

# Clean Object-Oriented Design

---

Objektumorientált szoftvertervezés  
Object-oriented software design

Dr. Balázs Simon  
BME, IIT

# Outline

---

- What is clean code?
- Guidelines:
  - Selecting meaningful names
  - Writing functions
  - Writing and omitting comments
  - Defining and handling exceptions
  - Objects vs. data structures

# What is clean code?

---

# Bad code

---

- Code will be here for a long time
  - program code in a general purpose language
  - higher abstraction level: domain-specific languages
  - the abstraction level will continue to increase
- Bad code:
  - hard to read
  - hard to understand
  - hard to maintain
  - usual causes:
    - hurry
    - tired of working on the software and wanting it to be over
    - laziness
    - inexperience
    - it will be cleaned up later (=never)

# Problems with bad code

---

- Bad code decreases productivity
  - management adds more staff to increase or maintain productivity
  - but it will just get worse: new people don't understand the code, don't know the original design intentions and have to work under pressure
- Rewriting the whole system takes time
  - the old and new systems must be maintained and developed in parallel under changing requirements
  - by the time the new system is ready, its original developers are gone and it can be a mess again

# Who is responsible for bad code?

---

- Not responsible:
  - tight schedule
  - changing requirements
  - management
- Responsible:
  - we, developers
- Bad code:
  - slows us down
  - makes changing the code harder
  - is not an excuse
- The only way to go fast in the short run and in the long run is to keep the code as clean as possible *at all times*

# Clean code

---

- Code is mostly read, not written
  - time ratio is 10:1 or more
- Clean code is an art:
  - hard to define exactly
  - even if you recognize it, it does not mean you can write it
  - it must be practiced
- Properties of clean code:
  - easy to read: should read like a well written prose
  - easy to understand
  - elegant and efficient
  - minimal dependencies
  - complete error handling
  - focused: does one thing well
  - easy for other people to enhance it

# Meaningful names

---



# Problem

---

```
public List<int[]> GetThem() {  
    List<int[]> list1 = new List<int[]>();  
    for (int[] x in theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```

What does the function do?

Why is the 0<sup>th</sup> item special?

What is in the list?

What does 4 mean?

What to do with the result?

The answers cannot be explicitly seen, but they could be!

# Problem

---

Why write a comment? Why not give a better name to the variable?



```
int d; // elapsed time in days
```

# Use intention-revealing names

---

- Choosing good names takes time but saves more than it takes
- Take care with your names
  - should tell you why something exists, what it does, and how it is used
- Change them when you find better ones
- Everyone who reads your code (including you) will be happier if you do
- If a name requires a comment, then the name does not reveal its intent
- A long descriptive name is better than a short enigmatic name
- IDE helps with long names, no need for abbreviations

# Solution

---

```
public List<int[]> getThem() {  
    List<int[]> list1 = new List<int[]>();  
    for (int[] x in theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```



```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new List<int[]>();  
    for (int[] cell in gameBoard)  
        if (cell[Consts.StatusValue] == Consts.Flagged)  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

# Or a more object-oriented solution

---

```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new List<int[]>();  
    for (int[] cell in gameBoard)  
        if (cell[Consts.StatusValue] == Consts.Flagged)  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```



```
public List<Cell> getFlaggedCells() {  
    List<Cell> flaggedCells = new List<Cell>();  
    for (Cell cell in gameBoard)  
        if (cell.isFlagged())  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

# Solution

---

```
int d; // elapsed time in days
```



```
int elapsedTimeInDays;
```

← No comment is necessary!

Other examples:

```
int daysSinceCreation;  
int daysSinceModification;  
int fileAgeInDays;
```

Notice the units!

We don't need to read the documentation or to write additional comments.  
The code documents itself.

# Problem

`string xml; // example` ← XML means something else

`Set<Account> accountList;` ← this is not a list

```
class XYZControllerForEfficientHandlingOfStrings
{
}
```

hard to spot the difference

```
class XYZControllerForEfficientStorageOfStrings
{
}
```

```
int a = 1;
if ( 0 == 1 )
    a = 01;
else
    1 = 01;
```

Difference between O and 0?

Difference between L and 1?

Maybe in another font?

```
int a = 1;
if ( 0 == 1 )
    a = 01;
else
    1 = 01;
```

# Avoid disinformation

---

- Avoid false clues that obscure the meaning of code
- Don't name something a **List** if it is not a list
  - better names: **accountGroup** or just simply **accounts**
- Don't use names which vary in small ways
  - it takes time to spot the difference
  - it can easily lead to bugs, if we use the wrong one
- Don't use '**0**' and '**1**' as names, they can be easily confused with  $\emptyset$  and 1



# Solution

---

```
string example;  
  
Set<Account> accounts;  
  
class StringCache  
{  
}  
  
class StringDatabase  
{  
}  
  
int node = leaf;  
if (current == leaf)  
    node = right;  
else  
    leaf = 1;
```

# Problem

Which is the source, which is the destination?  
Cannot be seen from the outside!

```
public static void CopyChars(char a1[], char a2[]) {  
    for (int i = 0; i < a1.Length; i++) {  
        a2[i] = a1[i];  
    }  
}
```

```
GetActiveAccount();  
GetActiveAccounts();  
GetActiveAccountInfo();
```

How are the programmers supposed to know  
which of these functions to call?

# Make meaningful distinctions

---

- Don't use names like **a1**, **a2**, ..., **aN**
  - these names are non-informative
- Don't use noise words like '**a**', '**the**', '**variable**', '**info**', etc.
  - **theMessage** vs. **Message**
  - **NameString** vs. **Name**
  - **AccountData** vs. **Account**
  - **CustomerObject** vs. **Customer**
- Don't reuse variables for different purposes
  - the compiler is *very* clever:
    - it can assign the same register to different variables
    - it can optimize a lot of things (e.g. loops, tail recursion, etc.)
  - make the code readable and leave the rest to the compiler

# Solution

---

```
public static void CopyChars(char source[], char destination[]) {  
    for (int i = 0; i < source.Length; i++) {  
        destination[i] = source[i];  
    }  
}
```

```
GetActiveAccount();  
GetAllActiveAccounts();  
GetActiveAccountProfile();
```

# Problem

---

How do you pronounce these?  
How can you talk about these?  
How do you search these?

```
class DtaRcrd102 {  
    private DateTime genymdhms;  
    private DateTime modymdhms;  
    private const string pszqint = "102";  
    /* ... */  
};
```

```
public class Part {  
    private string m_dsc; // The textual description  
    void SetName(string name) {  
        m_dsc = name;  
    }  
}
```

# Use pronounceable names

---

- Use names that can be pronounced
  - so that you can talk about it
  - “gen why emm dee aich emm ess” vs. generationTimestamp
- Examples:
  - “Hey, Mikey, take a look at this record! The gen why emm dee aich emm ess is set to tomorrow’s date! How can that be?”
  - vs.
  - “Hey, Mikey, take a look at this record! The generation timestamp is set to tomorrow’s date! How can that be?”

# Use searchable names

---

- Use names that can be searched
  - `ymdhms` vs. `timestamp`
  - `10` vs. `MaxClassesPerStudent`
  - a searchable name is better than a constant
- Don't use abbreviations or made up acronyms
  - `ymdhms` vs. `timestamp`
  - Elon Musk also banned making up acronyms at SpaceX:
    - “The key test for an acronym is to ask whether it helps or hurts communication.”
  - commonly known acronyms are fine (`XML`, `GUI`, etc.)

# Avoid encodings

---

- Don't use Hungarian notation (= type is part of the variable name)
  - modern compilers enforce types
  - What if the type of a variable changes? Will you rename it? What if you forget to?
- Don't use member prefixes (**m\_name**, **\_name**, etc.)
  - classes should be small enough that you do not need them
  - IDEs can color members and local variables differently
  - or just simply write the **this** keyword explicitly
  - people tend to get used to prefixes and get 'prefix-blind'
- Naming interfaces and their implementing classes
  - **IList** vs. **List**
  - **List** vs. **ListImpl** or **CList**
- Encodings make code harder to read
- Exceptions:
  - framework convention (e.g. .NET interfaces are prefixed: **IList**)
  - company coding standard requires it
    - (but think about this rule when creating the coding standard)



# Solution

---

```
class DtaRcrd102 {  
    private DateTime genymdhms;  
    private DateTime modymdhms;  
    private const string pszqint = "102";  
    /* ... */  
};
```



```
class Customer {  
    private DateTime generationTimestamp;  
    private DateTime modificationTimestamp;  
    private const string recordId = "102";  
    /* ... */  
};
```

# Solution

---

```
public class Part {  
    private string m_dsc; // The textual description  
    void SetName(string name) {  
        m_dsc = name;  
    }  
}
```



```
public class Part {  
    private string description;  
    void SetDescription(string description) {  
        this.description = description;  
    }  
}
```

# Problem

What does the function do?  
What are the parameters?

