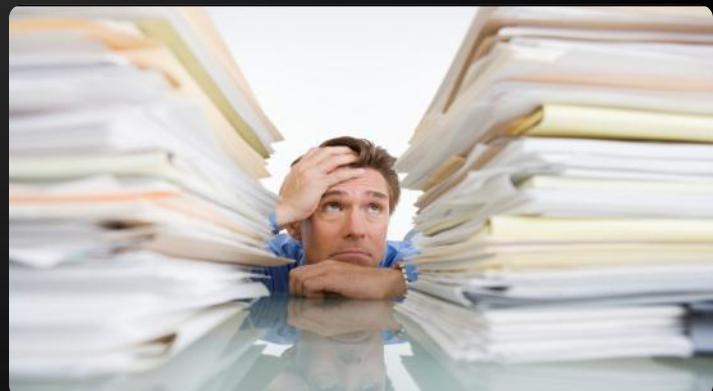




Code Documentation and Comments in the Program

Revealing the Secrets of Self-Documenting Code

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



1. The Concept of Self-Documenting Code
2. Bad Comments
3. Good Programming Style
4. To Comment or Not to Comment?
5. Key Points commenting
6. Recommended practices
7. C# XML Documentation Comments





Comments and Code Documentation

The Concept of Self-Documenting Code

What is Project Documentation?

- ◆ **Consists of documents and information**
 - ◆ Both inside the source-code and outside
- ◆ **External documentation**
 - ◆ At a higher level compared to the code
 - ◆ Problem definition, requirements, architecture, design, project plans, test plans. etc.
- ◆ **Internal documentation**
 - ◆ Lower-level – explains a class, method or a piece of code



Programming Style

- ◆ Main contributor to code-level documentation
 - The program structure
 - Straight-forward, easy-to-read and easily understandable code
 - Good naming approach
 - Clear layout and formatting
 - Clear abstractions
 - Minimized complexity
 - Loose coupling and strong cohesion



Bad Comments – Example

```
public static List<int> FindPrimes(int start, int end)
{
    // Create new list of integers
    List<int> primesList = new List<int>();
    // Perform a loop from start to end
    for (int num = start; num <= end; num++)
    {
        // Declare boolean variable, initially true
        bool prime = true;
        // Perform loop from 2 to sqrt(num)
        for (int div = 2; div <= Math.Sqrt(num); div++)
        {
            // Check if div divides num with no remainder
            if (num % div == 0)
            {
                // We found a divider -> the number is not prime
                prime = false;
                // Exit from the loop
                break;
            }
        }
        if (prime)
            primesList.Add(num);
    }
    return primesList;
}
```



(continues on the next slide)

Bad Comments – Example (2)

```
// Continue with the next loop value  
}  
  
// Check if the number is prime  
if (prime)  
{  
    // Add the number to the list of primes  
    primesList.Add(num);  
}  
}  
  
// Return the list of primes  
return primesList;  
}
```



Self-Documenting Code – Example

```
public static List<int> FindPrimes(int start, int end)
{
    List<int> primesList = new List<int>();
    for (int num = start; num <= end; num++)
    {
        bool isPrime = IsPrime(num);
        if (isPrime)
        {
            primesList.Add(num);
        }
    }
    return primesList;
}
```



Good code does not
need comments. It is
self-explaining.

(continues on the next slide)

Self-Documenting Code – Example (2)

```
private static bool IsPrime(int num)
{
    bool isPrime = true;
    int maxDivider = Math.Sqrt(num);
    for (int div = 2; div <= maxDivider; div++)
    {
        if (num % div == 0)
        {
            // We found a divider -> the number is not prime
            isPrime = false;
            break;
        }
    }
    return isPrime;
}
```



Good methods have good name and are easy to read and understand.

This comment explain non-obvious details. It does not repeat the code.

Bad Programming Style – Example

```
for (i = 1; i <= num; i++)
{
    meetsCriteria[i] = true;
}
for (i = 2; i <= num / 2; i++)
{
    j = i + i;
while (j <= num)
{
    meetsCriteria[j] = false;
    j = j + i;
}
}
for (i = 1; i <= num; i++)
{
    if (meetsCriteria[i])
    {
        Console.WriteLine(i + " meets criteria.");
    }
}
```



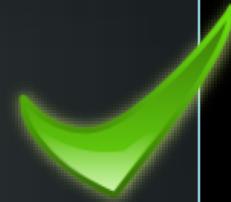
Uninformative variable names. Crude layout.

