Clean Code

Class Design

Single Responsibility Principle (SRP)
A class should have one, and only one, reason to change.

Open Closed Principle (OCP)
You should be able to extend a classes behavior, without modifying it.

Liskov Substitution Principle (LSP)
Derived classes must be substitutable for their base classes.

Interface Segregation Principle (ISP)
Make fine grained interfaces that are client specific.

Dependency Inversion Principle (DIP)
Depend on abstractions, not on concretions.

Package Cohesion

Release Reuse Equivalency Principle (RREP)
The granule of reuse is the granule of release.

Common Closure Principle (CCP)
Classes that change together are packaged together.

Common Reuse Principle (CRP)
Classes that are used together are packaged together.

Package Coupling

Acyclic Dependencies Principle (ADP)
The dependency graph of packages must have no cycles

Stable Dependencies Principle (SDP)
Depend in the direction of stability.

Stable Abstractions Principle (SAP)
Abstractness increases with stability.

General

Follow Standard Conventions
Coding-, Architecture-, Design-Guidelines (check them with tools)

Keep it simple, stupid (KISS)
Simpler is always better. Reduce complexity as much as possible.

Boy Scout Rule
Leave the campground cleaner than you found it.

Root Cause Analysis
Always look for the root cause of a problem. Otherwise, it will get you again and again.

Multiple Languages In One Source File
XML, HTML, XAML, English, German, JavaScript, …

Environment

Project Build Requires Only One Step
Check out and then build with a single command

Executing Tests Requires Only One Step
Run all unit tests with a single command

Source Control System
Always use a source control system.

Continuous Integration
Assure integrity with Continuous Integration

Overridden Safeties
Do not override Warnings, Errors, Exception Handling – they will catch you.

Dependency Injection

Decouple Construction from Runtime
Decoupling the construction phase completely from the runtime helps to simplify unit tests.

Design

Keep Configurable Data At High Levels
If you have a constant such as default or configuration value that is known and expected at a high level of abstraction, do not bury it in a low-level function. Expose it as an argument to the low-level function called from the high-level function.

Don't Be Arbitrary
Have a reason for the way you structure your code, and make sure that reason is communicated by the structure of the code. If a structure appears arbitrary, others will feel empowered to change it.

Be Precise
When you make a decision in your code, make sure you make it precisely. Know why you have made it and how you will deal with any exceptions.

Structure Over Convention
Enforce design decisions with structure over convention.
Naming conventions are good, but they are inferior to structures that force compliance.

Prefer Polymorphism To If/Else Or Switch/Case
"ONE SWITCH": There may be no more than one switch statement for a given type of selection. The cases in that switch statement must create polymorphic objects that take the place of other such switch statements in the rest of the system.

Symmetry / Analogy
Favour symmetric designs (e.g. Load – Safe) and designs the follow analogies (e.g. same design as found in .NET framework).

Misplaced Responsibility
Something put in the wrong place.

Code At Wrong Level Of Abstraction
Functionality is at wrong level of abstraction, e.g. a PercentageFull property on a generic IStack<T>.

Fields Not Defining State
Fields holding data that does not belong to the state of the instance but are used to hold temporary data. Use local variables or extract to class abstraction the performed action.

Over Configurability
Prevent configuration just for the sake of it – or because nobody can decide how it should be. Otherwise, this will result in too complex, instabile systems.

Dependencies

Avoid Transitive Navigation
aka Law of Demeter, Writing shy code
A module should know only its direct dependencies.

Make Logical Dependencies Physical
If one module depends upon another, that dependency should be physical, not just logical. Don't make assumptions.

Singletons / Service Locator
Use dependency injection. Singletons hide dependencies.

Base Classes Depending On Their Derivatives
Base classes should work with any derived class.

Too Much Information
minimize interface to minimize coupling

Feature Envy
The methods of a class should be interested in the variables and functions of the class they belong to, and not the variables and functions of other classes. When a method uses accessors and mutators of some other object to manipulate the data within that object, then it envies the scope of the class of that other object. It wishes that it were inside that other class so that it could have direct access to the variables it is manipulating.

Artificial Coupling
Things that don't depend upon each other should not be artificially coupled.

Hidden Temporal Coupling
If for example the order of some method calls is important then make sure that they cannot be called in the wrong order.

