High-Quality Programming Code

Code Correctness, Readability, Maintainability, Testability, Refactoring

QUALITY
CODE

SoftUni Team

Technical Trainers

Software
University

SoftUni

Software University

https://softuni.bg

Table of Contents

1. What is High-Quality Code?
2. Code Conventions
3. Managing Complexity
4. Characteristics of Quality Code
5. Refactoring Principles, Patterns and Levels

High-Quality Programming Code

What is It?

Why Quality is Important?

- What does this code do? Is it correct?

```
static void Main()
{
  int value=010, i=5, w;
  switch(value){case 10:w=5;Console.WriteLine(w);break;case
9:i=0;break;
          case 8:Console.WriteLine("8 ");break;
     default:Console.WriteLine("def ");{
               Console.WriteLine("hoho ");  }
     for (int k = 0; k < i; k++, Console.WriteLine(k -
'f'));break;} { Console.WriteLine("loop!"); }
}
```

Why Quality is Important? (2)

- Now the code is formatted, but is still unclear

```
int value = 010, i = 5, w;
switch (value)
{
  case 10: w = 5; Console.WriteLine(w); break;
  case 9: i = 0; break;
  case 8: Console.WriteLine("8 "); break;
  default:
    Console.WriteLine("def ");
    Console.WriteLine("hoho ");
    for (int k = 0; k < i; k++,
     Console.WriteLine(k - 'f')) ;
    break;
}
Console.WriteLine("loop!");
```

Software Quality

- External quality
  - Does the software behave **correctly**?
  - Are the produced **results correct**?
  - Does the software run **fast**?
  - Is the software UI **easy-to-use**?
  - Is the code **secure** enough?

- Internal quality
  - Is the code **easy** to **read** and understand?
  - Is the code **well structured**?
  - Is the code **easy** to **modify**?

What is High-Quality Programming Code?

- High-quality programming code:
  - Easy to **read** and understand
    - Easy to modify and **maintain**
  - Correct **behavior** in all cases
    - Well **tested**
  - Well architectured and **designed**
  - Well **documented**
    - Self-documenting code
  - Well **formatted**

What is High-Quality Programming Code? (2)

- High-quality programming code:
  - Strong **cohesion** at all levels: modules, classes, methods, etc.
    - Single **unit** is responsible for single **task**
  - Loose **coupling** between modules, classes, methods, etc.
    - Units are **independent** one of another
  - Good **formatting**
  - Good **names** for classes, methods, variables, etc.
  - **Self-documenting** code style

Code Conventions

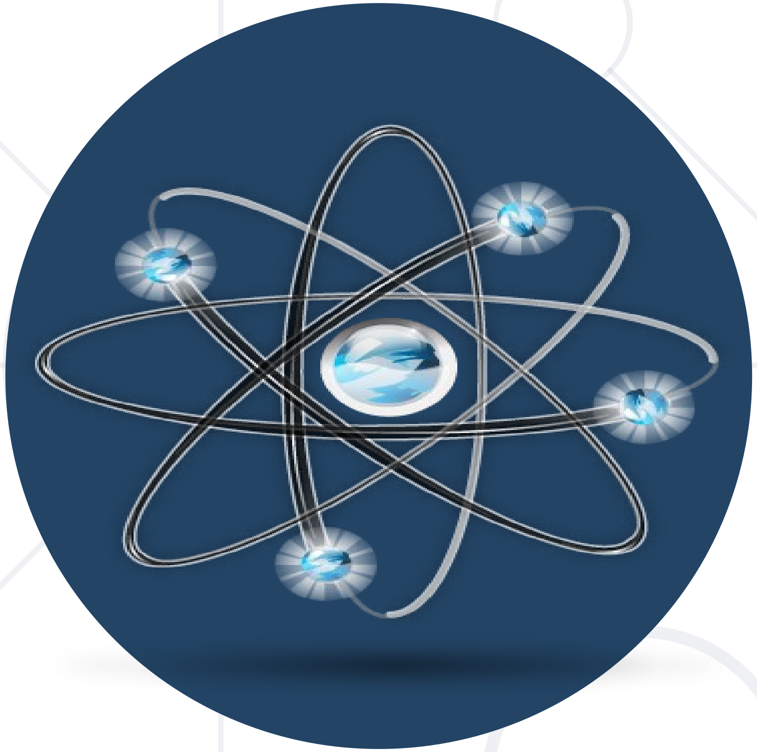
Code Formatting and Naming Conventions

Code Conventions

- Code conventions are formal guidelines about the style of the source code:
  - Code formatting conventions
    - Indentation, whitespace, etc.
  - Naming conventions
    - **PascalCase** or **camelCase**, **prefixes**, **suffixes**, etc.

