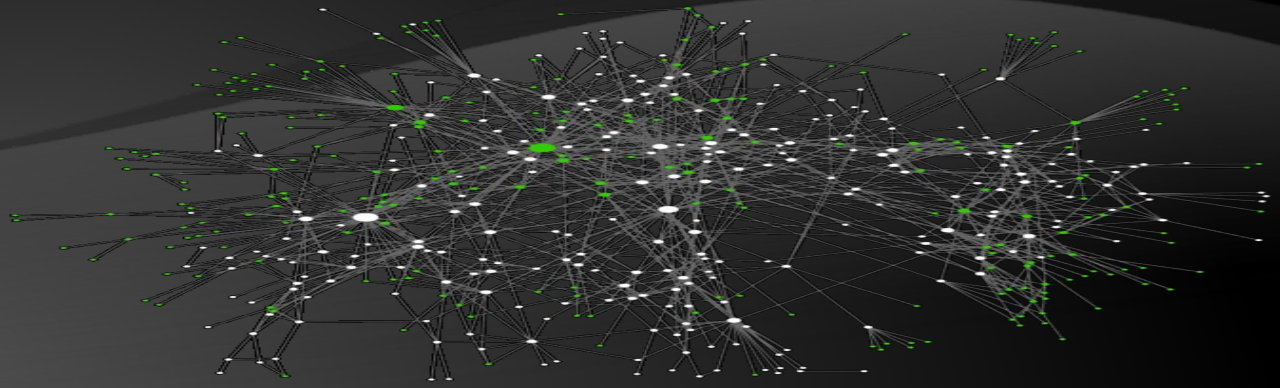
Telerik Academy



High-Quality Methods

How to Design and Implement High-Quality Methods? Understanding Cohesion and Coupling

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

Telerik Academy

Table of Contents

- Why Do We Need Methods?
- Strong Cohesion
- Loose Coupling
- Methods Parameters
- Pseudo Code

Telerik Academy

```
start:
   mov ah,08
   int 21h
   mov bl,al
  JMP output

   mov ah,01
   int 21h
output:
   mov dl,"("
   mov ah,02
   int 21h
   mov dl,bl
   int 21h
   mov dl,")"
   int 21h
exit:
   mov ah,4ch
   mov al,00
   int 21h
```

Jump

Imagine a long program consisting of instructions and jumps not organized in any structural way

Why Do We Need Methods?

Why We Need Methods?

- Methods (functions, routines) are important in software development
  - Reduce complexity
    - Divide and conquer: complex problems are split into composition of several simple
  - Improve code readability
    - Small methods with good method names make the code self-documenting
  - Avoid duplicating code
    - Duplicating code is hard to maintain

Why We Need Methods? (2)

- Methods simplify software development
  - Hide implementation details
    - Complex logic is encapsulated and hidden behind a simple interface
    - Algorithms and data structures are hidden and can be transparently replaced later
  - Increase the level of abstraction
    - Methods address the business problem, not the technical implementation:

```
Bank.accounts[customer].deposit(500);
```

Telerik Academy

Using Methods: Fundamentals

- Fundamental principle of correct method usage:

> A method should do what its name says or should indicate an error (throw an exception). Any other behavior is incorrect!

- Methods should do exactly what their names say
  - Nothing less (work in all possible scenarios)
  - Nothing more (no side effects)
- In case of incorrect input or incorrect preconditions, an error should be indicated

Bad Methods – Examples

```
int Sum(int[] elements)
{
  int sum = 0;
  foreach (int element in elements)
  {
      sum = sum + element;
  }
  return sum;
}
```

What will happen if we sum 2,000,000,000 + 2,000,000,000?

Result: -294967296

```
double CalcTriangleArea(double a, double b, double c)
{
  double s = (a + b + c) / 2;
  double area = Math.Sqrt(s * (s - a) * (s - b) * (s - c));
  return area;
}
```

What will happen if a = b = c = -1?

The same result as when a = b = c = 1 → both triangles have the same size.

Good Methods – Examples

```
long Sum(int[] elements)
{
  long sum = 0;
  foreach (int element in elements)
  {
      sum = sum + element;
  }
  return sum;
}
```

```
double CalcTriangleArea(double a, double b, double c)
{
  if (a <= 0 || b <= 0 || c <= 0)
  {
    throw new ArgumentException(
     "Sides should be positive.");
  }
  double s = (a + b + c) / 2;
  double area = Math.Sqrt(s * (s - a) * (s - b) * (s - c));
  return area;
}
```

Indicating an Error

- Some methods do not correctly indicate errors

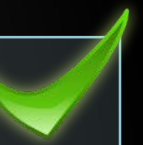
```
internal object GetValue(string propertyName)
{
    PropertyDescriptor descriptor =
        this.propertyDescriptors[propertyName];

    return descriptor.GetDataBoundValue();
}
```