Naming

Choose Descriptive / Unambiguous Names
Names have to reflect what a variable, field, property stands for. Names have to be precise.

Choose Names At Appropriate Level Of Abstraction
Choose names that reflect the level of abstraction of the class or method you are working in.

Name Interfaces After Functionality They Abstract
The name of an interface should be derived from ist usage by the client, like IStream.

Name Classes After Implementation
The name of a class should reflect how it fulfills the functionality provided by its interface(s), like MemoryStream : IStream

Name Methods After What They Do
The name of a method should describe what is done, not how it is done.

Use Long Names For Long Scopes
fields → parameters → locals → loop variables
long → short

Names Should Describe Side-Effects
Names have to reflect the entire functionality.

Avoid Encodings In Names
No prefixes, no type/scope information

Use Standard Nomenclature Where Possible

Understandability

Consistency
If you do something a certain way, do all similar things in the same way: same variable name for same concepts, same naming pattern for corresponding concepts

Poorly Written Comment
Comment does not add any value (redundant to code), is not well formed, not correct grammar/spelling

Obscured Intent
Too dense algorithms that loose all expressivness.

Obvious Behaviour Is Unimplemented
Violations of "the Principle of Least Astonishment"
What you expect is what you get

Hidden Logical Dependency
A method can only work correctly when invoked correctly depending on something else in the same class, e.g. a DeleteItem method must only be called if a CanDeleteItem method returned true, otherwise it will fail.

Methods

Methods Should Do One Thing
loops, exception handling, …
encapsulate in sub-methods

Methods Should Descend 1 Level Of Abstraction
The statements within a method should all be written at the same level of abstraction, which should be one level below the operation described by the name of the function.

Method With Too Many Arguments
Prefer less arguments. Maybe functionality can be outsourced to dedicated class that holds the information in fields.

Method With Out/Ref Arguments
Prevent usage. Return complex object holding all values, split into several methods. If your method must change the state of something, have it change the state of the object it is called on.

Selector / Flag Arguments
public int Foo(bool flag)
split method into several independent methods that can be called from the client without the flag.

Inappropriate Static
Static method that should be an instance method.

Exception Handling

Catch Specific Exceptions
Catch exceptions as specific as possible. Catch only the exceptions for which you can react meaningful.

Catch Exceptions Where You Can React Meaningful
Only catch exceptions when you can react in a meaningful way. Otherwise, let someone up in the call stack react to it.

Using Exceptions for Control Flow
Using exceptions for control flow: has bad performance, is hard to understand, results in very hard handling of real exceptional cases.

Swallowing Exceptions
Exceptions can be swallowed only if the exceptional case is completely resolved after leaving the catch block. Otherwise, the system is left in an inconsistent state.

Source Layout

Vertical Separation
Variables and methods should be defined close to where they are used. Local variables should be declared just above their first usage and should have a small vertical scope.

Use Explanatory Variables
Use locals to give steps in algorithms names.

Nesting
Nested code should be more specific or handle less probable scenarios than unnested code.

Separate Multi-Threading Code
Do not mix code that handles multi-threading aspects with the rest of the code. Separate them into different classes.

Conditionals

Encapsulate Conditionals
if (this.ShouldBeDeleted(timer)) is preferable to if (timer.HasExpired && !timer.IsRecurrent)

Avoid Negative Conditionals
Negative conditionals are harder to read than positive conditionals.

Encapsulate Boundary Conditions
Boundary conditions are hard to keep track of. Put the processing for them in one place.
E.g. nextLevel = level + 1;

Useless Stuff

Dead Comment, Code
Just delete not used things

Clutter
Code that is not dead but does not add any functionality.

Inappropriate Information
Comment holding information better held in a different kind of system: product backlog, source control
Use comments for technical notes only.

Maintainability Killers

Duplication
Eliminate duplication. Violation of the „Don't repeat yourself" (DRY) principle.

Magic Numbers / Strings
Replace Magic Numbers with named constants.

Enums (persistent or defining behaviour)
Use reference codes instead of enums if they have to be persisted.
Use polymorphism instead of enums if they define behaviour.

Practices | Smells

V 1.3

Author: Urs Enzler

bbv Software Services AG