Code Conventions (2)

- Best practices
  - Using C# language features the right way - classes, interfaces, enumerations, structures, inheritance, exceptions, properties, events, constructors, fields, operators, etc.
- Microsoft official C# code conventions
  - Design Guidelines for Developing Class Libraries: http://msdn.microsoft.com/en-us/library/ms229042.aspx



Managing Complexity

Maximizing Program Effectiveness

Managing Complexity

- **Managing complexity** has a central role in software construction
  - Minimize the amount of complexity that anyone's brain has to deal with at certain time
- Architecture and design challenges
  - Design modules and classes to **reduce complexity**
- Code construction challenges
  - Apply **good software construction practices**: classes, methods, variables, naming, statements, error handling, formatting, comments, unit testing, etc.

Managing Complexity (2)

- Key to being an **effective programmer**:
  - Maximizing the portion of a program that you can safely ignore
    - While working on any one section of code
  - Most practices discussed later propose ways to **achieve** this **important goal**

Code Quality

Key Characteristics

Key Characteristics of High-Quality Code

- **Correct behavior**
  - Conforming to the **requirements**
  - **Stable**, no hangs, no crashes
  - **Bug free** – works as expected
  - Correct **response** to incorrect usage
- **Readable** – easy to read
- **Understandable** – self-documenting
- **Maintainable** – easy to modify when needed

Key Characteristics of High-Quality Code (2)

- Good **identifiers** names
  - Good names for variables, constants, methods, parameters, classes, structures, fields, properties, interfaces, structures, enumerations, namespaces,
- High-quality classes, interfaces and class hierarchies
  - Good **abstraction** and **encapsulation**
  - Correct use of **inheritance** and **polymorphism**
  - **Simplicity**, reusability, minimal complexity
  - Strong **cohesion**, loose **coupling**

Examples

Tight Coupling – Example

```
class MathParams {
    public static double operand;
    public static double result;
}
class MathUtil {
    public static void Sqrt() {
        MathParams.result = CalcSqrt(MathParams.operand);
    }
}
MathParams.Operand = 64;
MathUtil.Sqrt();
Console.WriteLine(MathParams.result);
```

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);
    }
}
```

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();
}
```

Is this name good?

Does this class really have a single purpose?

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) { … }
}
```

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();
    }
}
```

Extract the Repeating Code into Class

```
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 on the next slide
```

Extract the Repeating Code into Class (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 + ")";
    }
}
```

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**

Key Characteristics of High-Quality Code (3)

- High-quality methods
  - Reduced complexity, improved readability
  - Good method **names** and parameter names
  - Strong cohesion, loose coupling
- Variables, data, expressions and constants
  - **Minimal** variable **scope**, span, live time
  - Simple **expressions**
  - Correctly used **constants**
  - Correctly organized **data**

Acceptable Types of Cohesion

- **Functional cohesion** (independent function)
  - Method performs certain well-defined calculation and returns a single result
  - The entire input is passed through parameters and the entire output is returned as result
  - No external dependencies or side effects

```
Math.Sqrt(value) → square root
```

```
char.IsLetterOrDigit(ch)
```

```
string.Substring(str, startIndex, length)
```

Acceptable Types of Cohesion (2)

- **Sequential cohesion** (algorithm)
  - Method performs certain sequence of operations to perform a single task and achieve certain result
    - It encapsulates an algorithm
  - Example:

```
SendEmail(recipient, subject, body)
```

    - Connect to mail server
    - Send message headers
    - Send message body
    - Disconnect from the server

Acceptable Types of Cohesion (3)

- **Communicational cohesion** (common data)
  - A set of operations used to process certain data and produce a result
  - Example:

```
DisplayAnnualExpensesReport(int employeeId)
```

    - Retrieve input data from database
    - Perform internal calculations over retrieved data
    - Build the report
    - Format the report as Excel worksheet
    - Display the Excel worksheet on the screen

Acceptable Types of Cohesion (4)

- Temporal cohesion (time related activities)
  - Operations that are generally not related but need to happen in a certain moment
  - Examples:

