item)
    {
        (this.Items as List<T>).Add(item);
    }
}
```



Bad practice:
hidden `List<T>`

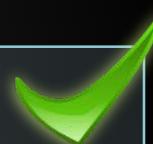
Hidden Interpretation of Base Class as Its Specific Child Class (2)

- ◆ Use `List<T>` in the field and return it where `IEnumerable<T>` is required:

```
public class Container<T>
{
    private List<T> items = new List<T>();

    public IEnumerable<T> Items
    {
        get { return this.items; }
    }

    public void AddItem (T item)
    {
        this.Items.Add(item);
    }
}
```



This partially breaks encapsulation. Think about cloning to keep your items safe.

Repeating Code Not Moved Upper in the Class Hierarchy

```
public abstract class Course : ICourse
{
    public string Name { get; set; }
    public ITeacher Teacher { get; set; }
}

public class LocalCourse : Course, ILocalCourse
{
    public string Lab { get; set; }
    public override string ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.Append(this.GetType().Name);
        sb.AppendFormat("(Name={0})", this.Name);
        if (!(this.Teacher == null))
            sb.AppendFormat("; Teacher={0}", this.Teacher.Name);
        sb.AppendFormat("; Lab={0})", this.Lab);
        return sb.ToString();
    }
}
```

// continues at the next slide



Repeating code

Repeating Code Not Moved Upper in the Class Hierarchy (2)

```
public class OffsiteCourse : Course, ILocalCourse
{
    public string Town { get; set; }

    public override string ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.Append(this.GetType().Name);
        sb.AppendFormat("(Name={0})", this.Name);
        if (!(this.Teacher == null))
            sb.AppendFormat("; Teacher={0}", this.Teacher.Name);
        sb.AppendFormat("; Town={0})", this.Town);
        return sb.ToString();
    }
}
```

Repeating code



- When overriding methods, call the base method if you need its functionality, don't copy-paste it!

Move the Repeating Code in Upper in the Class Hierarchy

```
public abstract class Course : ICourse
{
    public string Name { get; set; }
    public ITeacher Teacher { get; set; }

    public override string ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.Append(this.GetType().Name);
        sb.AppendFormat("(Name={0})", this.Name);
        if (!(this.Teacher == null))
            sb.AppendFormat("; Teacher={0}", this.Teacher.Name);
        return sb.ToString();
    }
}
```

// continues at the next slide



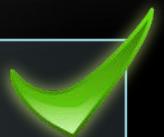
Move the Repeating Code in Upper in the Class Hierarchy (2)

```
public class LocalCourse : Course, ILocalCourse
{
    public string Lab { get; set; }

    public override string ToString()
    {
        return base.ToString() + "; Lab=" + this.Lab + ")";
    }
}

public class OffsiteCourse : Course, ILocalCourse
{
    public string Town { get; set; }

    public override string ToString()
    {
        return base.ToString() + "; Town=" + this.Town + ")";
    }
}
```



High-Quality Classes and Class Hierarchies

Questions?

- ◆ Use the provided source code "High-Quality-Classes-Homework.zip".
 1. Take the VS solution "Abstraction" and refactor its code to provide good abstraction. Move the properties and methods from the class Figure to their correct place. Move the common methods to the base class's interface. Remove all duplicated code (properties / methods / other code).
 2. Establish good encapsulation in the classes from the VS solution "Abstraction". Ensure that incorrect values cannot be assigned in the internal state of the classes.

Homework (2)

3. Take the VS solution "Cohesion-and-Coupling" and refactor its code to follow the principles of good abstraction, loose coupling and strong cohesion. Split the class `Utils` to other classes that have strong cohesion and are loosely coupled internally.
4. Redesign the classes and refactor the code from the solution "Inheritance-and-Polymorphism" to follow the best practices in high-quality classes. Extract abstract base class and move all common properties in it. Encapsulate the fields and make sure required fields are not left without a value. Reuse the repeating code though base methods.

Free Trainings @ Telerik Academy

- ◆ C# Programming @ Telerik Academy

- ◆ csharpfundamentals.telerik.com



- ◆ Telerik Software Academy

- ◆ academy.telerik.com



- ◆ Telerik Academy @ Facebook

- ◆ facebook.com/TelerikAcademy



- ◆ Telerik Software Academy Forums

- ◆ forums.academy.telerik.com

