API in C#:
The Best Practices of Design and Implementation

Elias Fofanov

ENGINEER SPOCK

http://engineerspock.com

Intro

- **API** – Application Programming Interface
- It's important to understand how to build convenient, robust and easily maintainable APIs
- By the end of this course, you'll get a deep understanding of what is a good API
- The course will be extended

Course Outline

1. API Characteristics, API Development Principles
2. Naming Rules, Conventions
3. Common problems of design and implementation of APIs: classes vs structures, abstract classes vs interfaces, creational patterns vs constructors, how to implement dispose pattern
4. Common implementation smells: poor naming, long methods, output parameters
5. Common Architectural Design Smells: Primitive Obsession, Hidden Dependencies, Violation of Law of Demeter and other
6. How to deal with errors
7. How to deal with Nulls

Audience

1. All the C# developers from beginners to seniors
2. Relevant material for all kinds of C# developers
3. Topics are different in complexity (solid C# background is required in some lectures)

P.S.

You can learn modules and in some cases
even lectures inside modules in any order you like.

Introduction

Elias Fofanov

ENGINEER SPOCK

http://engineerspock.com

Course Outline

1. Naming Rules, Conventions
2. Common problems of design and implementation of APIs: classes vs structures, abstract classes vs interfaces, creational patterns vs constructors, how to implement dispose pattern
3. Common implementation smells: poor naming, long methods, output parameters
4. Common Architectural Design Smells: Primitive Obsession, Hidden Dependencies, Violation of Law of Demeter and other
5. How to deal with errors
6. How to deal with Nulls

Introduction Outline

1. API Characteristics
2. Public API vs Private API
3. API Development Principles

API Intro

- API (Application Programming Interface) – set of functionality
- The perfect API is an oxymoron

Types of APIs:

- Private (“zoo”)
- Public (“wilderness”)

API Characteristics

- Simplicity
- Expressiveness and Compromises
- Extensibility
- Consistency

Simplicity

- **Rule of Thumb:** “You can always add, but never remove.”
- **Compromise between power and simplicity:**
  When power of an API grows, its simplicity degrades.
- The only way to understand whether an API is simple or not is to estimate the time spent on understanding it by its users.

Expressiveness and Compromises

- Resources which can be allocated on API development are always limited
- API it is almost impossible to create universal APIs
- API developers have to implement first things first
- The only way to understand whether an API is simple or not is to estimate the time spent on understanding it by its users

Extensibility

- Reflects the capabilities to increase the power of an API without big rewritings
- You should be able to add new functionality and preserve the backward compatibility
- Open-Closed Principle (OCP) (mainly applicable in “zoo” APIs
- In public APIs we should at first preserve the backwards compatibility
  (if any doubts regarding a new API member - don’t introduce it)

Consistency

API has to be logical and consistent:
design decisions – strongly opinionated!

Example of poor consistency in the PHP String library in PHP:

- str_repeat
- strcmp
- str_cplit
- strlen
- str_word_count
- strrev

Public API vs Private API

- The cost of bad decisions in public API may be extremely high
- Private APIs should be developed bearing in mind all API characteristics
- Zookeepers must strive to become rangers

API Development Principles

- APIs should be **as simple as possible**, but no simpler.
- A good API should allow to **do a lot without learning a lot.**
- APIs should be based on **use cases**.
  It means two things:
  - imagine that you're a client of that API
  - sketch API as soon as possible

API Development Principles

- Provide **a low barrier** for using an API.

  In practice it means that you always:

  - Should provide the simplest constructors with default values of other required parameters
  - Should throw exceptions with messages which explain what to do to fix the problem
  - Shouldn’t require from clients to explicitly create more than one type for accomplishing main use cases
  - Shouldn’t require from clients to perform a wide initialization of an object

API Development Principles

- Build **self-explanatory APIs**
- Provide a **decent documentation**

Conclusion

Main characteristics:

- Simplicity
- Expressiveness and Compromises
- Extensibility
- Consistency

There’s a difference between public and private APIs.

Six development principles:

- API should be as simple as possible
- API should allow to do a lot without learning a lot
- API should be based on use cases
- Provide a low barrier for using an API
- Build self-explanatory APIs
- Provide a decent documentation

Names

Elias Fofanov

ENGINEER SPOCK

http://engineerspock.com

Intro

- **Correct naming** is extremely important from the perspective of **readability** and as a consequence from the perspective of **maintenance**
- **Naming rules** were different in the past
- Modern powerful PCs and managed languages with metadata make us able to develop very **powerful IDEs** which endow us by great power

Outline

1. General Principles of Naming
2. Naming Conventions in the .NET framework

Intention-Revealing Names

Give names which reflect the intention of a member.

What does “z” name mean? → “zipCode” is much better

What does “c” name mean? → “customer” is much better

int age; //is it a good name? → “int ageInYears” or “int ageInDays” are much better.

Disinformative Names

This code is “translated” from Perl to C#:

```
int length(string str);
int length(Array array);
```

A function should do what is expected from its name. Different meanings should not be expressed by the same or similar name.

Use Easily Readable Names

HorizontalAlignment is better than **AlignmentHorizontal**

CanScrollVertically is better than **ScrollableY**

counter is better than **theCounter**

Bad names: “gwvwe”, “gwrsp”.
Give pronounceable names.

“kolichestvo” is bad, “count” is good.
Use English for names.

No Encodings

Forget about the **Hungarian notation**.

Bad names: “i_age”, “i_count”.

I-prefix for interfaces is an exception from the rule: **IEnumerable**.

No Jokes

Code is not a place for jokes!

“Kill” is better than “Whack”. **No cute names.**

Use Programming Terms

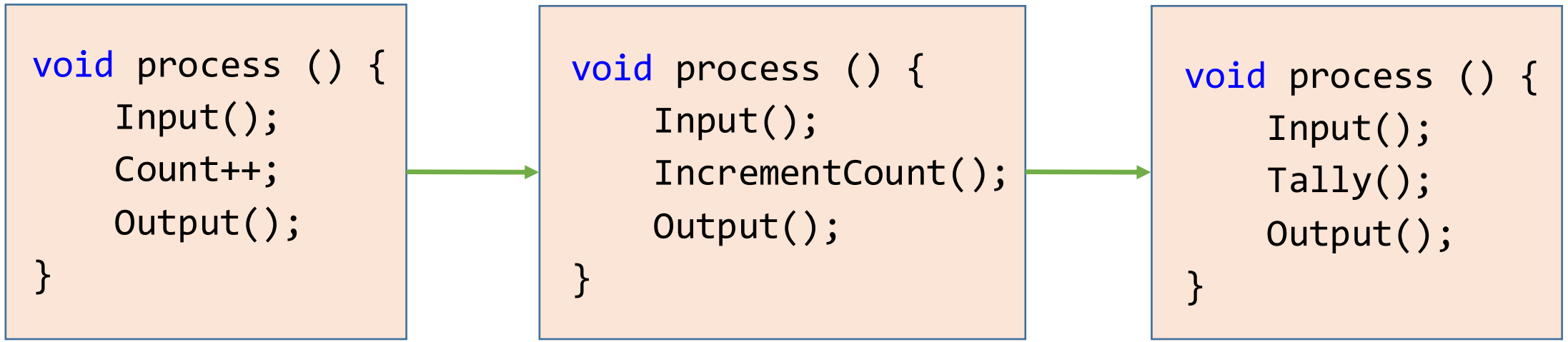
CustomerBuilder is a good name,
since a reader knows that it is implemented via the “Builder” pattern.

CustomerFactory implies that it is implemented via the “Factory” pattern.

Use Names from the Problem Domain

If people who work in the problem domain say “**Shift**” instead of “**Session**” what implies a working period, then you should name a class “**Shift**”, not “**Session**”, even if you like the latter.