```
InitializeApplication()
```

    - Load user settings
    - Check for updates
    - Load all invoices from the database

```
ButtonConfirmClick()
```

    - Sequence of actions to handle the event

Unacceptable Cohesion

- Logical cohesion
  - Performs a different operation depending on an input parameter
  - Incorrect example:

```
object ReadAll(int operationCode)
{
    if (operationCode == 1) … // Read person name
    else if (operationCode == 2) … // Read address
    else if (operationCode == 3) … // Read date
    …
}
```

  - Can be acceptable in event handlers
    - E.g. the **KeyDown** event in Windows Forms)

Unacceptable Cohesion

- Coincidental cohesion (spaghetti)
  - Not related (random) operations grouped in a method for unclear reason
  - Incorrect example:

```
HandleStuff(customerId, int[], ref sqrtValue, mp3FileName, emailAddress)
```

    - Prepares annual incomes report for given customer
    - Sorts an array of integers in increasing order
    - Calculates the square root of given number
    - Converts given MP3 file into WMA format
    - Sends email to given customer

Loose Coupling

- What is **loose coupling**?
  - **Minimal dependences** of the method on the other parts of the source code
  - Minimal dependences on the class members or external classes and their members
  - No side effects
  - If the coupling is loose, we can easily reuse a method or group of methods in a new project
- Tight coupling → **spaghetti code**

Loose Coupling (2)

- The **ideal coupling**
  - A methods depends only on its parameters
  - Does not have any other input or output
  - Example: **Math.Sqrt()**
- Real world
  - Complex software cannot avoid coupling but could make it as loose as possible
  - Example: complex encryption algorithm performs initialization, encryption, finalization

Coupling – Example

- Intentionally increased coupling for more flexibility (.NET cryptography API):

```
byte[] EncryptAES(byte[] inputData, byte[] secretKey) {
    Rijndael cryptoAlg = new RijndaelManaged();
    cryptoAlg.Key = secretKey;
    cryptoAlg.GenerateIV();
    MemoryStream destStream = new MemoryStream();
    CryptoStream csEncryptor = new CryptoStream(
        destStream, cryptoAlg.CreateEncryptor(),
        CryptoStreamMode.Write);
    csEncryptor.Write(inputData, 0, inputData.Length);
    csEncryptor.FlushFinalBlock();
    return destStream.ToArray();
}
```

Loose Coupling – Example

- To reduce coupling we can make **utility classes**
  - Hide the complex logic and provide simple straightforward interface (a.k.a. façade):

```
byte[] EncryptAES(byte[] inputData, byte[] secretKey)
{
    MemoryStream inputStream = new MemoryStream(inputData);
    MemoryStream outputStream = new MemoryStream();
    EncryptionUtils.EncryptAES(
       inputStream, outputStream, secretKey);
    byte[] encryptedData = outputStream.ToArray();
    return encryptedData;
}
```

Tight Coupling – Example

- Passing parameters through class fields
  - Typical example of tight coupling
  - Don't do this unless you have a good reason!