```
public Account A(List<Account> al, string n)
{
    for (int loopVariable = 0; loopVariable < al.Count; ++loopVariable)
    {
        if (al[loopVariable].Number == n) return al[loopVariable];
    }
    return null;
}
```

A long loop variable name  
is inconvenient to use.

Too long name.

```
protected string GenerateSqlStatementFromHardCodedValuesAndSafeDataTypes(
{
    StringBuilder sb = new StringBuilder(1024);
    sb.AppendFormat("select comment_id, comment, commentor_id from {0} ",
                    TableName);
    sb.AppendFormat("where Comment_id={0} ", Filter.Value);
    return sb.ToString();
}
```

# Length of names

---

- Shorter names are generally better than longer ones, so long as they are clear
  - encodings, abbreviations and made up acronyms are **not** clear
- If you cannot find a clear short name, use a longer name
  - don't be afraid to make a name long
  - a long descriptive name is better than a short enigmatic name
  - a long descriptive name is better than a long descriptive comment
  - use a naming convention that allows multiple words to be easily read
- Single-letter names can **only** be used as local variables inside short methods
  - e.g. loop variables can be **i**, **j**, **k** (but not **l**)
- The length of a name should correspond to the size of its scope
  - local variables can be short
  - method names, parameter names and class names should be longer

# Solution

We know what the function does.

We know what the parameters are.

```
public Account FindAccountByName(List<Account> accounts, string name)
{
    for (int i = 0; i < accounts.Count; ++i)
    {
        if (accounts[i].Name == name) return accounts[i];
    }
    return null;
}
```

Short loop variable is more convenient.

Short and meaningful name.

```
protected string GenerateSafeSql()
{
    StringBuilder sb = new StringBuilder(1024);
    sb.AppendFormat("select comment_id, comment, commentor_id from {0} ",
        TableName);
    sb.AppendFormat("where Comment_id={0} ", Filter.Value);
    return sb.ToString();
}
```

# Class names

---

- Classes and objects should have noun or noun phrase names like **Customer**, **WikiPage**, **Account**, and **AddressParser**
- Avoid words like **Manager**, **Processor**, **Data**, or **Info** in the name of a class
- A class name should not be a verb
- Exceptions:
  - extracting behavior, e.g. Visitor, Strategy design patterns

# Method names

- Methods should have verb or verb phrase names like **PostPayment**, **DeletePage**, or **Save**
- Accessors, mutators, and predicates should be named for their value and prefixed with **Get**, **Set**, and **Is**

```
string name = employee.Name;  
customer.Name = "Mike";  
if (paycheck.IsPosted)...
```

```
String name = employee.getName();  
customer.setName("Mike");  
if (paycheck.isPosted())...
```

- When constructors are overloaded, use static factory methods with names that describe the arguments
  - consider enforcing their use by making the corresponding constructors private

```
Complex centerPoint = Complex.FromRealNumber(23.0);
```

- vs.

```
Complex centerPoint = new Complex(23.0);
```

# Problem

---

```
class Bank
{
    Set<Account> FetchAccounts();
}
```

```
class Account
{
    Set<Person> RetrieveOwners();
}
```

```
class Person
{
    string GetName();
}
```

Inconsistent names





# Pick one word per concept

---

- Don't use different names for the same concept
- They are confusing
- They are hard to remember
- Examples:
  - fetch, retrieve, get
  - stop, kill, quit, end
  - controller, manager, driver
    - What is the essential difference between a DeviceManager and a ProtocolController? Why are both not controllers or both not managers? Are they both Drivers really?

# Solution

---

```
class Bank
{
    Set<Account> GetAccounts();
}
```

```
class Account
{
    Set<Person> GetOwners();
}
```

```
class Person
{
    string GetName();
}
```

# Use the right domain


---

- Use solution domain names
  - code is read by programmers, they know computer science
  - use the appropriate algorithm names, pattern names, math terms, etc.
  - e.g. **AccountVisitor**, **JobQueue**
- Use problem domain names
  - it is easier to make changes and extend the model if the requirements change
- Separating solution and problem domain concepts is part of the job of a good programmer and designer
  - the code that has more to do with problem domain concepts should have names drawn from the problem domain

# Problem

```
private void PrintGuessStatistics(char candidate, int count) {
    string number;
    string verb;
    string pluralModifier;
    if (count == 0) {
        number = "no";
        verb = "are";
        pluralModifier = "s";
    } else if (count == 1) {
        number = "1";
        verb = "is";
        pluralModifier = "";
    } else {
        number = count.ToString();
        verb = "are";
        pluralModifier = "s";
    }
    String guessMessage = string.Format(
        "There {0} {1} {2}{3}", verb, number, candidate, pluralModifier
    );
    Print(guessMessage);
}
```

We have to read through and understand the whole function to know the context and meaning of the variables.



# Add meaningful context

---

- There are a few names which are meaningful in and of themselves, but most are not
  - Example:
    - If we see `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state`, and `zipcode`, we can infer that they are part of an `Address`
    - But what if you just saw the `state` variable being used alone in a method? Would you automatically infer that it was part of an address?
- Place names in context for your reader by enclosing them in well-named classes, functions, or namespaces
  - e.g. `Address` class
- When all else fails, then prefixing the name may be necessary as a last resort
  - e.g. `addrFirstName`, `addrLastName`, `addrState`

# Don't add meaningless context

---

- Don't prefix every class with the same thing
  - e.g. **GSD**AccountAddress, **GSD**Person
  - the prefix won't add any meaningful context to the class name
  - the IDE will list all the classes when you start typing
- Examples:
  - **accountAddress** and **customerAddress** are fine names for instances of the class **Address**
  - **accountAddress** and **customerAddress** are poor names for classes
  - **MAC** vs. **MACAddress**
  - **URI** vs. **WebAddress**

# Solution

```
public class GuessStatisticsMessage {  
    private string number;  
    private string verb;  
    private string pluralModifier;
```

A separate class with a meaningful name provides a clear context for all the variables.