Use Symmetry



Use Symmetry

- begin/end
- first/last
- locked/unlocked
- min/max
- next/previous
- old/new
- opened/closed
- visible/invisible
- source/target
- source/destination
- up/down

Scope and Name's Length

Rule: "The wider the scope the lengthier the name. The narrower the scope, the shorter the name."

```
public decimal GetTotalSalary(List<Employer> employers){
        decimal total;
        foreach(var e in employers){ // "e" is OK
                total += e.Salary;
    }
    return total;
}
```

Scope and Name's Length

Rule: "The wider the scope the lengthier the name. The narrower the scope, the shorter the name."

```
Document d;
…
public void FormReport(){
      Report report = CreateReport(d); //what is d?
       …
   }
}
```

Names of Classes

Rule: “A name of a class should be a noun.”

Examples: Customer, Painter, DateTimeParser.

Avoid: Manager, Data, Info, CustomerData.

Names of Functions

Rule: “A name of a function should be a verb.”

Examples: GetCustomers, GenerateDump,
bool CanRemoveStudent.

Boolean parameters: isCorrect, isBusy, CanExtract.

Well-Written Prose

Rule: “Code should be readable as a well-written prose.”

```
if(list.Contains(item))
if(expression.Matches(text)
```

vs

```
if(list.IsContained(item)
if(expression.Match(text)
```

Conventions

There are two ways of naming any API members in .NET:
PascalCasing and **camelCasing**.
Camel casing is used for naming private and protected fields and for parameters.
The other members are named with pascal casing.

Conventions Example

```
public class Customer {
    private int totalOrders;
    public event EventHandler StatusChanged;

    public int TotalOrders {
        get { return totalOrders; }
        set { totalOrders = value; }
    }

    public void TransferPayment(Payment payment) {

    }
}
```

Acronyms in .NET

Acronyms which consist of two letters and are not a part of a camel cased parameter have to be upper cased:

```
using System.IO;
public void StartIO(Stream ioStream){}
```

Acronyms which are lengthier than two letters and are not a part of a camel cased parameter have to be upper cased only at the first letter:

```
using System.Xml;
public void ProcessHtmlTag(string htmlTag){}
```

Compound Words

Pascal	Camel	Wrong
BitFlag	bitFlag	Bitflag or bitflag
Email	email	EMail
Id	id	ID
Ok	ok	OK
Pi	pi	PI
Metadata	metadata	MetaData

Shorted Names

Window is better than **Win**, **Extension** is better than **Ex**.

General Names

When a parameter does not bear any semantic payload you should name it as “**value**” or “**item**”.

```
void Write(double value);
void Write(float value);
void Write(short value);

void Add(T item);
```

Naming New Versions of API

StringParser → **StringParser2**

Prefer **numeric suffixes** for new API versions.

Naming Inheritors

Add a parent's class name as a suffix to the inheritors name.

```
class FileStream : Stream { … }

class CustomUserException : Exception { … }

class ClickedEventArgs : EventArgs { … }
```

Interfaces “I” prefix

Interfaces names should begin with “I” prefix.

Default interface implementation name should just remove the “I” prefix.

```
class Customer : ICustomer { … }
```

Naming Generic Types

The most common name for generic types is “T”.

```
Add(T item), List<T>.
```

Valuable suffix:

```
Func<TIn, TOut>
```

Naming Enumerations

Try to use singular names for enumerations.

```
enum Device {
    CardDispenser,
    BillAccepter
}
```

Use plural names for bit enumerations.

```
[Flags]
enum KeyboardKeys {
    Alt,
    Space
}
```

Naming Events

Use verbs for naming events.

```
event EventHandler StatusChanged;
event EventHandler ErrorOccured;

event EventHandler Processing;
event EventHandler Processed;

event EventHandler Closing;
event EventHandler Closed;

Action<Status> StatusChanged;
event EventHandler<Status> StatusChanged;
```

Naming Constants

Constants should have only the first letter uppercased:

```
public static int Age;
public const int Max = 100;
```

Don’t make constants uppercased:

```
public static int AGE;
public const int MAX = 100;
```

Conclusion

- Use intention-revealing names
- Don’t use Disinformative or controversial names
- Use easily readable and pronounceable names
- Use English as a coding language
- Don’t rely on encodings such as Hungarian notation
- Don’t be funny in your code
- Use well-known programming terms such as pattern names
- Use domain names
- Use symmetry in naming like Add and Remove, Push and Pop
- The wider the scope of a variable the longer its name can be
- Nouns for classes, verbs for functions
- Remember: code should be readable as a well-written prose

Designing and Implementing Types and their Members

Elias Fofanov

ENGINEER SPOCK

http://engineerspock.com

Outline

- How to choose between class and structure
- What is the difference between abstract classes and interfaces and when to choose one or another
- Peculiarities of implementing abstract classes
- How to choose between method and property
- Peculiarities of implementing constructors
- How to choose between constructor and a creational pattern
- How to implement a tester-doer pattern and what are pros and cons
- How to choose between exposing conversion methods and casting operators
- Peculiarities of implementing parameters
- How to implement the IDisposable pattern. It's not so simple as you may think. I encourage everyone to learn the corresponding lecture.

Class VS Struct

Question:

“What is the difference between class and structure?”.

Answer:

Structures implement the semantic of copying by value, whereas classes implement the semantic of copying by reference.

Abstract class VS Interface

Question:

“What is the difference between abstract class and interface?”.

Answer:

They are different in two aspects: mechanical and semantical.
Let’s take a closer look)))

Mechanical Difference

```
abstract class AlgorithmBase {
    public int Do() {
        int a1 = Operation1();
        int a2 = Operation2();
        return a1 + a2;
    }
    public virtual int Operation1() {
        return 55 ^ 35;
    }
    public virtual int Operation2() {
        return 21 + 48;
    }
}
```

```
public interface Algorithm {
    int Do();
    int Operation1();
    int Operation2();
}
```

Semantical Difference

Statement:
“interfaces define contracts” or “interface is a contract”

Doubt: Really?

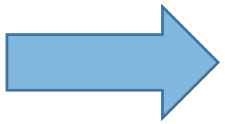
Rebuttal: Contracts posses semantic payload. Interfaces don’t bear any semantic payload. They expose shapes expressed in signatures.

Real Contract

```
abstract class CollectionContract<T> : IList<T> {
    public void Add(T item) {
        AddCore(item);
        count++;
    }
    public int Count {
        get { return count; }
    }
    protected abstract void AddCore(T item);
    private int count;
    ...
}
```

API Design Considerations

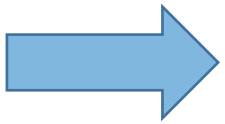
- An interface can’t be easily changed without breaking existing clients
- An interface is easily extendable by clients via extensions
- It’s possible to inherit multiple interfaces



Interface is more supple from the client’s perspective,
while it’s more rigid from the developer’s perspective.

API Design Considerations

- Abstract class supports reusability of logic
- Abstract class supports encapsulation
- Abstract class can be extended without breaking existing clients



Abstract class is more supple from the developer's perspective, while it's more rigid from the client's perspective.

Implementing Abstract Classes

Bad design:

```
public abstract class Device {
        public Device() {

        }
    }
```

Good design:

```
public abstract class Device {
        protected Device() {

        }
    }
```

Implementing Abstract Classes

Implement some minimum amount of logic in the abstract class and then implement at least one inheritor.

Property VS Method

Sometimes methods may require more than three parameters. In such a case you generally have three options:

1. All the parameters which can be set to a meaningful default state convert to properties.
2. Create a bunch of overloaded methods.
3. Create a method with a long list of optional parameters.