```
class Sumator {
    public int a, b;
    int Sum()
        return a + b;
    static void Main() {
        Sumator sumator = new Sumator() { a = 3, b = 5 };
        Console.WriteLine(sumator.Sum());
    }
}
```

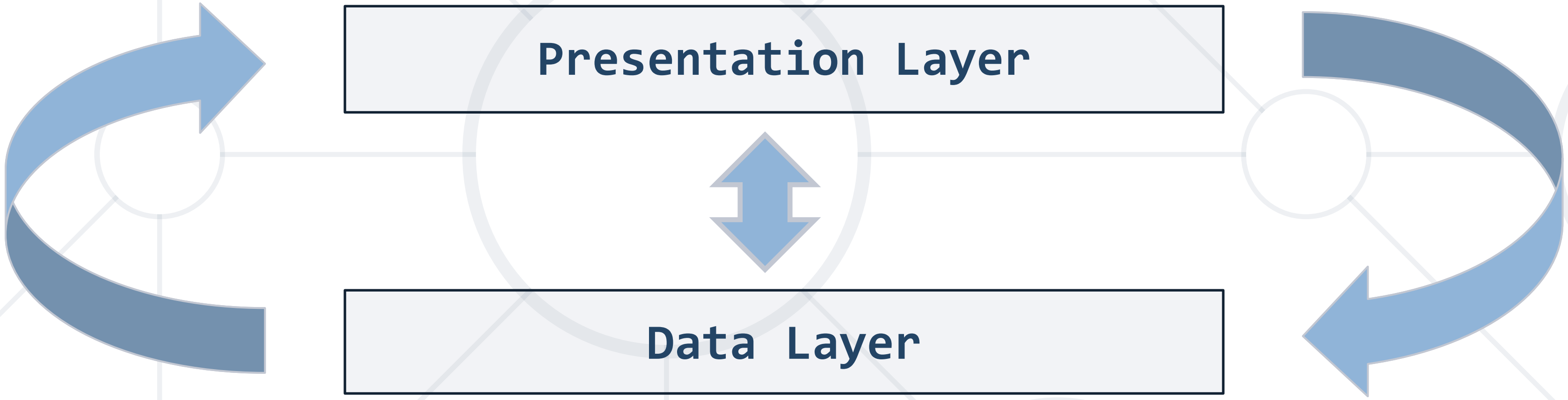
Tight Coupling in Real World Code

- Say, we have a large piece of software
  - We need to update subsystems and the subsystems are not really independent
  - E.g. a change in filtering affects sorting, etc:

```
class GlobalManager
{
    public void UpdateSorting() {…}
    public void UpdateFiltering() {…}
    public void UpdateData() {…}
    public void UpdateAll () {…}
}
```

Cohesion Problems in Real-World Code

- Say, we have an application consisting of two layers:



- Do not update top-down and bottom-up from a single method!
  - E.g. **RemoveCustomer()** method in the **DataLayer** changes also the presentation layer
  - Better use a notification (observer pattern / event)

Pass Entire Object or Its Fields?

- When should we pass an object containing few values and when these values separately?
  - Sometime we pass an object and use only a single field of it
  - Is this a good practice?
  - Examples:

```
CalculateSalary(Employee employee, int months);
```

```
CalculateSalary(double rate, int months);
```

  - Look at the method's level of abstraction
    - Is it intended to operate with employees of with rates and months? → the first is incorrect

How Many Parameters a Method Should Have?

- Limit the number of parameters to **7 (+/-2)**
  - 7 is a "magic" number in psychology
  - Human brain cannot process more than 7 (+/-2) things in the same time
- If the parameters need to be too many, reconsider the method's intent
  - Does it have a clear intent?
  - Consider extracting few of the parameters in a new class

Key Characteristics of High-Quality Code (4)

- Correctly used **control structures** (**if**, **switch**)
  - Simple **statements**
  - Simple **conditional** statements and simple conditions
  - Well organized **loops** without deep nesting
- Good code formatting
  - Reflecting the **logical structure** of the program
  - Good formatting of classes, methods, blocks, whitespace, long lines, alignment, etc.

Method Length

- How long should a method be?
  - There is no specific restriction
  - Avoid methods longer **than one screen** (**30 lines**)
  - Long methods are not always bad
    - Be sure you have a good reason for their length
  - **Cohesion** and **coupling** are more important than the method length!
  - Long methods often contain portions that could be extracted as separate methods with good names and clear intent

Key Characteristics of High-Quality Code (5)

- High-quality documentation and comments
  - Effective **comments**
  - **Self-documenting** code
- **Defensive** programming and exceptions
  - Ubiquitous use of defensive programming
  - Well **organized** exception handling
- Code tuning and optimization
  - Quality code instead of good performance
  - Code performance when required

Key Characteristics of High-Quality Code (6)

- Following the corporate code conventions
  - **Formatting** and **style**, **naming**, etc.
  - Domain-specific best practices
- Well tested and reviewed
  - Testable code
  - Well designed **unit tests**
    - Tests for all scenarios
    - High code coverage
  - Passed code reviews and inspections

Refactoring

What is It?

What is Refactoring?

> **Refactoring means "to improve the design and quality of existing source code without changing its external behavior".**
>
> ***Martin Fowler***

- A step by step process that turns the bad code into good code
  - Based on "refactoring patterns" → well-known recipes for improving the code

Code Refactoring

- What is **refactoring** of the source code?
  - Improving the design and quality of existing source code without changing its behavior
  - Step by step process that turns the bad code into good code (if possible)
- **Why** we need refactoring?
  - Code constantly changes and its quality constantly degrades (unless refactored)
  - Requirements often change and code needs to be changed to follow them

When to Refactor?

- **Bad smells in the code** indicate need of refactoring
- Refactor:
  - To make adding a new function easier
  - As part of the process of fixing bugs
  - When reviewing someone else's code
  - Have technical debt (or any problematic code)
  - When doing test-driven development
- **Unit tests** guarantee that refactoring does not change the behavior
  - If there are no unit tests, write them

Refactoring: Main Principles

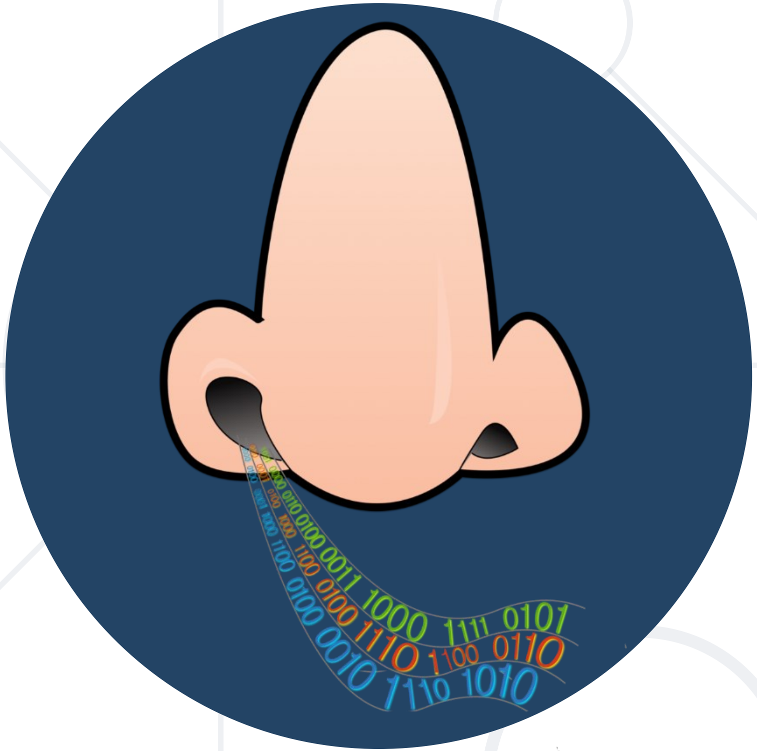
- Keep it simple (**KISS** principle)
- Avoid duplication (**DRY** principle)
- Make it expressive (self-documenting, comments, etc.)
- Reduce overall code (**KISS** principle)
- Separate concerns (decoupling)
- Appropriate level of abstraction (work through abstractions)
- **Boy scout** rule
  - Leave your code better than you found it

Refactoring: the Typical Process

- Save the code you start with
  - Check-in or backup the current code
- Prepare tests to assure the behavior after the code is refactored