- If the property name does not exist
  - A null reference exception will be thrown (implicitly) → it is not meaningful

Indicating an Error (2)

- Use the correct exception handling instead:



```
internal object GetValue(string propertyName)
{
    PropertyDescriptor descriptor =
        this.propertyDescriptors[propertyName];
    if (descriptor == null)
    {
        throw new ArgumentException("Property name: "
         + propertyName + " does not exists!");
    }
    return descriptor.GetDataBoundValue();
}
```

Symptoms of Wrong Methods

- Method that does something different than its name is wrong for at least one of these reasons:
  - The method sometimes returns incorrect result → bug
  - The method returns incorrect output when its input is incorrect or unusual → low quality
    - Could be acceptable for private methods only
  - The method does too many things → bad cohesion
  - The method has side effects → spaghetti code
  - Method returns strange value when an error condition happens → it should indicate the error

Wrong Methods – Examples

```
long Sum(int[] elements)
{
  long sum = 0;
  for (int i = 0; i < elements.Length; i++)
  {
    sum = sum + elements[i];
    elements[i] = 0; // Hidden side effect
  }
  return sum;
}
```

```
double CalcTriangleArea(double a, double b, double c)
{
  if (a <= 0 || b <= 0 || c <= 0)
  {
    return 0; // Incorrect result
  }
  double s = (a + b + c) / 2;
  double area = Math.Sqrt(s * (s - a) * (s - b) * (s - c));
  return area;
}
```

Strong Cohesion

- Methods should have strong cohesion
  - Should address single task and address it well
  - Should have clear intent
- Methods that address several tasks in the same time are hard to be named
- Strong cohesion is used in engineering
  - In computer hardware any PC component solves a single task
  - E.g. hard disk performs a single task – storage

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
  - Examples:

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

```
Char.IsLetterOrDigit(ch)
```

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

Telerik Academy

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)
```

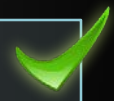
1. Connect to mail server
2. Send message headers
3. Send message body
4. Disconnect from the server

Telerik Academy

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)
```



    1. Retrieve input data from database
    2. Perform internal calculations over retrieved data
    3. Build the report
    4. Format the report as Excel worksheet
    5. Display the Excel worksheet on the screen

Telerik Academy

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()
```

1. Load user settings
2. Check for updates
3. 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 are grouped in a method for unclear reason
  - Incorrect example:

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

1. Prepares annual incomes report for given customer
2. Sorts an array of integers in increasing order
3. Calculates the square root of given number
4. Converts given MP3 file into WMA format
5. 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();
  byte[] encryptedData = destStream.ToArray();
  return encryptedData;
}
```

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());
  }
}
```

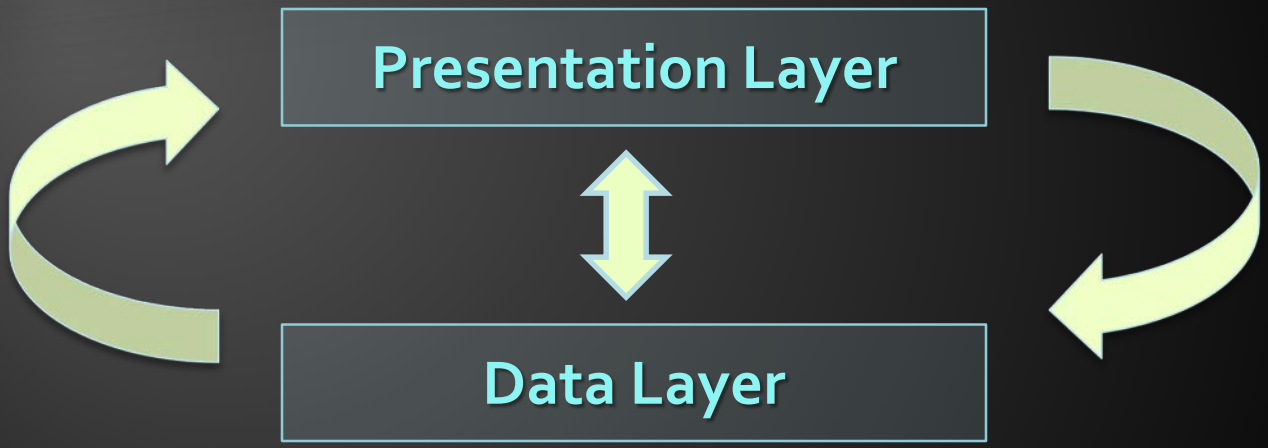
Tight Coupling in Real Code

- 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 Code

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



- Do not use a method for both top-down and bottom-up updates!
  - Both updates are essentially different, e.g. a `RemoveCustomer()` method in the `DataLayer`

Loose Coupling and OOP

- Reducing coupling with OOP techniques
  - Abstraction
    - Define a public interface and hide the implementation details
  - Encapsulation
    - Make methods and fields private unless they are definitely needed
    - Define new members as private
    - Increase visibility as soon as this is needed

Acceptable Coupling

- Method is coupled to its parameters
  - This is the best type of coupling