Good Programming Style – Example

```
for (primeCandidate = 1; primeCandidate <= num; primeCandidate++)
{
    isPrime[primeCandidate] = true;
}

for (int factor = 2; factor < (num / 2); factor++)
{
    int factorableNumber = factor + factor;
    while (factorableNumber <= num)
    {
        isPrime[factorableNumber] = false;
        factorableNumber = factorableNumber + factor;
    }
}

for (primeCandidate = 1; primeCandidate <= num; primeCandidate++)
{
    if (isPrime[primeCandidate])
    {
        Console.WriteLine(primeCandidate + " is prime.");
    }
}
```



Self-Documenting Code

- ◆ Code that relies on good programming style
 - ◆ To carry major part of the information intended for the documentation
- ◆ Self-documenting code fundamental principles

The best documentation is the code itself.

Make the code self-explainable and self-documenting, easy to read and understand.

Do not document bad code, rewrite it!

Self-Documenting Code Checklist

◆ Classes

- Does the class's interface present a consistent abstraction?
- Does the class's interface make obvious how you should use the class?
- Is the class well named, and does its name describe its purpose?
- Can you treat the class as a black box?
- Do you reuse instead of repeating code?

Self-Documenting Code Checklist (2)

◆ Methods

- Does each routine's name describe exactly what the method does?
- Does each method perform one well-defined task with minimal dependencies?

◆ Data Names

- Are type names descriptive enough to help document data declarations?
- Are variables used only for the purpose for which they're named?

Self-Documenting Code Checklist (3)

- ◆ Data Names
 - ◆ Does naming conventions distinguish among type names, enumerated types, named constants, local variables, class variables, and global variables?
- ◆ Others
 - ◆ Are data types simple so that they minimize complexity?
 - ◆ Are related statements grouped together?



To Comment or Not to Comment?

"Everything the Compiler
Needs to Know is in the Code!"

Effective Comments

- ◆ Effective comments do not repeat the code
 - They explain it at a higher level and reveal non-obvious details
- ◆ The best software documentation is the source code itself – keep it clean and readable!
- ◆ Self-documenting code is self-explainable and does not need comments
 - Simple design, small well named methods, strong cohesion and loose coupling, simple logic, good variable names, good formatting, ...

◆ Misleading comments

```
// write out the sums 1..n for all n from 1 to num  
current = 1;  
previous = 0;  
sum = 1;  
for (int i = 0; i < num; i++)  
{  
    Console.WriteLine( "Sum = " + sum );  
    sum = current + previous;  
    previous = current;  
    current = sum;  
}
```



What problem does
this algorithm solve?

Can you guess that sum
is equal to the i^{th}
Fibonacci number?

Effective Comments – Mistakes (2)

- Comments repeating the code:

```
// set product to "base"  
product = base;
```

Obviously...



```
// loop from 2 to "num"  
for ( int i = 2; i <= num; i++ )  
{  
    // multiply "base" by "product"  
    product = product * base;  
}  
Console.WriteLine( "Product = " + product );
```

You don't say...

Effective Comments – Mistakes (3)

◆ Poor coding style:

```
// compute the square root of Num using
// the Newton-Raphson approximation
r = num / 2;
while (abs(r - (num/r)) > TOLERANCE)
{
    r = 0.5 * (r + (num/r) );
}
Console.WriteLine( "r = " + r );
```



◆ Do not comment bad code, rewrite it



Key Points for Effective Comments

- ◆ Use commenting styles that don't break down or discourage modification

// Variable	Meaning
// -----	-----
// xPos	X coordinate position (in meters)
// yPos	Y coordinate position (in meters)
// zPos	Z coordinate position (in meters)
// radius	The radius of the sphere where the battle ship is located (in meters)
// distance	The distance from the start position (in meters)



- ◆ The above comments are unmaintainable

Key Points for Effective Comments (2)

- ◆ Comment the code intent, not implementation details

```
// Scan char by char to find the command-word terminator ($)
done = false;
maxLen = inputString.Length;
i = 0;
while (!done && (i < maxLen))
{
    if (inputString[i] == '$')
    {
        done = true;
    }
    else
    {
        i++;
    }
}
```



Key Points for Effective Comments (3)