Property VS Method

- **Remember**: you should create properties as simple as you can.
- DateTime.Now is a bad property.
- DateTime.Now() is better.
- If every time the getter of a property returns different values, then this is a method, not a property.
- DateTime.Now().Year is a good property.
- Guid.NewGuid() is a good method.

Property VS Method

```
public string ImportantTxt {
    get { return TxtBlock.Text; }
    set {
        Application.Current.Dispatcher.BeginInvoke(() = > {
            TxtBlock.Text = value;
        });
    }
}
```

```
string txt = ExternalService.GetTxt();
ImportantTxt = txt;
if (ImportantTxt == "expectedValue") {
    //do something
}
```

Implementing Constructors

- Always look at constructors from the invariants point of view. Constructor MUST require all the necessary parameters without which your object is in an invalid state.
- No long operations
- Validate input, throw exceptions
- Always define default constructor (if it is appropriate to have one)
- No virtual calls from constructors
- Don’t throw exceptions from static constructors

Creational Pattern VS Constructor

Patterns: Factory, Builder, Singleton.

Factory Method and Abstract Factory are the most commonly used.

- Constructor is simpler than any creational pattern
- Factories are not discoverable by Intellisense!

When Factories Are Better

Apply a constraint on a client:
limit the number of possible instances by implementing, for example, the Singleton pattern.

When Factories Are Better

You need a fluent API of object creation:
the Builder pattern may be implemented.

```
Order order = Order
                .AddFreeShipping()
                .IncludeItem(10)
                .SetQuantity(2);
```

When Factories Are Better

When you need to return a base type from a creational procedure:

```
public class Type {
    /// <summary>
    /// returns PropertyInfo, ConstructorInfo, MethodInfo...
    /// </summary>
    MemberInfo[] GetMember(string name);
}

public static class Activator {
    public static object CreateInstance(Type type) { }
}
```

When Factories Are Better

When you need to expose more contextual information about the creation process:

```
public class Money {
    public static Money CreateFromCents(int cents);
    public static Money FromDollars(decimal dollars);
}
```

When Factories Are Better

When you need to expose conversion operations.

There are two common types of this rule:

Try-Parse pattern and direct conversion.

```
DateTime dt;
bool result = DateTime.TryParse("12/08/1988", out dt);
if (result) {

}
DateTime dt = DateTime.Parse("12/08/1988");
```

Tester-Doer Pattern

If you want to allow users of your API to avoid dealing with exceptions, then provide a tester property:

```
ICollection<Student> students = GetStudents();
if (!students.IsReadOnly)//tester
{
   students.Add(new Student("Joe")); //doer
}
```

Conversion VS Casting Operators

Misuse of casting operators:

```
class TimeSpan {
    public static implicit operator int(TimeSpan value) { }
    public static explicit operator TimeSpan(int value) { }
}
```

```
TimeSpan timeSpan  = 10;
int timeSpan = (int)(new TimeSpan());
```

Conversion VS Casting Operators

The right way:

```
class TimeSpan {
      public static TimeSpan FromSeconds(int seconds) { }
      public static int ToSeconds(TimeSpan timeSpan) { }
}
```

Conversion VS Casting Operators

In the case you really need to implement casting operators bear in mind that:

- Implicit conversion operator shouldn't be implemented
  if there is at least little chance of data or precision loss.
  There's no implicit conversion from int to double since double is wider than int.
- Implicit conversions should not generate exceptions.
  Such a behavior violates the least astonishing principle.
  In other words, users don't expect such a behavior from implicit conversions
- If there is a loss during the explicit conversion,
  throw the InvalidCastException.

Implementing Parameters

- Use the most specific type for parameters.
  For example, if a method receives a collection which can't be modified, use IEnumerable instead of IList
- Avoid out and ref parameters, especially in public APIs
  (out in TryParse-methods is the exception from the rule)
- Throw exceptions defined in the BCL when parameters validation fails
- Methods should not require more than three parameters

Implementing Parameters

Feels good?)))

```
Stream stream = File.Open("foo.txt", true, false);
```

Right way:

```
Stream stream = File.Open("foo.txt", FileMode.Append,
                                     FileAccess.Read);
```

Implementing Parameters

Function which takes either Boolean or Enum parameter most likely violates the SRP!

```
void SetSwitch(bool isOn);
```

Better:

```
void SetOn();
void SetOff();
```

Implementing Parameters

With bool:

```
void ChangeStudentStatus(bool enroll);
```

Better:

```
void Enroll();
void Expell();
```

Implementing Parameters

Passing Boolean parameters, always use Named Parameters feature:

```
void ChangeStudentStatus(bool enroll);
```

Call:

```
void ChangeStudentStatus(enroll:true);
```

Instead of:

```
void ChangeStudentStatus(true);
```

Implementing Dispose Pattern

Any meaningful program acquires system resources such as files, handles, sockets.

If you forget to remove them from memory, then you likely will get a so-called memory leak.

Implementing Dispose Pattern

System.Object exposes:

```
protected virtual void Finalize();
```

Objects which override the Finalize method are called finalizable.

Implementing Dispose Pattern

- The moment of finalization is undetermined
- When CLR finalize an object, it puts off the actual reclaiming of that object's memory. GC will reclaim the memory only in the next collection process.

Implementing Dispose Pattern

System.IDisposable exposes:

```
void Dispose();
```

Allows the manual (explicit) removal of the unmanaged resources.

Implementing Dispose Pattern

Type which implements **IDisposable** can be wrapped in the **using** statement:

```
using (SqlConnection conn = new SqlConnection()) {}
```

Will be compiled as:

```
SqlConnection conn = new SqlConnection());
try {
} finally {
    conn.Dispose();
}
```

Implementing Dispose Pattern

Do we need to implement IDisposable and override the Finalize method at the same time each time we deal with acquired resources?

Implementing Dispose Pattern

In 99% of cases you don’t need to implement the finalization method.

Just implement the IDisposable and you’ll be fine.

Implementing Dispose Pattern

- If the disposable class is not sealed then you have to define the Dispose method which takes the Boolean flag as virtual, since there can be inheritors. If the disposable class is sealed, then you have to make that method private.
- The disposable object can also keep track of its state in a field like bool isDisposed.
- If an application domain gets unloaded, then there is no guarantee that finalizers will be invoked.
  If you want to finalize an object even in such circumstance like unloading of the application domain, then you need to inherit your finalizable object from the CriticalFInalizerObject class.

Implementing Dispose Pattern

The simple version of IDisposable implementation where you don't implement a finalizing method.

```
class OnlyManagedResources : IDisposable {
  public void Dispose() {
    DisposeManagedResources();
  }
  protected virtual void DisposeManagedResources() {}
}
```

Summary

- Class VS Structure
- Implementing Abstract Classes
- Property VS Method
- Implementing Constructors
- Creational Patterns VS Constructors
- Tester-Doer
- Conversion VS Casting
- Implementing Parameters
- Implementing Dispose Pattern

Implementation Smells

Elias Fofanov

ENGINEER SPOCK

http://engineerspock.com

Outline

1. Poor Names
2. Naming Conventions
3. Incorrect Variable Declarations
4. Magic Numbers
5. Functions will too many arguments
6. Smell of Long Methods. “Extract Till You Drop” refactoring technique.
7. Poor conditional clauses.
8. The smell of out-parameters.
9. Comments.
10. Positive if-statements.

Mysterious Names

button1, **button2**, **button3**, **button4** – bad names for UI-controls
button1_click, **button2_click** - bad names for event handlers

GenerateReport for a button which triggers the generation of a report.

CloseWorkShift for a button which triggers the closing process of a work shift.

CloseWindow for a button which triggers the closing of a window.

Class1 is a bad name for a class.
Customer is much better.