```
static int Sum(int[] elements) { … }
```

- Method in a class is coupled to some class fields
  - This coupling is usual, do not worry too much

```
static int CalcArea()
{ return this.Width * this.Height; }
```

- Method in a class is coupled to static methods, properties or constants in external class
  - This is normal, usually is not a problem

```
static double CalcCircleArea(double radius)
{ return Math.PI * radius * radius; }
```

Non-Acceptable Coupling

- Method in a class is coupled to static fields in external class
  - Use private fields and public properties
- Methods take as input data some fields that could be passed as parameters
  - Check the intent of the method
  - Is it designed to process internal class data or is utility method?
- Method is defined public without being part of the public class's interface → possible coupling

Methods Parameters

- Put most important parameters first
  - Put the main input parameters first
  - Put non-important optional parameters last
  - Example:

```
void RegisterUser(string username, string password,
Date accountExpirationDate, Role[] roles)
```

  - Incorrect example:

```
void RegisterUser(Role[] roles, string password, string
username, Date accountExpirationDate)
```

```
void RegisterUser(string password, Date
accountExpirationDate, Role[] roles, string username)
```

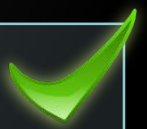
Methods Parameters (2)

- Do not modify the input parameters
  - Use new variable instead
  - Incorrect example:

```
bool CheckLogin(string username, string password)
{
  username = username.ToLower();
  // Check the username / password here …
}
```

  - Correct example:

```
bool CheckLogin(string username, string password)
{
  string usernameLowercase = username.ToLower();
  // Check the username / password here …
}
```



Method Parameters (3)

- Use parameters consistently
  - Use the same names and the same order in all methods
  - Incorrect example:

```
void EncryptFile(Stream input, Stream output, string key);
void DecryptFile(string key, Stream output, Stream input);
```

  - Output parameters should be put last

```
FindCustomersAndIncomes(Region region, out Customer[]
customers, out decimal[] incomes)
```

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 Much Parameters Methods 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

Methods 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 name and clear intent → check this!

Pseudocode

- Pseudocode can be helpful in:
  - Routines design
  - Routines coding
  - Code verification
  - Cleaning up unreachable branches in a routine

Designing in Pseudocode

- What the routine will abstract i.e. the information a routine will hide?
- Routine input parameters
- Routine output
- Preconditions
  - Conditions that have to be true before a routine is called
- Postconditions
  - Conditions that have to be true after routine execution

Design Before Coding

- Why it is better to spend time on design before you start coding?
  - The functionality may be already available in a library, so you do not need to code at all!
  - You need to think of the best way to implement the task considering your project requirements
  - If you fail on writing the code right the first time, you need to know that programmers get emotional to their code

Pseudocode Example

Routine that evaluates an aggregate expression for a database column (e.g. Sum, Avg, Min)

Parameters: Column Name, Expression

Preconditions:

(1) Check whether the column exists and throw an argument exception if not

(2) If the expression parser cannot parse the expression throw an ExpressionParsingException

Routine code: Call the evaluate method on the DataView class and return the resulting value as string

Public Routines in Libraries

- Public routines in libraries and system software is hard to change
  - Because customers want no breaking changes
- Two reasons why you need to change a public routine:
  - New functionality has to be added conflicting with the old features
  - The name is confusing and makes the usage of the library unintuitive
- Design better upfront, or refactor carefully

Method Deprecation

- Deprecated method
  - About to be removed in future versions
- When deprecating an old method
  - Include that in the documentation
  - Specify the new routine that has to be used
- Use the [Obsolete] attribute in .NET

```
[Obsolete("CreateXml() method is deprecated.
  Use CreateXmlReader instead.")]
public void CreateXml (…)
{
  …
}
```

Inline Routines

- Inline routines provide two benefits:
  - Abstraction
  - Performance benefit of not creating a new routine on the stack
- Some applications (e.g. games) need that optimization
  - Used for the most frequently used routines
  - Example: a short routine called 100,000 times
- Not all languages support Inline routines

Conclusion

- Designing and coding routines is engineering activity
  - There is no perfect solution
- Competing solutions usually demonstrate different trade-offs
  - The challenge of the programmer is to
    - Evaluate the requirements
    - Choose the most appropriate solution from the available options
    - Ensure loose coupling / strong cohesion

Telerik Academy

High-Quality Methods

http://academy.telerik.com

Homework

- Use the provided source code "High-Quality-Methods-Homework.zip".
  1. Take the VS solution "`Methods`" and refactor its code to follow the guidelines of high-quality methods. Ensure you handle errors correctly: when the methods cannot do what their name says, throw an exception (do not return wrong result). Ensure good cohesion and coupling, good naming, no side effects, etc.

Telerik Academy

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