```
    public string Make(char candidate, int count) {  
        CreatePluralDependentMessageParts(count);  
        return string.Format("There {0} {1} {2}{3}",  
            verb, number, candidate, pluralModifier );  
    }
```

```
    private void CreatePluralDependentMessageParts(int count) {  
        if (count == 0) {  
            ThereAreNoLetters();  
        } else if (count == 1) {  
            ThereIsOneLetter();  
        } else {  
            ThereAreManyLetters(count);  
        }  
    }
```

Smaller functions are easier to read and understand.

Cases are separated into smaller functions with meaningful names.

...

# Solution

```
public class GuessStatisticsMessage {
```

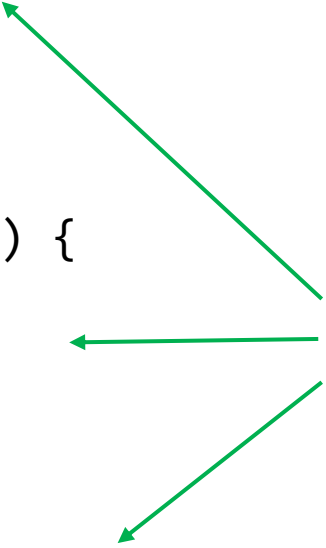
```
...
```

```
    private void ThereAreManyLetters(int count) {  
        number = count.ToString();  
        verb = "are";  
        pluralModifier = "s";  
    }
```

```
    private void ThereIsOneLetter() {  
        number = "1";  
        verb = "is";  
        pluralModifier = "";  
    }
```

```
    private void ThereAreNoLetters() {  
        number = "no";  
        verb = "are";  
        pluralModifier = "s";  
    }
```

```
}
```



Cases are separated into smaller functions with meaningful names.

Smaller functions are easier to read and understand.



# Functions

---

# Problem

```
public static string TestableHtml(PageData pageData, boolean includeSuiteSetup) {
    WikiPage wikiPage = pageData.GetWikiPage();
    StringBuilder builder = new StringBuilder();
    if (pageData.HasAttribute("Test")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup = PageCrawlerImpl.GetInheritedPage(SuiteResponder.SuiteSetupName, wikiPage);
            if (suiteSetup != null) {
                WikiPagePath pagePath = suiteSetup.GetPageCrawler().GetFullPath(suiteSetup);
                String pagePathName = PathParser.Render(pagePath);
                builder.Append("!include -setup .").Append(pagePathName).Append("\n");
            }
        }
        WikiPage setup = PageCrawlerImpl.GetInheritedPage("SetUp", wikiPage);
        if (setup != null) {
            WikiPagePath setupPath = wikiPage.GetPageCrawler().GetFullPath(setup);
            String setupPathName = PathParser.Render(setupPath);
            builder.Append("!include -setup .").Append(setupPathName).Append("\n");
        }
    }
    builder.Append(pageData.GetContent());
    if (pageData.HasAttribute("Test")) {
        WikiPage teardown = PageCrawlerImpl.GetInheritedPage("TearDown", wikiPage);
        if (teardown != null) {
            WikiPagePath tearDownPath = wikiPage.GetPageCrawler().GetFullPath(teardown);
            String tearDownPathName = PathParser.Render(tearDownPath);
            builder.Append("\n").Append("!include -teardown .").Append(tearDownPathName).Append("\n");
        }
        if (includeSuiteSetup) {
            WikiPage suiteTeardown = PageCrawlerImpl.GetInheritedPage(SuiteResponder.SuiteTearDownName, wikiPage);
            if (suiteTeardown != null) {
                WikiPagePath pagePath = suiteTeardown.GetPageCrawler().GetFullPath(suiteTeardown);
                String pagePathName = PathParser.Render(pagePath);
                builder.Append("!include -teardown .").Append(pagePathName).Append("\n");
            }
        }
    }
    pageData.SetContent(builder.ToString());
    return pageData.GetHtml();
}
```

Function is too large!

# Problem

High abstraction level: WikiPage

```
public static string TestableHtml(PageData pageData, boolean includeSuiteSetup)
{
    WikiPage wikiPage = pageData.GetWikiPage();
    StringBuilder builder = new StringBuilder();
    if (pageData.HasAttribute("Test")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup = PageCrawlerImpl.GetInheritedPage(SuiteResponsePageData);
            if (suiteSetup != null) {
                WikiPagePath pagePath = suiteSetup.getPageCrawler().GetFullPath(pageData.getWikiPagePath());
                String pagePathName = PathParser.Render(pagePath);
                builder.Append("!include -setup ").Append(pagePathName).Append("\n");
            }
        }
    }
    ...
}
```

Deeply nested conditions  
(repeated in the last 15 lines!)

Low abstraction level: StringBuilder

Multiple page resolution

And this is just the first 11 lines of 38!

# Functions should be small

---

- Functions should hardly ever be 20 lines long
- Best if functions are 2-4 lines long
- Functions should be easy to read
- Functions should tell a story

# Blocks and indenting

---

- Blocks within if statements, else statements, while statements, try-catch statements, and so on should be one line long
  - probably that line should be a function call
- This also adds documentary value
  - the function called within the block can have a nicely descriptive name
- Functions should not be large enough to hold nested structures
  - the indent level of a function should not be greater than one or two
- This makes the functions easier to read and understand

# Functions should do one thing

---

- *Functions should do one thing. They should do it well. They should do it only.*
- What is one thing?
  - If a function does only those steps that are one level below the stated name of the function, then the function is doing one thing.
  - Exception handling, logging, concurrency, etc. are already one abstraction level, they are already one thing!
- What is more than one thing?
  - If you can extract another function from it with a name that is not merely a restatement of its implementation
  - If the function can be divided into sections
  - If there are multiple abstraction levels within the function

# One abstraction level per function

---

- The statements within a function should all be at the same level of abstraction
- Mixing levels of abstraction within a function is always confusing
- Readers may not be able to tell whether a particular expression is an essential concept or a detail
- Once details are mixed with essential concepts, more and more details tend to accrete within the function

# Stepdown rule

---

- We want the code to read like a top-down narrative
- We want to be able to read the program as though it were a set of TO paragraphs:
  - *To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.*
    - *To include the setups, we include the suite setup if this is a suite, then we include the regular setup.*
    - *To include the suite setup, we search the parent hierarchy for the “SuiteSetUp” page and add an include statement with the path of that page.*
    - *To search the parent ...*
- Order of the functions:
  - every function should be followed by those at the next level of abstraction
  - so that we can read the program, descending one level of abstraction at a time, as we read down the list of functions
- It is an effective technique for keeping the abstraction level consistent



# Solution

```
public class SetupTeardownIncluder {  
    private PageData pageData;  
    private bool isSuite;  
    private WikiPage testPage;  
    private StringBuilder newPageContent;  
    private PageCrawler pageCrawler;
```

Function lengths: 2-4 lines

```
    public static string Render(PageData pageData) {  
        return Render(pageData, false);  
    }
```

Static factory methods for different cases

```
    public static string Render(PageData pageData, bool isSuite)  
    {  
        return new SetupTeardownIncluder(pageData).Render(isSuite);  
    }
```

```
    private SetupTeardownIncluder(PageData pageData) {  
        this.pageData = pageData;  
        testPage = pageData.GetWikiPage();  
        pageCrawler = testPage.GetPageCrawler();  
        newPageContent = new StringBuilder();  
    }
```

Private constructor

# Solution

```
public class SetupTeardownIncluder {
```

Function lengths: 2-4 lines

```
...
```

```
private string Render(bool isSuite) {
```

```
    this.isSuite = isSuite;
```

```
    if (IsTestPage())
```

```
        IncludeSetupAndTeardownPages();
```

```
    return pageData.GetHtml();
```

```
}
```

One line long "if"

Stepdown rule:

lowering abstraction levels

```
private boolean IsTestPage() {
```

```
    return pageData.hasAttribute("Test");
```

```
}
```

```
private void IncludeSetupAndTeardownPages() {
```

```
    IncludeSetupPages();
```

```
    IncludePageContent();
```

```
    IncludeTeardownPages();
```

```
    UpdatePageContent();
```

```
}
```

Stepdown rule:

lowering abstraction levels

```
...
```

# Solution

```
public class SetupTeardownIncluder {
```

Function lengths: 2-4 lines

```
...
```

```
private void IncludePageContent() {  
    newPageContent.Append(pageData.GetContent());  
}
```

```
private void IncludeTeardownPages() {  
    IncludeTeardownPage();  
    if (isSuite)  
        IncludeSuiteTeardownPage();  
}
```

One line long "if"

```
private void IncludeTeardownPage() {  
    Include("TearDown", "-teardown");  
}
```

Stepdown rule:  
lowering abstraction levels

```
private void IncludeSuiteTeardownPage() {  
    Include(SuiteResponder.SuiteTearDownName, "-teardown");  
}
```

```
private void UpdatePageContent() {  
    pageData.SetContent(newPageContent.ToString());  
}
```

# Solution