Meaningless Names

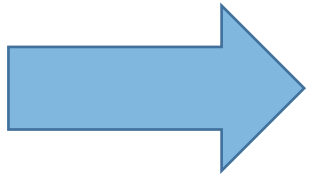
```
public void StartProcessing();
public void FillData();
```

Better:

```
public void ParseXmlFile();
public void FillStudentInfo();
```

Shortened Names

```
string iniMess;
string DocsV;
string AddrTo;
```



```
string InitializationMessage;
string DocumentGroup;
string Addressee;
```

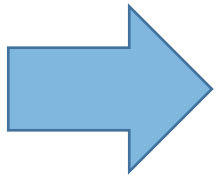
Notations

We don’t need any encodings like Hungarian notation, because of powerful IDEs!

```
bool GetResponse(DataSet ASet, out DataRow ARow);
```

Bad:

```
int iNameLength;
List<int> p_Collection;
```



Good:

```
int nameLength;
List<int> numbers;
```

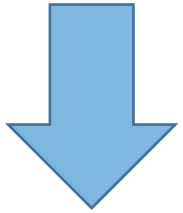
Ambiguous Names

```
string documentNameId;
int MoneyToString(string money);
```

Names should clearly reveal their only one well-expressed intent!

Noisy Names

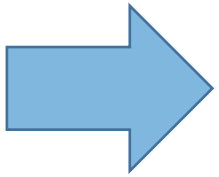
```
void EnrollTheStudent(Student student);
List<Student> studentsList;
```



```
void EnrollStudent(Student student);
List<Student> students;
```

Naming Predicates

```
if(isFound){}
```



```
if(found){}
```

Poor Naming Conventions

Consistency is very important!

- StyleCop – free tool that enforces the naming rules
- FxCop – free tool that enforces designing rules, integrated into Visual Studio since version 2012
- ReSharper – is a paid tool that has a great number of features including enforcing naming and designing rules and much more.

Variable Declaration On the Top

Bad code:

```
string emailAddress = string.Empty;
string managerEmailAddress = string.Empty;
if (//condition) {
  emailAddress = Foo();
  EmailMessage.To = emailAddress;
}
else {
  emailAddress = Bar();
  // Do something different with emailAddress;
}
// No more usages of emailAddress
```

Variable Declaration On the Top

Good code:

```
if (//condition) {
  var emailAddress = Foo();
  EmailMessage.To = emailAddress;
}
else {
  var emailAddress = Bar();
  // Do something different with emailAddress;
}
```

Magic Numbers

```
int responseCode = GetDeviceResponse();
if (responseCode == 188) {
}
```

```
const int NoConnection = 188;
int responseCode = GetDeviceResponse();
if (responseCode == NoConnection) {
}
```

Too Long Method

Make your functions no longer than 10 lines of code.
Roughly.

Too Long Method

“Extract Till You Drop” - extract chunks of code from a method until there is nothing to extract. Introduced by Uncle Bob.

Christin Gorman argued that this technique sounds like a no-brainer.

For addition, there was a blog post by John Sonmez about refactoring of a .NET BCL-method using this technique.

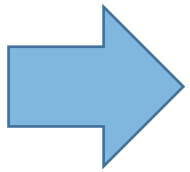
Poor Conditional Clauses

```
int discount = 0;
if (customer.Status == Status.Platinum) {
    discount = 75;
}
else {
    discount = 25;
}
```

```
int discount = customer.Status == Status.Platinum ? 75 : 25;
```

Poor Conditional Clauses

```
int discount =
    customer.Status == Status.Platinum
      ? 75
          : customer.Status == Status.Silver
          ? customer.TotalOrders > 100
                    ? 50 : 35
          : 25;
```



```
switch (customer.Status) {
    case Status.Platinum:
        discount = 75;
        break;
    case Status.Silver:
        discount = customer.TotalOrders > 100
                                        ? 50 : 35;
        break;
    default:
        discount = 25;
        break;
}
```

Poor Conditional Clauses

```
if (isAdjecentOnTop && isAdjecentOnBottom &&
    isAdjecentOnLeft && isAdjecentOnRight) { }
```

```
bool pieceCanBeCaptured = isAdjecentOnTop && isAdjecentOnBottom &&
                          isAdjecentOnLeft && isAdjecentOnRight;
if (pieceCanBeCaptured) { }
```

Output Parameters

```
int GetDictionaries(out ITariff tariffs, out IStation[] stations,
                    out ITransporter[] transporters,
                    out ITrainCategory[] categories,
                    out ITariffPlan[] tariffPlans);
```

```
int GetDictionaries(out RailwayDictionariesContainer container);
```

```
Result<RailwayDictionariesContainer> GetDictionaries();
```

Output Parameters

```
public (int sum, int count) Tally(IEnumerable<int> values);

var t = Tally(myValues);
Console.WriteLine($"Sum: {t.sum}, count: {t.count}");
```

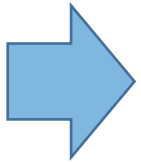
Comments

In the early days of programming, languages and existing tools didn’t provide enough features for making the code base expressive and self-explanatory.

C#, Java and their environment is the different story. Nowadays, the role of comments is significantly diminished.

Comments

```
int age = 45;
// If age is greater than 18 a person can vote
if (age >= 18)
     Console.WriteLine("Vote");
else
     Console.WriteLine("Can't Vote");
```



```
int age = 45;
if (CanVote(age))
     Console.WriteLine("Vote");
else
     Console.WriteLine("Can't Vote");
private bool CanVote(int age) {
      return age >= 18;
}
```

Comments

```
public class PrimeGenerator {
    /// <summary>
    /// Uses Eratosthenes sieve algorithm for generating prime numbers.
    /// </summary>
    /// <param name="count">Determines how many prime numbers to generate.</param>
    public static int[] GetPrimes(int count) {}
}
```

Comments which describe internal details which cannot be expressed by names are helpful.

Remember: comments should add value to the understanding of code.

Prefer Positive if-statements

```
Student student = FindStudentById(10);
bool notFound = student == null;
if (notFound) {
}
else {
}
```

```
if (!notFound) {
    //if found
}
```

```
Student student = FindStudentById(10);
bool found = student != null;
if (found) {
}
else {
}
```

Conclusion

1. Poor Names, impose Naming Conventions by special Tools
2. Incorrect Variable Declarations – Don’t declare variables on the top.
3. Magic Numbers. Refactor them into well-named constants.
4. Functions will too many arguments. Refactor them out into separate classes.
5. Smell of Long Methods. Make methods 10 lines long, roughly.
6. Use ternary operators, but don’t abuse them.
7. The smell of out-parameters. Refactor classes out, rely on the Result monad.
8. Comments. They should add valuable information about implementation details.
9. Positive if-statements. Don’t write double negative predicate expressions.

Architectural Design Smells

Elias Fofanov

ENGINEER SPOCK

http://engineerspock.com

Outline

Currently, we will look at the following topics in this module:

1. Primitive Obsession. This smell is about misusing primitive types for representing high level concepts.
2. Hidden Dependencies. This topic requires from you the acquaintance with IoC-containers.
3. Violation of Law of Demeter. This law is about coupling between objects.
4. Temporal coupling. Here we will address the problem of coupling between API members.
5. Switch Statements. Switch-statement is a procedural structure. We will look at how to refactor switch-statements out.