  - Unit tests / characterization tests
- Do refactoring one at a time
  - Keep refactoring small
  - Don't underestimate small changes
- Run the tests and they should pass / else revert
- Check-in (into the source control system)

Refactoring Tips

- Keep refactoring small
- One at a time
- Make a checklist
- Make a "later" / TODO list
- Check-in / commit frequently
- Add tests cases
- Review the results
  - Pair programming
- Use tools (Visual Studio + add-ins / Eclipse + plugins / others)



Code Smells

What? How Can Code "Smell"?

Code Smells

- Code smells == certain structures in the code that suggest the possibility of refactoring
- Types of code smells:
  - The bloaters
  - The obfuscators
  - Object-oriented abusers
  - Change preventers
  - Dispensables
  - The couplers

https://sourcemaking.com/refactoring/smells

Code Smells: the Bloaters

- **Long method**
  - Small methods are always better (easy naming, understanding, less duplicate code)
- **Large class**
  - Too many instance variables or methods
  - Violating "Single Responsibility" principle
- **Primitive obsession** (overused primitives)
  - Over-use of primitive values, instead of better abstraction
  - Can be extracted in separate class with encapsulated validation

Code Smells: the Bloaters (2)

- **Long parameter list** (**in** / **out** / **ref** parameters)
  - May indicate procedural rather than OO style
  - May be the method is doing too much things
- **Data clumps**
  - A set of data are always used together, but not organized together
  - E.g. credit card fields in the **Order** class
- **Combinatorial explosion**
  - Ex. **ListCars()**, **ListByRegion()**, **ListByManufacturer()**, **ListByManufacturerAndRegion()**, etc.
  - Solution may be the **Interpreter** pattern (LINQ)

Code Smells: the Bloaters (3)

- **Oddball solution**
  - A different way of solving a common problem
  - Not using consistency
  - Solution: Substitute algorithm or use an **Adapter**
- **Class doesn't do much**
  - Solution: Merge it with another class or remove it
- **Required setup** / **teardown code**
  - Requires several lines of code before its use
  - Solution: use parameter object, factory method, **IDisposable**

Code Smells: the Obfuscators

- **Regions**
  - The intent of the code is unclear and needs commenting (smell)
  - The code is too long to understand (smell)
  - Solution: partial class, a new class, organize code
- **Comments**
  - Should be used to tell **WHY**, not **WHAT** or **HOW**
  - Good comments: provide additional information, link to issues, explain an algorithm, explain reasons, give context
  - Link: Funny comments

Code Smells: the Obfuscators (2)

- **Poor** / **improper names**
  - Should be proper, descriptive and consistent
- **Vertical separation**
  - You should define variables just before first use to avoid scrolling
  - In JS variables are defined at the function start → use small functions
- **Inconsistency**
  - Follow the POLA (Principle of Least Astonishment)
  - Inconsistency is confusing and distracting
- **Obscured intent**
  - Code should be as expressive as possible

Code Smells: OO Abusers

- **Switch statement**
  - Can be replaced with polymorphism
- **Temporary field**
  - When passing data between methods
- **Class depends on subclass**
  - The classes cannot be separated (circular dependency)
  - May break the Liskov substitution principle
- **Inappropriate static field**
  - Strong coupling between **static** and callers
  - Static things cannot be replaced or reused

Code Smells: Change Preventers

- **Divergent change**
  - A class is commonly changed in different ways / different reasons
  - Violates SRP (single responsibility principle)
  - Solution: extract class
- **Shotgun surgery**
  - One change requires changes in many classes
    - Hard to find them, easy to miss some
  - Solution: move methods, move fields, reorganize the code

Code Smells: Dispensables

- **Lazy class**