```
public class SetupTeardownIncluder {  
    ...  
    private void Include(string pageName, string arg) {  
        WikiPage inheritedPage = FindInheritedPage(pageName);  
        if (inheritedPage != null) {  
            string pagePathName = GetPathNameForPage(inheritedPage);  
            BuildIncludeDirective(pagePathName, arg);  
        }  
    }  
    private WikiPage FindInheritedPage(string pageName) {  
        return PageCrawlerImpl.GetInheritedPage(pageName, testPage);  
    }  
    private string GetPathNameForPage(WikiPage page) {  
        WikiPagePath pagePath = pageCrawler.GetFullPath(page);  
        return PathParser.Render(pagePath);  
    }  
    private void BuildIncludeDirective(string pagePathName, string arg) {  
        newPageContent.Append("\n!Include ").Append(arg).Append(" ").Append  
    }  
}
```

Function lengths: 2-4 lines

Almost one line long "if"

Stepdown rule:  
lowering abstraction levels

# Problem

---

`void Transform(Point p)`  Mutation: transform in place

`string a = ...`

`string b = ...`

`AssertEquals(a, b);` 

Which is the expected value?

Which is the actual value?

`Save("Document.txt", false, true, false, false)` 

What do all these options mean?

# Function arguments

---

- Functions should have as few arguments as possible
  - the ideal number of arguments for a function is zero
  - next comes one
  - followed closely by two
  - three arguments should be avoided where possible
- More than three arguments requires very special justification
- Too many arguments usually means that abstraction levels are mixed
- Arguments are even harder from a testing point of view: testing all the combinations

# One argument

- Two common reasons for supplying a single argument:
  - asking a question about that argument
  - operating on that argument, transforming it into something else and *returning it*
- Functions with single arguments may be events
- Passing a boolean value (flag) into a function is ugly
  - it complicates the signature of the method
  - indicating that the function does more than one thing
- Examples:
  - `boolean FileExists("MyFile")` – question
  - `InputStream FileOpen("MyFile")` – transformation
  - `void IncludeSetupPageInto(StringBuffer pageText)` – mutation
    - should be: `void IncludeSetupPage()`, and use 'pageText' as a field
  - `void Transform(StringBuffer out)` – mutation
    - should be: `StringBuffer Transform(StringBuffer in)`, even if it just returns 'in'
  - `void Render(bool isSuite)` – flag
    - should be: `void RenderForSuite()` and `void RenderForSingleTest()`

# Two arguments

---

- A function with two arguments is harder to understand than a function with one argument
- Two arguments are appropriate when they are ordered components of a single value
- Two arguments are problematic if they have no natural ordering
- Two arguments aren't evil, but they come at a cost
  - consider turning them into single arguments
  - e.g. making one of them a field
- Examples:
  - `new Point(0,0)` – OK: ordered components of a single value
  - `AssertEquals(expected, actual)` – no natural ordering: requires practice to learn
    - should be: `AssertExpectedEqualsActual(expected, actual)`
  - `WriteField(outputStream, name)` – probably OK
    - better: `WriteField(name)` and making 'outputStream' a field
  - `string.Format(string format, params object[] args)` – OK, still counts as two arguments, since the items of args are treated identically, as if args were a normal array or list



# Three or more arguments

- Functions that take three arguments are significantly harder to understand than functions with fewer arguments
- There are issues of
  - ordering: which parameter is which
  - pausing: thinking about the meaning of each parameter
  - ignoring: we have to include common parameters (e.g. stream)
  - testing: testing all the combinations of values is hard
- Consider wrapping the arguments into a class of their own
  - they probably deserve a name of their own
- Examples:
  - `AssertEquals(message, expected, actual)` –  
Can you remember which is which?  
What if you extend an `AssertEquals(expected, actual)` call?
  - `AssertEquals(1.0, amount, 0.001)` –  
Probably worth the thinking: equality of floating point values
  - `Save("Document.txt", false, true, false, false)` –  
What do all these options mean?  
Should be: `Save("Document.txt", saveOptions)`

# Naming functions

---

- Functions should have zero, one or two arguments
- The name can explain the intent and the order of the arguments
- Name of a function with zero arguments:
  - a verb can be nice, but can be longer
  - e.g. `Save()`, `Quit()`, `Transform()`, `IncludePage()`, `UpdatePageContent()`
- Name of a function with one argument:
  - a verb-noun pair can be nice by encoding the name of the argument into the function name
  - e.g. `WriteField(name)` instead of just `Write(name)`
- Name of a function with two arguments:
  - encode the names of the arguments into the function name
  - e.g. `AssertExpectedEqualsActual(expected, actual)` instead of just `AssertEquals(expected, actual)`

# Solution

---

`Point Transform(Point p)` ← Transformed point as a result

`string a = ...`

`string b = ...`

`AssertExpectedEqualsActual(a, b);`

← We know which is which.

`var saveOptions = new SaveOptions();`

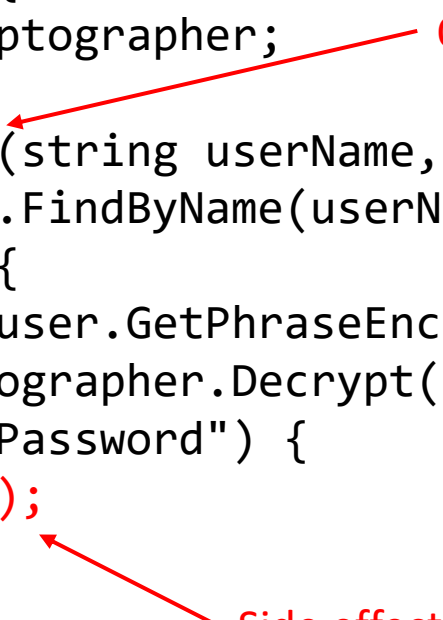
`saveOptions.AddUtf8Bom = true;`

`Save("Document.txt", saveOptions)`


← We know what the options mean.

# Problem

```
public class UserValidator {  
    private Cryptographer cryptographer;  
  
    public bool CheckPassword(string userName, string password) {  
        User user = UserGateway.FindByName(userName);  
        if (user != User.None) {  
            string codedPhrase = user.GetPhraseEncodedByPassword();  
            string phrase = cryptographer.Decrypt(codedPhrase, password);  
            if (phrase == "Valid Password") {  
                Session.Initialize();  
                return true;  
            }  
        }  
        return false;  
    }  
}
```



```
if (Set("username", "alice")) {  
    ...  
}
```



# Have no side effects

---

- Your function promises to do one thing
- But with side effects it also does other hidden things
  - unexpected changes to the variables of its own class
  - changes to the parameters passed
  - changes to system globals
- Side effects can create temporal coupling
  - the function can only be called at certain times
- Side effects can also cause concurrency issues

# Command and query separation

---

- Functions should either do something or answer something, but not both
  - either change the state of an object
  - or return some information about that object
- Doing both often leads to confusion
- Exceptions:
  - atomic operations for thread-safety
    - e.g. `TestAndSet(flag)`, `Interlocked.CompareExchange()`

# Solution

```
public class UserValidator {  
    private Cryptographer cryptographer;  
  
    public bool CheckPassword(string userName, string password) {  
        User user = UserGateway.FindByName(userName);  
        if (user != User.None) {  
            string codedPhrase = user.GetPhraseEncodedByPassword();  
            string phrase = cryptographer.Decrypt(codedPhrase, password);  
            if (phrase == "Valid Password") {  
                Session.Initialize();  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

Query function

Remove this line, do it somewhere else.

```
if (AttributeExists("username")) {  
    SetAttribute("username", "alice");  
}
```

Query

Command

# How to write functions like this?

---

- It is usually easier to write longer functions
  - with a lot of indenting, loops, conditions, etc.
  - with a lot of arguments
  - with duplicated code
- You should have unit tests to cover all the lines of these functions
- Then you can refine the code while keeping the tests passing
  - extracting new functions or new classes
  - changing names
  - eliminating duplication
  - shrinking methods
  - reordering methods



# Writing and omitting comments

---

# Comments

---

- Proper use of comments is to compensate for our *failure to express ourselves in code*
- Well-placed comments can be very useful
- But meaningless or even lying comments can be very damaging
  - code changes and evolves, chunks are moving around
  - comments don't always follow the change
- Inaccurate comments are far worse than no comments at all
  - they are misleading
  - they contain old rules that need not or should not be followed any longer
- The only source of accurate information is the *code*
- Don't comment bad code – rewrite it!