Primitives Obsession

Don’t use primitives for representation of inappropriate concepts.

```
public class Address {
    public string ZipCode { get; set; }

    public void ValidateZipCode(string zipCode) {}
}
```

```
Address address = new Address();
// constructor
address.ZipCode = new ZipCode("12345");

// explicit casting operator
address.ZipCode = (ZipCode) "12345";

// implicit operator
 string zip = address.ZipCode;
```

```
public class ZipCode {
    private readonly string _value;
    public ZipCode(string value) {
         // perform regex matching to
         // verify XXXXX or XXXXX-XXXX
         // format
        _value = value;
    }
    public string Value {
        get { return _value; }
    }
}
```

Hidden Dependencies

```
public class OrderProcessor: IOrderProcessor {
    public void Process (Order order) {
        var validator = Locator.Resolve<IOrderValidator>();
        if (validator.Validate(order)) {
            var shipper = Locator.Resolve<IOrderShipper>();
            shipper.Ship(order);
        }
    }
}
```

```
var op = new OrderProcessor();
op.Process(order); //throws exception
```

- An instance of the **Order** type is required
- An instance of the **IOrderValidator** has to be resolved by using a global static bus
- An instance of the **IOrderShipper** has to be resolved by using a global static bus

Hidden Dependencies

Service Locator becomes an anti-pattern when it is used in the business logic here and there in the code base.

It is ok to use the Service Locator in the Infrastructural code.

Hidden Dependencies

```
public class OrderProcessor: IOrderProcessor {
    public void Process(Order order, IOrderValidator validator, IOrderShipper shipper);
        if (validator.Validate(order)) {
            shipper.Ship(order);
        }
    }
}
```

But what if the **IOrderProcessor** defined like this?

```
public interface IOrderProcessor {
    void Process(Order order);
}
```

Hidden Dependencies

```
public class OrderProcessor : IOrderProcessor {
    private readonly IOrderValidator validator;
    private readonly IOrderShipper shipper;

    public OrderProcessor(IOrderValidator validator, IOrderShipper shipper) {
        if (validator == null)
            throw new ArgumentNullException("validator");
        if (shipper == null)
            throw new ArgumentNullException("shipper");

        this.validator = validator;
        this.shipper = shipper;
    }

    public void Process(Order order) {
        if (this.validator.Validate(order))
            this.shipper.Ship(order);
    }
}
```

Hidden Dependencies

```
public class OrderProcessor : IOrderProcessor {
    public OrderProcessor(IOrderValidator validator, IOrderShipper shipper);
    public void Process(Order order);
}
```

Violation of Law of Demeter

Law of Demeter (LoD) or principle of least knowledge is a design guideline for developing software, particularly object-oriented programs.

Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
Or: Each unit should only talk to its friends; Don't talk to strangers.

Violation of Law of Demeter

A method of an object may only call methods of:

1. The object itself.
2. An argument of the method.
3. Any object created within the method.
4. Any direct properties/fields of the object.

Violation of Law of Demeter

Let's pretend that we should model the business relationships between a paperboy and a customer who wants to buy magazines.

A paperboy rings the doorbell, a customer opens it, a paperboy somehow has to be paid and then hand over a magazine to the customer.

Violation of Law of Demeter

The Law of Demeter is not about the number of dots.

This law is about **reducing the coupling and improving the encapsulation.**

It’s OK to use many dots when digging the data structure like:

ExcelDocument.Sheet.Cell

Temporal Coupling

Temporal Coupling - hidden interconnection between two members of an API, which requires the right sequence of calls made by a caller.

Temporal Coupling

EndpointAddressBuilder – factory for producing new endpoint addresses.

```
var builder  = new EndpointAddressBuilder();
EndpointAddress endpointAddress = builder.ToEndpointAddress();
```

```
var builder  = new EndpointAddressBuilder();
builder.Uri = new UriBuilder().Uri;
EndpointAddress endpointAddress = builder.ToEndpointAddress();
```

Temporal Coupling

```
class EndpointAddressBuilder {
      private Uri uri;
      public EndpointAddressBuilder(Uri uri) {
          this.uri = uri;
      }
}
```

Remember: Always try to make invalid states unrepresentable!

Switch Statements

- Switch-statement is a procedural construct, not object-oriented
- Switch-statements are not bad.
  Object-oriented programs mix procedural and object-oriented approaches.

Prefer Positive if-statements

```
Student student = FindStudentById(10);
bool notFound = student == null;
if (notFound) {
}
else {
}
```

```
if (!notFound) {
    //if found
}
```

```
Student student = FindStudentById(10);
bool found = student != null;
if (found) {
}
else {
}
```

Comments

```
public class PrimeGenerator {
    /// <summary>
    /// Uses Eratosthenes sieve algorithm for generating prime numbers.
    /// </summary>
    /// <param name="count">Determines how many prime numbers to generate.</param>
    public static int[] GetPrimes(int count) {}
}
```

Comments which describe internal details which cannot be expressed by names are helpful.

Remember: comments should add value to the understanding of code.

Conclusion

1. Primitives Obsession. We found out that representing the concept of a zip code by the string type is a bad idea. Primitives obsession leads to a procedural cluttered code over time and increases the maintenance cost.
2. Hidden dependencies. Make all the dependencies visible for a caller, because dependencies are a part of the API's contract.
3. The law of Demeter is not about number of dots. It's about coupling between objects and improving encapsulation. In short, the object shouldn't talk to strangers.
4. Temporal coupling - interconnection between two members of an API, which requires the right sequence of calls made by a caller.
5. Switch-statements. Switch statements can be refactored to the OO-design with a class for each case-statement. Another option is to rely on a dictionary instead of a switch-statement.