  - Classes that don't do enough to justify their existence should be removed
  - Every class costs something to be understood and maintained
- **Data class**
  - Some classes with only fields and properties
  - Missing validation? Class logic split into other classes?
  - Solution: move related logic into the class

Code Smells: Dispensables (2)

- **Duplicated code**
  - Violates the DRY principle
  - Result of copy-pasted code
  - Solutions: extract method, extract class, pull-up method, **Template Method** pattern
- **Dead code** (code that is never used)
  - Usually detected by static analysis tools
- **Speculative generality**
  - "Some day we might need this …"
  - The "YAGNI" principle

Code Smells: the Couplers

- **Feature envy**
  - Method that seems more interested in a class other than the one it actually is in
  - Keep together things that change together
- **Inappropriate intimacy**
  - Classes that know too much about one another
  - Smells: inheritance, bidirectional relationships
  - Solutions: move method / field, extract class, change bidirectional to unidirectional association, replace inheritance with delegation

Code Smells: the Couplers (2)

- **The Law of Demeter (LoD)**
  - A given object should assume as little as possible about the structure or properties of anything else
  - Bad e.g.: **customer.Wallet.RemoveMoney()**
- **Indecent exposure**
  - Some classes or members are public but shouldn't be
  - Violates encapsulation
  - Can lead to inappropriate intimacy

Code Smells: the Couplers (3)

- **Message chains**
  - Something.Another.SomeOther.Other.YetAnother
  - Tight coupling between client and the structure of the navigation
- **Middle man**
  - Sometimes delegation goes too far
  - Sometimes we can remove it or inline it
- **Tramp data**
  - Pass data only because something else needs it
  - Solutions: Remove middle-man data, extract class

Refactoring Patterns

Well-Known Recipes for Improving the Code Quality

Rafactoring Patterns

- **When** should we perform refactoring of the code?
  - **Bad smells** in the code indicate **need of refactoring**
- **Unit tests** guarantee that refactoring preserves the behavior
- Refactoring patterns
  - **Large repeating code** fragments → extract duplicated code in separate method
  - **Large methods** → split them logically
  - **Large loop** body or **deep nesting** → extract method

Rafactoring Patterns (3)

- **Two classes are tightly coupled** → merge them or redesign them to separate their responsibilities
- **Public non-constant fields** → make them private and define accessing properties
- **Magic numbers in the code** → consider extracting constants
- **Bad named class** / **method** / **variable** → rename it
- **Complex boolean condition** → split it to several expressions or method calls

Rafactoring Patterns (4)

- **Complex expression** → split it into few simple parts
- **A set of constants is used as enumeration** → convert it to enumeration
- **Too complex method logic** → extract several more simple methods or even create a new class
- **Unused** classes, methods, parameters, variables → remove them
- **Large data** is passed by value without a good reason → pass it by reference

Rafactoring Patterns (5)

- Few classes share **repeating functionality** → extract base class and reuse the common code
- Different classes need to be instantiated depending on configuration setting → use factory
- **Code is not well formatted** → reformat it
- Too many classes in a single namespace → split classes logically into more namespaces
- **Unused using definitions** → remove them
- **Non-descriptive error messages** → improve them
- **Absence of defensive programming** → add it

Façade Pattern

- To deliver convenient interface from higher level to a **group of subsystems or single complex subsystem**
- Used in many Win32 API based classes to hide Win32 complexity
- http://www.dofactory.com/net/facade-design-pattern

Façade Pattern – Example

- Complex way:

```
speakers.On();
speakers.SetSurroundSound(true);
speakers.SetVolume(25/100);
speakers.SetOptions(SoundOptions.BlueRay);
environment.DimLights();
projector.On();
projector.SetMode(Modes.WideScreen);
dvd.On();
dvd.Play(movieName);
```

- Façade:

```
homeTheater.WatchMovie(movieName)
```

Refactoring Levels

Data-Level Refactoring

- Replace a magic number with a named constant
- Rename a variable with more informative name
- Replace an expression with a method
  - To simplify it or avoid code duplication
- Move an expression inline
- Introduce an intermediate variable
  - Introduce explaining variable
- Convert a multi-use variable to a multiple single-use variables
  - Create separate variable for each usage

Data-Level Refactoring (2)

- Create a local variable for local purposes rather than a parameter
- Convert a data primitive to a class
  - Additional behavior / validation logic (money)
- Convert a set of type codes (constants) to **enum**
- Convert a set of type codes to a class with subclasses with different behavior
- Change an array to an object
  - When you use an array with different types in it
- Encapsulate a collection

Statement-Level Refactoring

- Decompose a boolean expression
- Move a complex boolean expression into a well-named boolean function
- Use **break** or **return** instead of a loop control variable
- Return as soon as you know the answer instead of assigning a return value
- Consolidate duplicated code in conditionals
- Replace conditionals with polymorphism
- Use null-object design pattern instead of checking for **null**

Method-Level Refactoring

- Extract method / inline method
- Rename a method
- Convert a long routine to a class
- Add / remove parameter
- Combine similar methods by parameterizing them
- Substitute a complex algorithm with simpler
- Separate methods whose behavior depends on parameters passed in (create new ones)
- Pass a whole object rather than specific fields
- Encapsulate downcast / return interface types

Class-Level Refactoring

- Change a structure to class and vice versa
- Pull members up / push members down the hierarchy
- Extract specialized code into a subclass
- Combine similar code into a superclass
- Collapse hierarchy
- Replace inheritance with delegation
- Replace delegation with inheritance

Class Interface Refactorings

- Extract interface(s) / keep interface segregation
- Move a method to another class
- Split a class / merge classes / delete a class
- Hide a delegating class
  - **A** calls **B** and **C** when **A** should call **B** and **B** call **C**
- Remove the man in the middle
- Introduce (use) an extension class
  - When you have no access to the original class
  - Alternatively use the **Decorator** pattern

Class Interface Refactoring (2)

- Encapsulate an exposed member variable
  - Always use properties
  - Define proper access to getters and setters
    - Remove setters to read-only data
- Hide data and routines that are not intended to be used outside of the class / hierarchy
  - private -> protected -> internal -> public
- Use strategy to avoid big class hierarchies
- Apply other design patterns to solve common class and class hierarchy problems (**Façade**, **Adapter**, etc.)

System-Level Refactoring

- Move class (set of classes) to another namespace / assembly
- Provide a factory method instead of a simple constructor / use fluent API
- Replace error codes with exceptions
- Extract strings to resource files
- Use dependency injection
- Apply architecture patterns

Summary

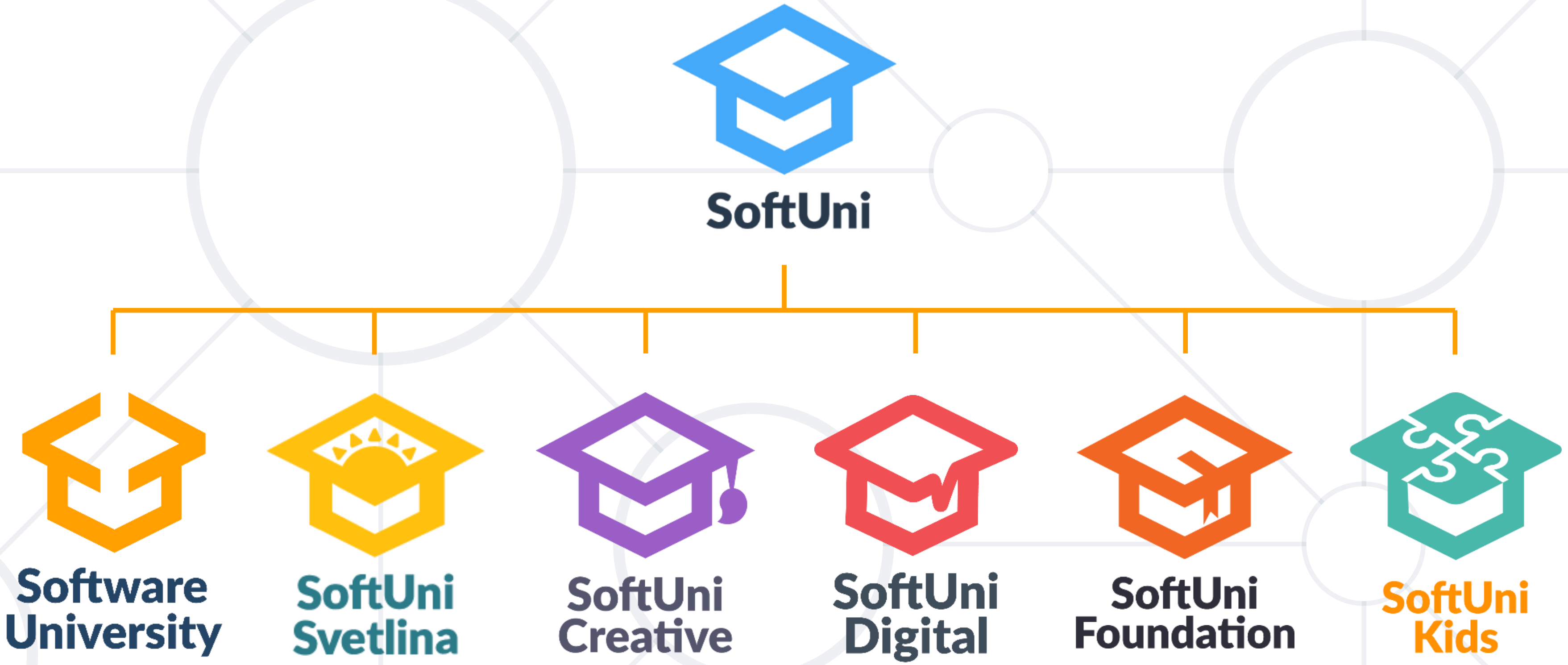
- Software Quality
  - External quality – software works correctly, no bugs and other problems
  - Internal quality – code is well structured, readable and maintainable
- Aspects of Code Quality
  - Quality classes, methods, control statements, loops, etc.
  - Good formatting, comments, strong cohesion and loose coupling
  - Testable code with unit tests

Summary

- Refactoring
  - Data-level
  - Statement-level
  - Method-level
  - Class-level
  - System-level
- Refactoring Patterns

Software University

Questions?



SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

License