# Problem

---

```
public void LoadProperties()
{
    try
    {
        string propertiesPath =
            Path.Combine(propertiesLocation, PropertyFileName);
        StreamReader propertiesStream = new StreamReader(propertiesPath);
        loadedProperties.Load(propertiesStream);
    }
    catch(IOException e)
    {
        // No properties files means all defaults are loaded
    }
}
```

What does this comment mean?

Apparently, if there is an IOException, then there is no properties file.

But who loads the defaults? Are they already loaded?

Or loadedProperties.Load() loads them?

Or they should have been loaded in the catch block, but have been forgotten?

# Bad comments: Mumbling

---

- Don't just write a comment because you feel you should or it is required
- If you decide to write a comment make sure it is the best comment you can write
- The comment should be meaningful on its own within its context
  - any comment that forces you to look in another module for the meaning of that comment has failed to communicate to you

# Problem

---

The comment just repeats what the code does. It is redundant.



```
// Utility method that returns when this.closed is true.  
// Throws an exception if the timeout is reached.  
public void WaitForClose(long timeoutMillis)  
{  
    if (!closed)  
    {  
        Wait(timeoutMillis);  
        if (!closed)  
            throw new Exception("Sender could not be closed");  
    }  
}
```

# Bad comments: Redundant comments

---

- Redundant comment:
  - it is not more informative than the code
  - it is not easier to read than the code
  - it may be less precise than the code and may mislead the reader
- Don't write redundant comment
- Instead:
  - write comments that justify the code, or provide intent or rationale
  - or just omit the comment if the code explains itself

# Problem

---

The comment is also subtly misleading:

The method does not return *when* this.closed becomes true.

It returns *if* this.closed is true; otherwise, it waits for a time-out and then throws an exception if this.closed is still not true.



```
// Utility method that returns when this.closed is true.  
// Throws an exception if the timeout is reached.  
public void WaitForClose(long timeoutMillis)  
{  
    if (!closed)  
    {  
        Wait(timeoutMillis);  
        if (!closed)  
            throw new Exception("Sender could not be closed");  
    }  
}
```

# Bad comments: Misleading comments

---

- Sometimes a comment written with all the best intentions may be inaccurate
  - it does not explain precisely what the code does
  - so it can be slightly misleading
  - another programmer calls the function based on the comment and later he can debug why it behaves differently than expected
- Don't write redundant comment, it can be misleading
- Only the code is accurate enough



# Problem

---

Probably mandatory comment with no added value.



```
/// <summary>
/// Adds a CD
/// </summary>
/// <param name="title">The title of the CD</param>
/// <param name="author">The author of the CD</param>
/// <param name="tracks">The number of tracks on the CD</param>
/// <param name="durationInMinutes">The duration of the CD in minutes</param>
public void AddCD(string title, string author,
    int tracks, int durationInMinutes)
{
    CD cd = new CD();
    cd.Title = title;
    cd.Author = author;
    cd.Tracks = tracks;
    cd.Duration = duration;
    cdList.Add(cd);
}
```

# Bad comments: Mandatory comments

---

- It is a bad idea to have rules like:
  - every function must have a documentation comment
  - every variable must have a comment
- Comments like this just clutter up the code, propagate lies, and lend to general confusion
- Don't make comments mandatory for everything

# Problem

```
public class AnnualDateRule
{
    /// <summary>
    /// Default constructor ← No, really?
    /// </summary>
    protected AnnualDateRule()
    {
    }

    /// <summary>
    /// The day of the month. ← I would never have guessed...
    /// </summary>
    private int dayOfMonth;

    /// <summary>
    /// Returns the day of the month. ← Just noise...
    /// </summary>
    /// <returns>the day of the month</returns>
    public int DayOfMonth
    {
        get { return dayOfMonth; }
    }
}
```

# Problem

---

// The name.  Noise...

private string name;

// The version.

private string version;

// The licenceName.

private string licenceName;

// The version.  Scary noise!!!

private string info;

# Bad comments: Noise comments

---

- Noise comments just restate the obvious and provide no new information
- These comments are so noisy that we learn to ignore them
- Eventually they begin to lie as the code around them changes
- Scary noise:
  - If authors aren't paying attention when comments are written (or pasted), why should readers be expected to profit from them?
- Don't write noise comments, especially if the code explains itself

# Problem

```
private void StartSending()
{
    try
    {
        DoSending();
    }
    catch (SocketException e)
    {
        // normal. someone stopped the request.
    }
    catch (Exception e)
    {
        try
        {
            response.Add(ErrorResponder.MakeExceptionString(e));
            response.CloseAll();
        }
        catch (Exception e1)
        {
            //Give me a break!
        }
    }
}
```

OK

The developer was angry.

# Bad comments: Angry comments

---

- Angry comments are also just noise, they provide no new information
- Frustration can be resolved by improving the structure of the code
- Replace the temptation to create noise with the determination to clean your code

# Problem

---

Could we rewrite the code so that we can omit the comment?



```
// does the module from the global list <mod> depend on the  
// subsystem we are part of?  
if (smodule.GetDependSubsystems().Contains(subSysMod.GetSubSystem()))
```



## Bad comments: Comment instead of a function or variable

---

- Don't write a comment if you can express the same thing in code
- Write expressive code instead

# Solution

---

```
List moduleDependees = smodule.GetDependSubsystems();  
string ourSubSystem = subSysMod.GetSubSystem();  
if (moduleDependees.Gontains(ourSubSystem))
```



The code is easy to understand, no need for comments

# Problem

```
public class ArrayBuilder<T>
{
```

```
    // Fields //////////////////////////////////////
```

```
    private int count;
    private T[] items;
```

```
    // Constructors //////////////////////////////////////
```

```
    public ArrayBuilder()
    { ... }
```

```
    // Properties //////////////////////////////////////
```

```
    public int Count { get { return this.count; } }
```

```
    // Methods //////////////////////////////////////
```

```
    public void Add(T item)
    { ... }
```

```
    public void AddRange(IEnumerable<T> items)
    { ... }
```

```
}
```

Noise



# Bad comments: Banner comments

---

- Banner or position marker comments add noise to the code
- If you overuse banners, they'll fall into the background noise and be ignored
- Use them very sparingly, and only when the benefit is significant

# Problem

```
static void Main(string[] args)
{
    string line;
    int lineCount = 0;
    int charCount = 0;
    int wordCount = 0;
    try
    {
        while ((line = Console.ReadLine()) != null)
        {
            lineCount++;
            charCount += line.Length;
            string[] words = line.Split(' ', '\t');
            wordCount += words.Length;
        } //while ← Closing while
        Console.WriteLine("wordCount = " + wordCount);
        Console.WriteLine("lineCount = " + lineCount);
        Console.WriteLine("charCount = " + charCount);
    } // try ← Closing try
    catch (IOException e)
    {
        Console.Error.WriteLine("Error:" + e.Message);
    } //catch ← Closing catch
} //main ← Closing main
```

# Bad comments: Closing brace comments

---

- Closing brace comments might make sense for long functions with deeply nested structures
- But functions should not be long and have deeply nested structures
- Make sure to format the code properly
- If you find yourself wanting to mark your closing braces, try to shorten your functions instead

# Problem

---

```
InputStreamResponse response = new InputStreamResponse();  
response.SetBody(formatter.GetResultStream(), formatter.GetByteCount());  
// InputStream resultsStream = formatter.GetResultStream();  
// StreamReader reader = new StreamReader(resultsStream);  
// response.SetContent(reader.Read(formatter.GetByteCount()));
```



Commented-out code: why is it there?