Dealing with Errors

Elias Fofanov

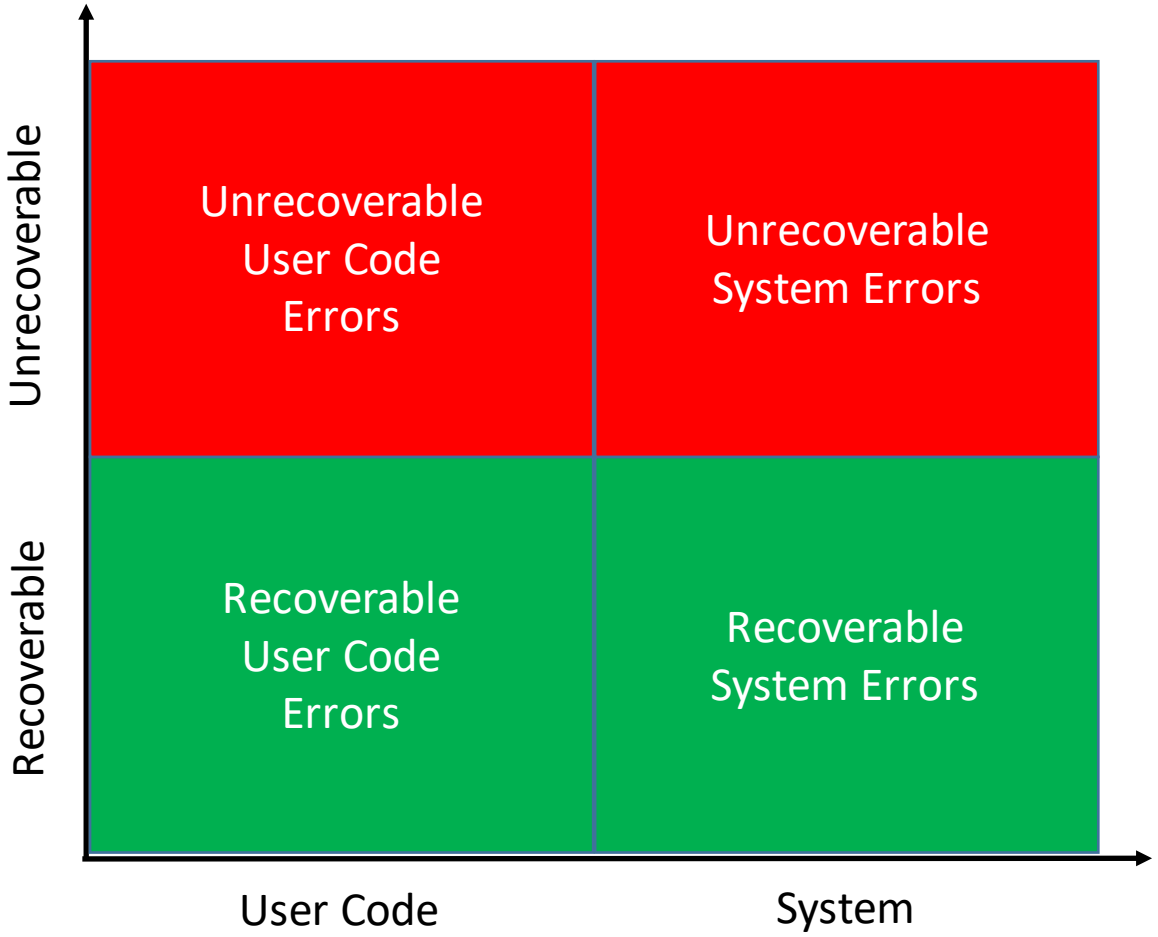
ENGINEER SPOCK

http://engineerspock.com

Outline

1. Is it a good idea to use exceptions for dealing with errors everywhere in every case?
2. Understand the gist of errors, their nature.
3. Exceptions mechanism is not the answer for all problems.
4. What tools do you have to tackle the complexity of handling errors in your programs. Temporal coupling.
5. What types of errors do we have to deal with and why they are different.

What does an “Error” mean?



Recoverable System Errors

Examples: FileNotFoundException, SocketException, and many others.

Should be handled!

Unrecoverable System Errors

Examples: StackOverflowException , OutOfMemoryException .

Can’t be handled meaningfully!

Unrecoverable User Code Errors

This type of errors represent **bugs in your code.**

Example: NullReferenceException.

Can’t be handled meaningfully!

Recoverable User Code Errors

This type or errors represent those errors which we expect to happen.

Avoid using Exceptions for representing and handling recoverable user code errors.

Exceptions = Errors?

An exception is a form of an error representation.

The original meaning of throwing exceptions is to fail fast, **unwinding the stack, until the appropriate catch block will handle the exception.**

How to handle Exceptions?

```
try {
 // do something
}
catch(Exception ex){
}
```

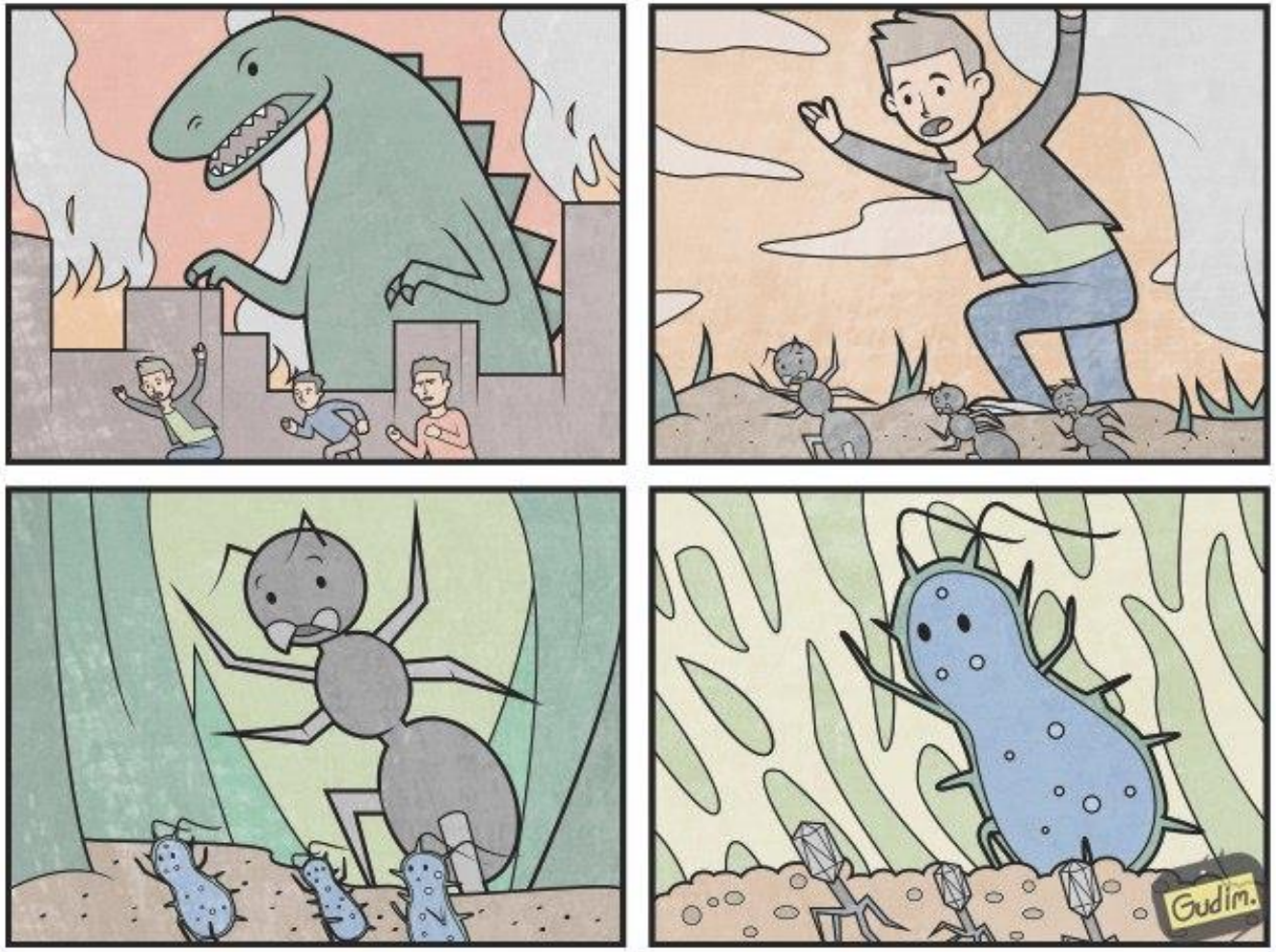
Example with Checked Exception in Java

```
public static void readFile(string filePath) throws IOException {
    FileReader file = new FileReader(filePath);
    BufferedReader fileInput = new BufferedReader(file);

    // Print first 3 lines of the file
    for (int counter = 0; counter < 3; counter++)
        System.out.println(fileInput.readLine());

    fileInput.close();
}
```

Scalability Problem



Versioning Problem

You can’t just simply go

and add a new type of exception

“

Their power comes at an extremely high cost; it becomes impossible to understand the flow of the program using only local analysis; the whole program must be understood. This is especially true when you mix exceptions with more exotic control flows like event-driven or asynchronous programming. Avoid, avoid, avoid; use exceptions only in the most exceptional circumstances, where the benefits outweigh the costs.

Eric Lippert

“

“…people don't care. They're not going to handle any of these exceptions. **There's a bottom level exception handler around their message loop.** That handler is just going to bring up a dialog that says what went wrong and continue.”

“It is funny how people think that the important thing about exceptions is handling them. That is not the important thing about exceptions. In a well-written application **there's a ratio of ten to one, in my opinion, of try finally to try catch**.”

Anders Hejlsberg

Which exceptions do you need to catch?

```
try
{
    File.ReadAllLines();
}
catch (?)
{

}
```

Exceptions from File.ReadAllLines()

- ArgumentException
- ArgumentNullException
- PathTooLongException
- NotSupportedException

- UnathorizedAccessException
- SecurityException

- IOException
- DirectoryNotFoundException
- FileNotFoundException

Do we know which exceptions to catch?

SEHException

If CLR doesn’t know how to map the unmanaged exception, it wraps it by the SEHException type.

Misconception 1

"…Exception is a software layer just like Entity, Data access, UI, logging .. etc."

Wrong.
Exceptions handling problem is an aspect oriented problem.