- ◆ Focus your documentation efforts on the code

```
// Find the command-word terminator
foundTheTerminator = false;
maxCommandLength = inputString.Length();
testCharPosition = 0;
while (!foundTheTerminator &&
       (testCharPosition < maxCommandLength))
{
    if (inputString[testCharPosition] == COMMAND_WORD_TERMINATOR)
    {
        foundTheTerminator = true;
        terminatorPosition = testCharPosition;
    }
    else
    {
        testCharPosition = testCharPosition + 1;
    }
}
```

Better code → less comments



Key Points for Effective Comments (4)

- ◆ Focus paragraph comments on the why rather than the how

```
// Establish a new account
if (accountType == AccountType.NewAccount)
{
    ...
}
```



- ◆ Use comments to prepare the reader for what is to follow
- ◆ Avoid abbreviations



Guidelines for Effective Comments (5)

- ◆ Comment anything that gets around an error or an undocumented feature
 - ◆ E.g. *// This is workaround for bug #3712*
- ◆ Justify violations of good programming style
- ◆ Don't comment tricky code – rewrite it
- ◆ Use built-in features for commenting
 - ◆ XML comments in C#
 - ◆ JavaDoc in Java, ...

General Guidelines for Higher Level Documentation

- ◆ Describe the design approach to the class
- ◆ Describe limitations, usage assumptions, and so on
- ◆ Comment the class interface (public methods / properties / events / constructors)
- ◆ Don't document implementation details in the class interface
- ◆ Describe the purpose and contents of each file
- ◆ Give the file a name related to its contents

C# XML Documentation Comments



- ◆ In C# you can document the code by XML tags in special comments
 - ◆ Directly in the source code
- ◆ For example:

```
/// <summary>
/// This class performs an important function.
/// </summary>
public class MyClass { }
```

- ◆ The XML doc comments are not metadata
 - ◆ Not included in the compiled assembly and not accessible through reflection

- ◆ **<summary>**

- ◆ A summary of the class / method / object

- ◆ **<param>**

```
<param name="name">description</param>
```

- ◆ Describes one of the parameters for a method

- ◆ **<returns>**

- ◆ A description of the returned value

- ◆ **<remarks>**

- ◆ Additional information (remarks)

XML Documentation Tags (2)

- ◆ <c> and <code>

- ◆ Gives you a way to indicate code

- ◆ <see> and <seealso> and cref

- ◆ Code reference

```
<seealso cref="TestClass.Main"/>
```

- ◆ <exception>

```
<exception cref="type">description</exception>
```

- ◆ Lets you specify which exceptions can be thrown
- ◆ All tags: <http://msdn.microsoft.com/en-us/library/5ast78ax.aspx>

XML Documentation Example

```
/// <summary>
/// The GetZero method. Always returns zero.
/// </summary>
/// <example>
/// This sample shows how to call the <see cref="GetZero"/> method.
/// <code>
/// class TestClass
/// {
///     static int Main()
///     {
///         return GetZero();
///     }
/// }
/// </code>
/// </example>
public static int GetZero()
{
    return 0;
}
```

C# XML Documentation

- ◆ Visual Studio will use the XML documentation for autocomplete
 - ◆ Automatically, just use XML docs
- ◆ Compiling the XML documentation:
 - ◆ Compile with /doc the to extract the XML doc into an external XML file
 - ◆ Use Sandcastle or other tool to generate CHM / PDF / HTML / other MSDN-style documentation
 - ◆ Example: <http://www.ewoodruff.us/shfbdocs/>



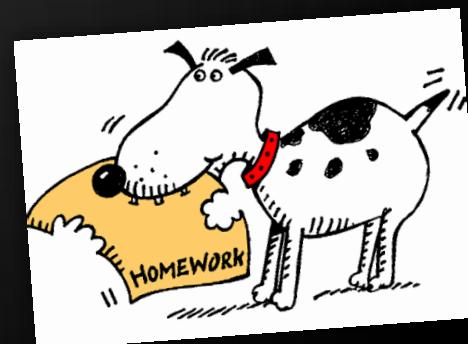
Demo: C# XML Documentation Comments

Code Documentation and Comments in the Program

Questions?

1. Open project located in [4. Code Documentation and Comments Homework.zip](#) and:

- Add comments where necessary
- For each public member add documentation as C# XML Documentation Comments
- * Play with Sandcastle / other tools and try to generate CHM book



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