# Bad comments: Commented-out code

---

- Commented-out code always seems important
  - Are they reminders?
  - Do they indicate an imminent change?
  - Were they just left there years ago and have been forgotten?
- Others who see it won't have the courage to delete it
- So commented-out code just gathers throughout the code
- Don't leave commented-out code in the source
- Use a source control system
  - you can safely delete the code
  - and you can restore it anytime



# Problem

---

- \* Changes (from 11-Oct-2001)
- \* -----
- \* 11-Oct-2001 : Re-organised the class and moved it to new package
- \* com.jrefinery.date (DG);
- \* 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
- \* class (DG);
- \* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
- \* class is gone (DG); Changed getPreviousDayOfWeek(),
- \* getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
- \* bugs (DG);
- \* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
- \* 29-May-2002 : Moved the month constants into a separate interface
- \* (MonthConstants) (DG);
- \* 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
- \* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
- \* 13-Mar-2003 : Implemented Serializable (DG);
- \* 29-May-2003 : Fixed bug in addMonths method (DG);
- \* 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);
- \* 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);



Journal comment: why is it there?

# Bad comments: Journal comments

---

- Long ago there was a good reason to create and maintain these log entries at the start of every file
- But today we have source control systems that maintain these entries
- Don't write journal comments, they just add more noise to the code

# Problem

---

```
/* Added by Rick */ ← Byline comment: why is it there?  
public class Date  
{  
}
```

# Bad comments: Attributions and bylines

---

- You might think that such comments would be useful in order to help others know who to talk to about the code
- But source control systems are much better at remembering who added what, when
- There is no need to pollute the code with little bylines

# Problem

```
/**
 * Task to run fit tests.
 * This task runs fitnesse tests and publishes the results.
 * <p/>
 * <pre>
 * Usage:
 * <code><taskdef name="execute-fitness-tests"
 * classname="fitnesse.ant.ExecuteFitnesseTestsTask"
 * classpathref="classpath" />
 * OR
 * <code><taskdef classpathref="classpath"
 * resource="tasks.properties" />
 * <p/>
 * <code><execute-fitness-tests
 * suitepage="FitNesse.SuiteAcceptanceTests"
 * fitnesseport="8082"
 * resultsdir="${results.dir}"
 * resultshtmlpage="fit-results.html"
 * classpathref="classpath" />
 * </code>
 */
```

Hard to read

# Bad comments: HTML comments

---


- HTML makes the comments hard to read in the one place where they should be easy to read: the editor/IDE
- It should be the responsibility of the documentation generator tool, and not the programmer to add HTML markers
- Unfortunately, Visual Studio prefers HTML comments

# Problem

---

Nonlocal information.  
Redundant.  
The function has no control over it.  
Will it remain accurate?

```
/// <summary>  
/// Port on which fitnessse would run. Defaults to <b>8082</b>.  
/// </summary>  
/// <param name="fitnesssePort"></param>  
public void SetFitnesssePort(int fitnesssePort)  
{  
    this.fitnesssePort = fitnesssePort;  
}
```



# Bad comments: Nonlocal Information

---

- Don't offer systemwide information in the context of a local comment
  - it is redundant
  - it is not describing the function, but some other, far distant part of the system
  - there is no guarantee that this comment will be changed when the systemwide information is changed
- If you must write a comment, then make sure it describes the code it appears near



# Problem

---

```
/*
```

Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.

XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

```
*/  
public string ToXml()  
{  
    ...  
}
```



Too much and irrelevant information

# Bad comments: Too much information

---

- Don't put interesting historical discussions or irrelevant descriptions of details into your comments
- Include only the relevant details
- Include a reference to the standard, if necessary
- The details of the standard are usually relevant only if you want to directly implement and use the standard
  - exchanging XML files is indirect use

# Problem

---

```
/*  
 * start with an array that is big enough to hold all the pixels  
 * (plus filter bytes), and an extra 200 bytes for header info  
 */  
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

Why 200?

Does this relate to +1? Or \*3? Or both?

Why +1?

Why \*3?

# Bad comments: Inobvious connection

---

- The connection between a comment and the code it describes should be obvious
- The purpose of a comment is to explain code that does not explain itself
- It is bad when a comment needs its own explanation

# Good comments: Informative comments

- It is sometimes useful to provide basic information with a comment

- Example:

```
// Returns an instance of the Responder being tested.  
protected abstract Responder responderInstance();
```

- but it is usually better to use the name of the function to convey the information where possible:

```
protected abstract Responder responderBeingTested();
```

- A better example:

```
// format matched kk:mm:ss EEE, MMM dd, yyyy  
Pattern timeMatcher = Pattern.Compile(  
    "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

- easier to read than trying to understand the regular expression
- but make sure the comment remains in sync with the expression

# Good comments: Explanation of intent

---

- A comment is valuable if it provides the intent behind a decision
  - it explains why the code works the way it is
  - it helps us to adhere to the original design decisions
- **This is the most important reason to write a comment**
- Example:

```
public void TestConcurrentAddWidgets() {  
    WidgetBuilder widgetBuilder = new WidgetBuilder(typeof(BoldWidget));  
    string text = ""'bold text'"";  
    ParentWidget parent = new BoldWidget(new MockWidgetRoot(), text);  
    FailFlag failFlag = new FailFlag();  
  
    //This is our best attempt to get a race condition  
    //by creating large number of threads.  
    for (int i = 0; i < 25000; i++) {  
        WidgetBuilderThread widgetBuilderThread =  
            new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);  
        Thread thread = new Thread(widgetBuilderThread);  
        thread.Start();  
    }  
    Debug.Assert(!failFlag.IsSet);  
}
```

# Good comments: Clarification

- Sometimes it is just helpful to translate the meaning of some obscure code into something that's readable
  - in general it is better to find a way to make the code clear in its own right
  - but when its part of the standard library, or in code that you cannot alter, then a helpful clarifying comment can be useful
  - make sure the clarification comment is correct
- Examples:

```
Debug.Assert(a.CompareTo(a) == 0); // a == a
Debug.Assert(a.CompareTo(b) != 0); // a != b
Debug.Assert(aa.CompareTo(ab) == -1); // aa < ab
```
- There is a substantial risk, of course, that a clarifying comment is (or later becomes) incorrect
- This explains both why the clarification is necessary and why it's risky

# Good comments: Warning of consequences

---

- Sometimes it is useful to warn other programmers about certain consequences
- Examples:

```
public static SimpleDateFormat MakeStandardHttpDateFormat()  
{  
    //SimpleDateFormat is not thread safe,  
    //so we need to create each instance independently.  
    SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z");  
    df.SetTimeZone(TimeZone.getTimeZone("GMT"));  
    return df;  
}
```

```
// Don't run unless you have some time to kill.  
public void _testWithReallyBigFile()  
{  
    WriteLinesToFile(10000000);  
    Debug.Assert(bytesSent > 10000000000);  
}
```



# Good comments: TODO comments

---

- TODOs are jobs that the programmer thinks should be done, but for some reason can't do at the moment
- It is sometimes reasonable to leave “To do” notes in the form of `//TODO` comments
- TODO comments are reminders:
  - implement a feature later
  - delete a deprecated feature
  - look at a problem later
  - etc.
- Example: 

```
//TODO-MdM these are not needed
// We expect this to go away when we do the checkout model
protected VersionInfo MakeVersion() {
    return null;
}
```
- But TODO is not an excuse to leave bad code in the system!
- Good IDEs provide special features to locate all the TODO comments, so it's not likely that they will get lost

# Good comments: Amplification

---

- A comment may be used to amplify the importance of something that may otherwise seem inconsequential
- Example:

```
string listItemContent = match.Group(3).Trim();  
// the trim is real important. It removes the starting  
// spaces that could cause the item to be recognized  
// as another list.  
new ListItemWidget(this, listItemContent, this.level + 1);  
return BuildList(text.Substring(match.End()));
```