Misconception 2

“Calling to File.Exists() before trying to open a file
frees you from catching exceptions. ”

Wrong.
True returned from that call doesn’t guarantee that File.Open() will not fail.

Misconception 3

“Exceptions are very slow and degrade performance.”

Wrong.
Jon Skeet’s laptop in 2006 was able to throw more than 100 exceptions per millisecond.

Misconception 4

“What a method can throw is pretty well understood.”

Wrong.
You can’t list the potential exceptions from literally any call to a method from the BCL.

Misconception 5

“Don’t catch stuff inside your method, let it cascade up.”

Wrong.
You should handle exceptions as soon as possible, as close to the point where they arise as you can.

Misconception 6

“ONLY catch exceptions if you can do something meaningful.”

Right and Wrong :)
Unfortunately, sometimes we have to suppress noise exceptions.

Conclusion

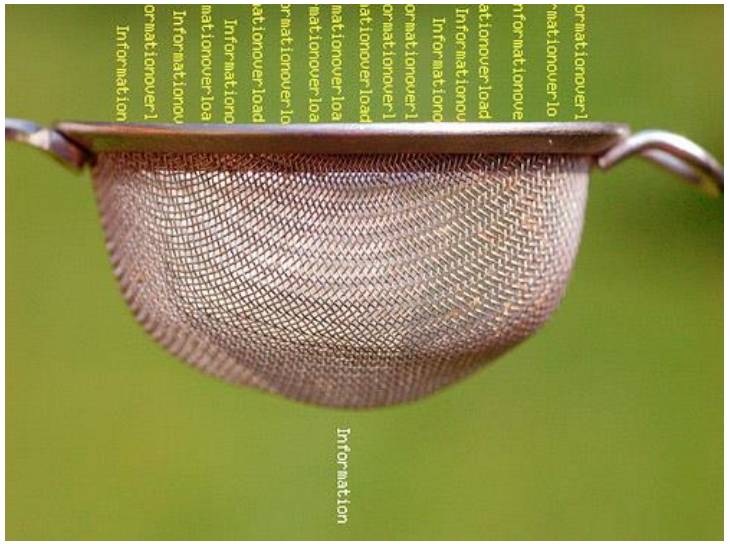
Exceptions should signalize a bug:

- Preconditions are violated
- The application is in the corrupted state and should be closed immediately

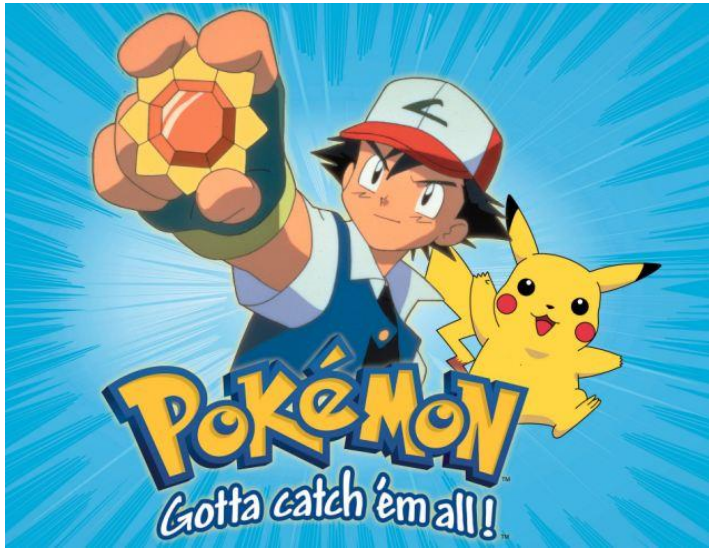
Catch exceptions as close to the source of that exception as you can.

Exception Handling Strategies

Catching only expected exceptions



Filtering Strategy



Catch Them All!

Catching Expected Exceptions

```
try {
    File.ReadAllLines();
}
catch (IOException ex) {
}
catch (SecurityException ex) {
}
catch (UnathorizedAccessException ex) {
}
```

Filtering Strategy

```
var policy = new ExceptionHandlingPolicy();
try {
    File.ReadAllLines();
}
catch (Exception ex) {
    policy.Handle(ex);
}
```

Logic resides in the ExceptionHandlingPolicy.

Catch Them All

```
try {
    File.ReadAllLines();
}
catch (Exception ex) {
}
```

Catch Them All

Choosing the exceptions handling strategy consider:

- What type of application are going to develop?
- What is the proficiency level of your team? “Correct” errors handling is very expensive.
- The size of an application is also a reason you should consider.

“

… most people don’t write robust error handling code in non-systems programs, throwing an exception *usually* gets you out of a pickle fast. Catching and then proceeding often works too. No harm, no foul. Statistically speaking, programs “work.”

Joe Duffy

Command Query Separation

Types of Functions:

- Functions which perform **commands**
- Functions which perform **queries** and return a result

CQS

Mix of a command and query:

```
public bool LogOn(string username, string password){}
if(LogOn (“spock”, “qwerty”)){
}
```

CQS

Separated command and query:

```
public void LogOn(string username, string password) {}
public bool IsLogedOn(string username, string password) {}
```

Errors and Functional Programming

Function should produce the same result for the same input and they should not produce any side effects (from the functional programming perspective).

Exceptions are side effects by their nature.

Recoverable User Code Errors

```
public void TransferMoney(Payment p)
{
    if (p == null)
        throw new ArgumentNullException("p");
    ValidatePayment(p);
}
```

Pipelining

Pipelining is a technique which allows an output of one function to be passed to the next one. This allows organizing the natural flow of data.

Pipelining

Reverted reading order:

```
File.WriteAllText(@"C:\tmp\Notes.txt",
                  Encoding.ASCII.GetString(
                      new byte[] {83, 80, 79, 67, 75}
                      )
                  );
```

```
var textInBytes = new byte[] { 83, 80, 79, 67, 75 };
var contents = Encoding.ASCII.GetString(textInBytes);
Console.WriteLine(contents);
```

Pipelining

Pipelining in F#

```
[| 83uy; 80uy; 79uy; 67uy; 75uy |]
|> Encoding.ASCII.GetString(textInBytes)
|> Console.WriteLine(contents)
```

Method chaining in C#

```
var sb = new StringBuilder();
sb.Append("Hello")
  .Append(",")
  .Append("World")
  .AppendLine("!");
```

«Try-Catch» Hell

```
public HttpResponseMessage CreateCustomer(string name,
                                          string billingInfo) {
    Result<BillingInfo> billingInfoResult = null;
    try {
        billingInfoResult = BillingInfo.Create(billingInfo);
    }
    catch (Exception ex) {
        Log(ex);
        return CreateResponseMessage(ex);
    }

    Result<CustomerName> customerNameResult = null;
    try {
        customerNameResult = CustomerName.Create(name);
    }
    catch (Exception ex) {
        Log(ex);
        return CreateResponseMessage(ex);
    }

    try {
        _paymentGateway.ChargeCommission(billingInfoResult.Value);
    }
    catch (Exception ex) {
        Log(ex);
        return CreateResponseMessage(ex);
    }

    var customer = new Customer(customerNameResult.Value);
    try {
        _repository.Save(customer);
    }
    catch (Exception ex) {
        Log(ex);
        _paymentGateway.RollbackLastTransaction();
    }
    try {
        _emailSender.SendGreetings(customerNameResult.Value);
    }
    catch (Exception ex) {
        Log(ex);
        return CreateResponseMessage(ex);
    }
    return CreateResponseMessage(true);
}
```