# Good comments: Public API documentation

---

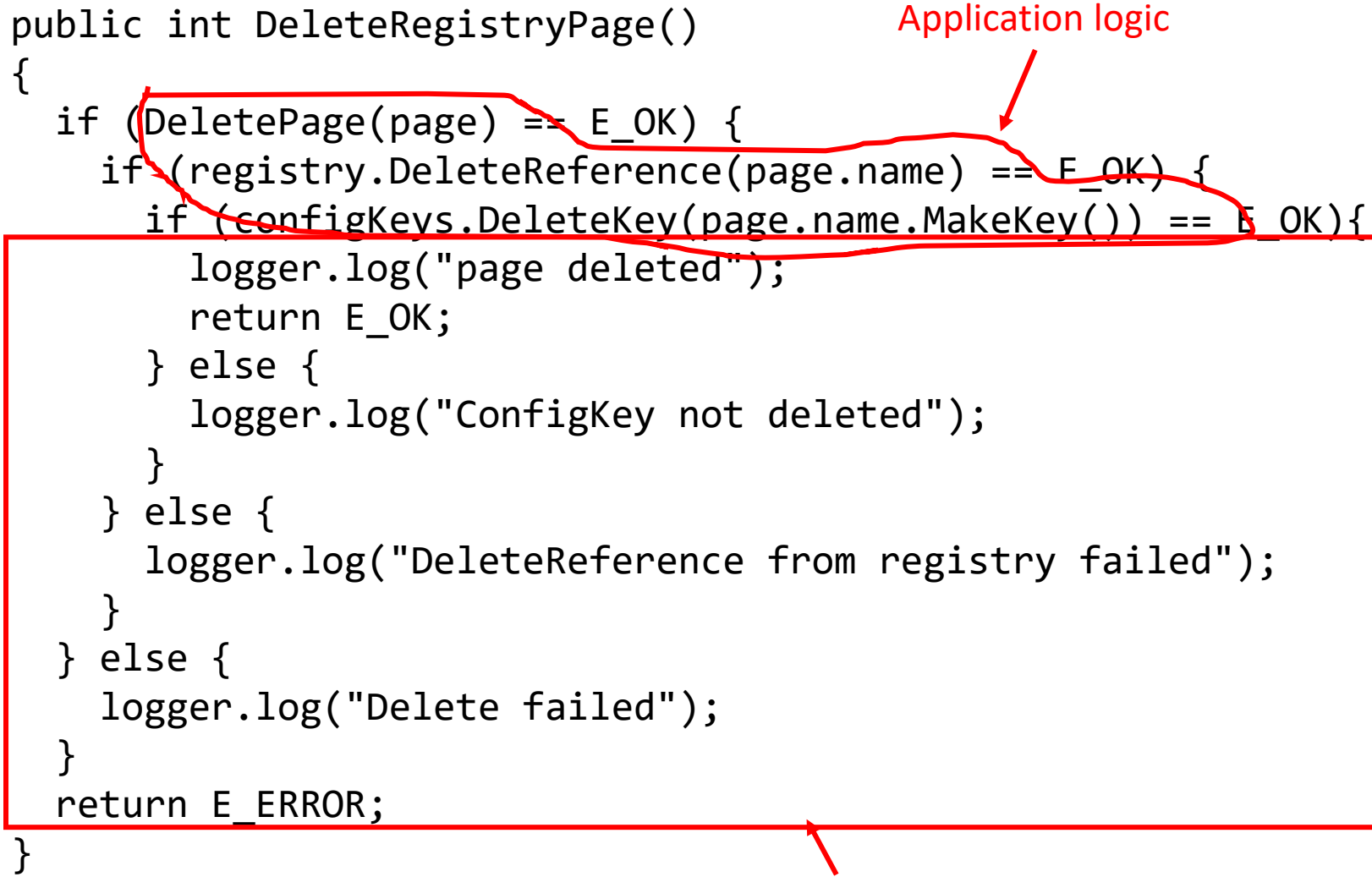
- A well-described public API is very helpful
- But make sure the comments are meaningful, always up-to-date and not misleading
  - include pre- and post-conditions, exceptions, etc.
  - include samples
- The public API must be documented
  - the user of the library does not see (and does not want to see) your source code
- The inner implementation of the library can follow the clean code principles
  - documentation comments are not generally useful here

# Defining and handling exceptions

---

# Problem

```
public int DeleteRegistryPage()
{
    if (DeletePage(page) == E_OK) {
        if (registry.DeleteReference(page.name) == E_OK) {
            if (configKeys.DeleteKey(page.name.MakeKey()) == E_OK){
                logger.log("page deleted");
                return E_OK;
            } else {
                logger.log("ConfigKey not deleted");
            }
        } else {
            logger.log("DeleteReference from registry failed");
        }
    } else {
        logger.log("Delete failed");
    }
    return E_ERROR;
}
```



Error handling

# Use exceptions rather than error codes

---

- Using error codes clutters the caller
- Error codes lead to deeply nested if statements, and the caller must deal with the error immediately
  - it is also easy to forget to handle the error
- If you use exceptions then the error processing code can be separated from normal execution
  - the calling code is cleaner
  - its logic is not obscured by error handling
- Functions should do one thing
  - error handling is one thing
  - thus, a function that handles errors should do nothing else
  - extract the bodies of the try and catch blocks out into functions of their own

# Error handling

---

- Error handling is important
- But if it obscures logic, it is wrong
  - too many catch blocks
  - catch blocks containing application logic
- Write try-catch-finally first
  - try defines a scope
  - catch and finally must leave the system in a consistent state
- Define the normal flow in the body of the try
  - don't put application logic in a catch block

# Solution

---

```
public void DeleteRegistryPage(Page page)
{
    try
    {
        TryDeleteRegistryPage(page);
    }
    catch(RegistryException e)
    {
        logger.Log(e);
    }
}
```

← Error handling

```
private void TryDeleteRegistryPage(Page page)
{
    DeletePage(page);
    registry.DeleteReference(page.name);
    configKeys.DeleteKey(page.name.MakeKey());
}
```

← Application logic



# Provide context with exceptions

---

- File and location of the error
- Reason of the error
  - convert it to a meaningful error message
  - include the operation that failed
  - type of the failure
- Resolution of the error
  - this helps the programmer to correct the error
  - e.g. what should be configured, where can be more information found, etc.
- Provide enough information for logging to reconstruct the problem if necessary

# Problem

---

```
ACMEPort port = new ACMEPort(12);
try {
    port.open();
} catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
    logger.log("Unlock exception", e);
} catch (GMXError e) {
    reportPortError(e);
    logger.log("Device response exception");
} finally {
    ...
}
```



A lot of catch blocks,  
but the same error handling logic

# Use unchecked (runtime) exceptions

---

- Checked exceptions in Java:
  - either must be handled by a catch or must be declared in the method signature
  - the client can reasonably be expected to recover from a checked exception
    - e.g. file not found
- Unchecked (runtime) exceptions in Java:
  - the client cannot do anything to recover from an unchecked exception
    - e.g. division by zero, null pointer
- Problem with checked exceptions:
  - they violate OCP
    - all methods in the call hierarchy must declare the exception
    - any change will result in changing all the methods
  - they violate LSP
    - incompatibility of an interface and its implementation: if an interface does not declare a checked exception that cannot be handled by the implementation of the interface
- Checked exceptions can sometimes be useful if you are writing a critical library: you must catch them
- But in general application development the dependency costs outweigh the benefits
- C# does not have checked exceptions

# Define exception classes in terms of a caller's needs

---

- Classify exceptions based on how they are caught
- Exceptions signal that something went wrong
- The low-level details are unimportant
- Don't force the caller to write many catch-blocks
  - their bodies will be the same
- If an external library forces too many exceptions on you, wrap it with your API
- It is always a good idea to wrap third-party APIs
  - minimizes dependency on it
  - easier to move to another library
  - easier to mock it for testing
  - you are not bound to the vendor's design choices: you can have your own API