«No Try-Catch» Paradise

```
public HttpResponseMessage CreateCustomer(string name, string billingInfo)
{
    Result<BillingInfo> billingInfoResult = BillingInfo.Create(billingInfo);
    Result<CustomerName> customerNameResult = CustomerName.Create(name);

    return Result.Combine(billingInfoResult, customerNameResult)
        .OnSuccess(() => _paymentGateway.ChargeCommission(billingInfoResult.Value))
        .OnSuccess(() => new Customer(customerNameResult.Value))
        .OnSuccess(customer => _repository.Save(customer)
                                          .OnFailure(() => _paymentGateway.RollbackLastTransaction()))
        .OnSuccess(() => _emailSender.SendGreetings(customerNameResult.Value))
        .OnBoth(Log)
        .OnBoth(CreateResponseMessage);
}
```

4 Types of Methods

Commands:

```
  void EnrolStudent(Student student); //not expected to fail
Result EnrolStudent(Student student); //expected to fail
```

Queries:

```
       Student GetStudent(string name); //not expected to fail
Result<Student> GetStudent(string name);//expected to fail
```

BCL Exception Types

- **System.Exception** is the base type of all exceptions
- Always throw the most concrete type
- Don’t ever use **ApplicationException**
- Throw **InvalidOperationException** in the case an object gets to an **inconsistent state**
- Throw **ArgumentException** in the case of invalid arguments
- If an argument violates allowed boundaries, throw ArgumentOutOfRangeException.
- Don’t throw NullReferenceException, IndexOutOfRangeException , AccessViolationException, ComException, ExecutionEngineException, SEHException. These are system exceptions.
- Don’t try to catch StackOverflowException and OutOfMemoryException.

Custom Exception Types

- Always inherit your exceptions from the System.Exception or other BCL exception types.
- Always add to the exception type name the “Exception” suffix: InvalidUserException etc.
- Never throw more than 100 exceptions per second. Exceptions can incur performance degradation in such circumstances.
- Always provide the following constructors in your custom exceptions types:

```
public class InvalidUserException : Exception, ISerializable {
    public InvalidUserException() {}
    public InvalidUserException(string message) : base(message) {}
    public InvalidUserException(string message, Exception inner) : base(message, inner) {}
    //for serialization purposes.
    protected InvalidUserException(SerializationInfo info, StreamingContext context) {

    }
}
```

Conclusion

- Problems with Exceptions
- Exceptions should reflect the unrecoverable errors
- Use Result monad for dealing with recoverable user code errors
- Pipelining and methods chaining
- CQS

Dealing with Nulls

Elias Fofanov

ENGINEER SPOCK

http://engineerspock.com

Outline

1. The problems of using null values
2. Null Object Pattern
3. Maybe monad
4. Tools
5. Static Analysis
6. Special case of returning the IEnumerable

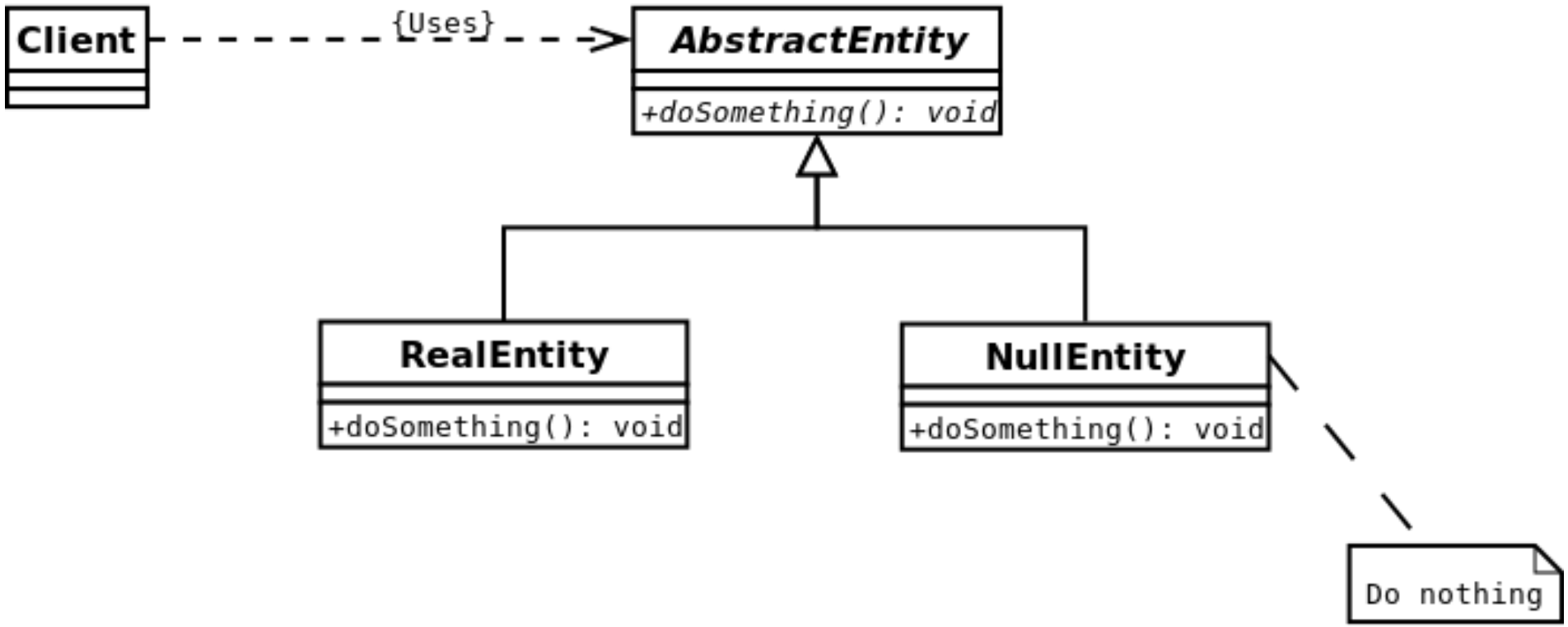
Problems with Nulls

- Nulls make code harder to read
- Imposing of Defensive Programming
- NullReferenceException

Semantical Problem

```
bool CheckCard(CardDescription description, out CardReadingResult readingResult) {
    readingResult = null;
    if (description != null) {
        CardReadingResult r = device.SearchForCard(description.CardType);
        if (r == null) {
            Notify("Card was not inserted.");
        }
        else {
            var info = CardInfoParser.GetInfo(r.CardInfo);
            if (info.Number == description.Number) {
                readingResult = r;
            }
        }
    }
    return readingResult != null;
}
```

Null Object Pattern



Maybe Monad

- Impossible to use non-nullable reference types in C#
- Maybe explicitly states that a method accepts or returns a nullable reference type

Maybe Monad

```
public void Student GetStudent();
```

```
public void Maybe<Student> GetStudent();
```

Automating Null Checks

- We can incidentally return a null from a method
- We're cursed to write null checks
- Fody.NullGuard is a special tool which weaves assemblies in compile time

Static Analysis

- Any analysis without actually running the code can be considered static
- C#-code does not provide enough information about possible nulls out of the box for static analysis tools such as ReSharper, SonarLint, PVS-Studio
- To empower static analysis tools we need to explicitly mark inputs and outputs by special attributes

Returning IEnumerable

```
public IEnumerable<Customer> GetCustomers(Func<Customer, bool> predicate)
```

What to return in the case the function didn't find any customers?

Returning IEnumerable

"Null" means missing information, e.g. when you don't know what the result actually is.

If you know there are no items which satisfy the predicate or when the result of a request is empty you should return an empty collection.

```
return Enumerable.Empty<Customer>();
```

Conclusion

1. Null values increase the maintenance costs and impose defensive programming style with null checks everywhere in the code base.
2. You can rely on the Null Object pattern. This pattern encapsulates the do-nothing approach, but the pattern is limited.
3. “Maybe” monad. Encapsulates null values.
4. we can use special tools like Fody.NullGuard.
5. Use static analysis tools like ReSharper, SonarLint, PVS-Studio.
6. Return an empty collection if you need to return a collection and there is “nothing” to return.