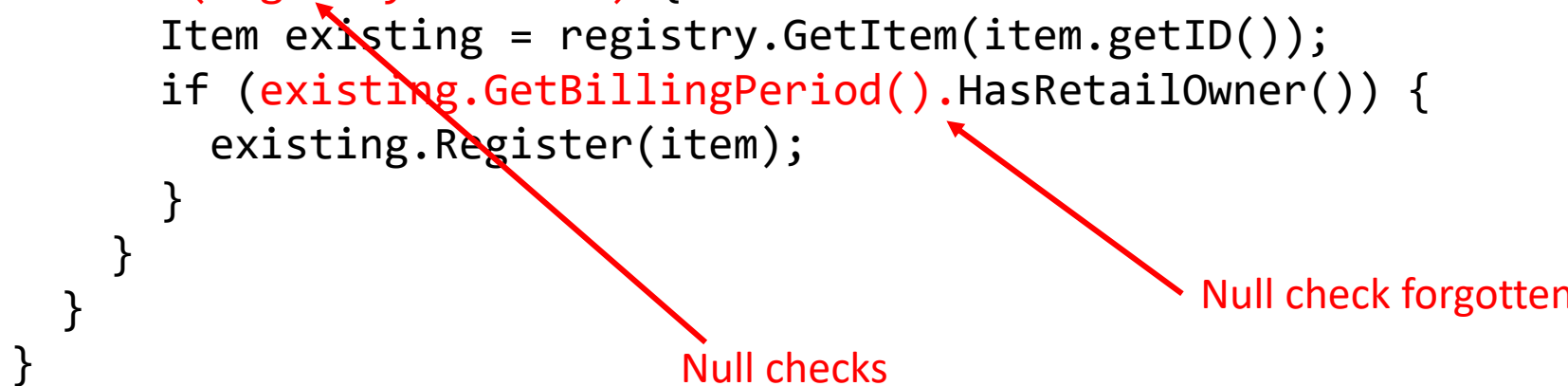
# Solution

---

```
LocalPort port = new LocalPort(12);
try {
    port.open();
} catch (PortDeviceFailure e) {
    reportError(e);
    logger.log(e.getMessage(), e);
} finally {
    ...
}
```

# Problems

```
public void RegisterItem(Item item) {  
    if (item != null) {  
        ItemRegistry registry = persistentStore.GetItemRegistry();  
        if (registry != null) {  
            Item existing = registry.GetItem(item.getID());  
            if (existing.GetBillingPeriod().HasRetailOwner()) {  
                existing.Register(item);  
            }  
        }  
    }  
}
```



Null checks

Null check forgotten

```
List<Employee> employees = GetEmployees();  
if (employees != null) {  
    foreach (Employee e in employees) {  
        totalPay += e.GetPay();  
    }  
}
```

# Don't return null

---

- If something is wrong, throw an exception
- If a collection is to be returned, return an empty collection
- If the method performs some operation, return the neutral element of the operation
- Return a special case object, e.g. `NullObject`
- The problem of returning null:
  - the client code will be cluttered with null-checks
  - if a null check is missing, there will be a `NullPointerException` from very deep

# Solution

---

```
public void RegisterItem(Item item) {  
    if (item == null) throw new ArgumentNullException(nameof(item));  
    ItemRegistry registry = persistentStore.GetItemRegistry();  
    Item existing = registry.GetItem(item.getID());  
    if (existing.GetBillingPeriod().HasRetailOwner()) {  
        existing.Register(item);  
    }  
}
```

No null checks are necessary



```
List<Employee> employees = GetEmployees();  
foreach (Employee e in employees) {  
    totalPay += e.GetPay();  
}
```



# Problem

---

```
public class MetricsCalculator
{
    public double Projection(Point p1, Point p2) {
        return (p2.x - p1.x) * 1.5;
    }
    ...
}
```

```
calculator.Projection(null, new Point(12, 13));
```



Passing null will result in `NullPointerException`

# Don't pass null

---

- Don't pass null unless the API expects it
- If the API is not prepared, `NullReferenceExceptions` will fly
- If you are writing the API check the input parameters, and throw `ArgumentNullException`
- If the programming language allows it, forbid undesired null values in your API
  - compile time checking
  - (reference types will be non-nullable by default in C# 8)

# Good example

---

```
public class MetricsCalculator
{
    public double Projection(Point p1, Point p2) {
        if (p1 == null) throw new ArgumentNullException(nameof(p1));
        if (p2 == null) throw new ArgumentNullException(nameof(p2));
        return (p2.x - p1.x) * 1.5;
    }
}
```



We will get an `ArgumentNullException` immediately,  
instead of a `NullReferenceException` later from somewhere deep in the code.

# Objects and data structures

---

# Data structures

---

- Main focus is data
  - data fields
- No behavior
- Typical OO implementation:
  - private fields
  - public properties / getters-setters
    - or just readonly properties / getters if data is immutable
- Behavior in separate functions

# Objects

---

- Main focus is behavior
- Data representation is hidden
- Public interface allows only to manipulate the essence of the data
  - e.g. the x and y coordinates of a point must be set together
- Don't just automatically add getters-setters, think about the abstract representation of the data hidden

# Data/object anti-symmetry

---

Procedural code	Object-oriented code
Code using data structures	Code focusing on behavior
Behavior in separate functions	Data representation hidden
Easy to add new functions without changing existing data structures	Easy to add new classes without changing existing functions
Hard to add new data structures because all the functions must change	Hard to add new functions because all the classes in the hierarchy must change

# Data/object anti-symmetry

---

- Things that are hard for OO are easy for procedures
- Things that are hard for procedures are easy for OO
- In a complex system:
  - there are cases when adding data types is more frequent than adding functions
    - use OO
  - there are cases when adding functions is more frequent than adding data types
    - use procedural code



# Law of Demeter (LoD)

---

- Methods should only be called on objects directly known
- “Talk to friends not to strangers”
- “Train wreck”:
  - `ctx.GetOptions().GetTmpDir().GetAbsolutePath()`
- LoD is only violated if the train of calls are performed on objects
  - then the ctx should provide a wrapper method to provide the same result
    - explosion in the number of methods
  - or it should be examined, why the result is needed
  - for example, if a new temp file is to be created, ctx should provide that instead
- However, if the links in the chain are data structures, LoD does not apply

# Entities and data transfer objects

---

- Classes with only data, no behavior
- Private fields with public properties/accessors
- Useful in various technologies where behavior is intentionally separated from the data:
  - ORM
  - MVC
  - data transfer through network
- Make sure not to create hybrids: data structure with some business logic or behavior

# Design considerations

---

- Sometimes it is better to separate behavior from data (procedural)
  - when new behavior is frequently added but the data structure is stable
    - database + business logic
  - when behavior is attachable or optional
    - visitor, strategy
- Sometimes it is better to keep related data and behavior together (OO)
  - when internal data representation can change
  - when new functions are rarely added
- Both of them are perfectly OK
  - just make sure to choose the approach that is the best for the job at hand

# Summary

---

# Summary

---

- Bad code:
  - hard to maintain
- Clean code:
  - easy to read, easy to understand
- Meaningful names:
  - shorter names are generally better than longer ones, so long as they are clear
  - a long descriptive name is better than a short enigmatic name
  - a long descriptive name is better than a long descriptive comment
  - the length of a name should correspond to the size of its scope
- Functions:
  - functions should be small
  - functions should do one thing
  - functions should not have more than 3 parameters
  - command-query separation

# Summary

---

## ■ Comments:

- proper use of comments is to compensate for our *failure to express ourselves in code*
- don't write misleading, redundant and noise comments
- comments should be informative, should clarify the code, should express design intent
- document the public API, but don't comment the internal implementation

## ■ Exceptions:

- define exception classes in terms of a caller's needs
- don't return null
- don't pass null

## ■ Objects and data structures:

- procedural code:
  - hard to add new data structures because all the functions must change
- object-oriented code:
  - hard to add new functions because all the classes in the hierarchy must change