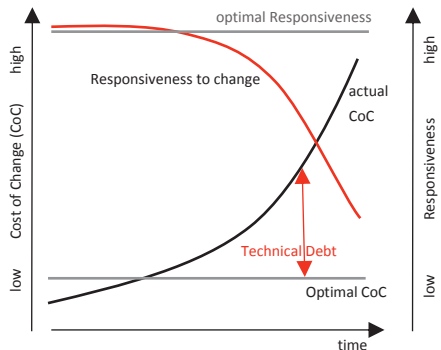


Clean Code Cheat Sheet

Why Clean Code

Code is clean if it can be understood easily – by everyone on the team. With understandability comes readability, changeability, extensibility and maintainability. All the things needed to keep a project going over a long time without accumulating up a large amount of technical debt.



Writing clean code from the start in a project is an investment in keeping the cost of change as constant as possible throughout the lifecycle of a software product. Therefore, the initial cost of change is a bit higher when writing clean code (grey line) than quick and dirty programming (black line), but is paid back quite soon. Especially if you keep in mind that most of the cost has to be paid during maintenance of the software. Unclean code results in technical debt that increases over time if not refactored into clean code. There are other reasons leading to Technical Debt such as bad processes and lack of documentation, but unclean code is a major driver. As a result, your ability to respond to changes is reduced (red line).

In Clean Code, Bugs Cannot Hide

Most software defects are introduced when changing existing code. The reason behind this is that the developer changing the code cannot fully grasp the effects of the changes made. Clean code minimises the risk of introducing defects by making the code as easy to understand as possible.

Principles

Loose Coupling +

Two classes, components or modules are coupled when at least one of them uses the other. The less these items know about each other, the looser they are coupled.

A component that is only loosely coupled to its environment can be more easily changed or replaced than a strongly coupled component.

High Cohesion +

Cohesion is the degree to which elements of a whole belong together. Methods and fields in a single class and classes of a component should have high cohesion. High cohesion in classes and components results in simpler, more easily understandable code structure and design.

Change is Local +

When a software system has to be maintained, extended and changed for a long time, keeping change local reduces involved costs and risks. Keeping change local means that there are boundaries in the design which changes do not cross.

It is Easy to Remove +

We normally build software by adding, extending or changing features. However, removing elements is important so that the overall design can be kept as simple as possible. When a block gets too complicated, it has to be removed and replaced with one or more simpler blocks.

Mind-sized Components

Break your system down into components that are of a size you can grasp within your mind so that you can predict consequences of changes easily (dependencies, control flow, ...).

Smells

Rigidity -

The software is difficult to change. A small change causes a cascade of subsequent changes.

Fragility -

The software breaks in many places due to a single change.

Immobility -

You cannot reuse parts of the code in other projects because of involved risks and high effort.

Viscosity of Design -

Taking a shortcut and introducing technical debt requires less effort than doing it right.

Viscosity of Environment -

Building, testing and other tasks take a long time. Therefore, these activities are not executed properly by everyone and technical debt is introduced.

Needless Complexity -

The design contains elements that are currently not useful. The added complexity makes the code harder to comprehend. Therefore, extending and changing the code results in higher effort than necessary.

Needless Repetition -

Code contains exact code duplications or design duplicates (doing the same thing in a different way). Making a change to a duplicated piece of code is more expensive and more error-prone because the change has to be made in several places with the risk that one place is not changed accordingly.

Opacity -

The code is hard to understand. Therefore, any change takes additional time to first reengineer the code and is more likely to result in defects due to not understanding the side effects.

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 behaviour without modifying it.

Liskov Substitution Principle (LSP) +

Derived classes must be substitutable for their base classes.

Dependency Inversion Principle (DIP) +

Depend on abstractions, not on concretions.

Interface Segregation Principle (ISP) +

Make fine grained interfaces that are client-specific.

Classes Should be Small +

Smaller classes are easier to grasp. Classes should be smaller than about 100 lines of code. Otherwise, it is hard to spot how the class does its job and it probably does more than a single job.

Do stuff or know others, but not both

Classes should either do stuff (algorithm, read data, write data, ...) or orchestrate other classes. This reduces coupling and simplifies testing.

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.

Multiple Languages in One Source File -

C#, Java, JavaScript, XML, HTML, XAML, English, German ...

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 the runtime behaviour.

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 – Save) and designs that follow analogies (e.g. same design as found in .NET framework).

Separate Multi-Threading Code +

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

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 a class abstracting 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 overly complex, unstable systems.

Micro Layers -

Do not add functionality on top, but simplify overall.

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 -

Minimise interface to minimise 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. Using accessors and mutators of some other object to manipulate its data, is envying the scope of the other object.

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.

Transitive Navigation -

Aka Law of Demeter, writing shy code. A module should know only its direct dependencies.

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 its usage by the client.

Name Classes After How They Implement Interfaces +

The name of a class should reflect how it fulfils the functionality provided by its interface(s), such as 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 Describe Side Effects +

Names have to reflect the entire functionality.

Standard Nomenclature Where Possible +

Don't invent your own language when there is a standard.

Encodings in Names -

No prefixes, no type/scope information

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.
Use Explanatory Variables +	Use locals to give steps in algorithms names.
Encapsulate Boundary Conditions +	Boundary conditions are hard to keep track of. Put the processing for them in one place, e.g. <code>nextLevel = level + 1;</code>
Prefer Dedicated Value Objects to Primitive Types +	Instead of passing primitive types like strings and integers, use dedicated primitive types: e.g. <code>AbsolutePath</code> instead of string.
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 lose all expressiveness.
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 when invoked correctly depending on something else in the same class, e.g. a <code>DeleteItem</code> method must only be called if a <code>CanDeleteItem</code> 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 fewer arguments. Maybe functionality can be outsourced to a 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 <code>Foo(bool flag)</code> 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
Source Code Structure	
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.
Nesting +	Nested code should be more specific or handle less probable scenarios than unnested code.
Structure Code into Namespaces by Feature +	Keep everything belonging to the same feature together. Don't use namespaces communicating layers. A feature may use another feature; a business feature may use a core feature like logging.

Conditionals	
Encapsulate Conditionals +	if (<code>this.ShouldBeDeleted(timer)</code>) is preferable to <code>if (timer.HasExpired && !timer.IsRecurrent)</code> .
Positive Conditionals +	Positive conditionals are easier to read than negative conditionals.
Useless Stuff	
Dead Comment, Code -	Delete unused things. You can find them in your version control system.
Clutter -	Code that is not dead but does not add any functionality
Inappropriate Information -	Comment holding information better held in a different system: product backlog, source control. Use code 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 and Strings with named constants to give them a meaningful name when meaning cannot be derived from the value itself.
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.
Tangles	The class dependencies should not be tangled. There should be no cyclic dependency chains. In a cycle there is no point to start changing the code without side-effects.
Exception Handling	
Catch Specific Exceptions +	Catch exceptions as specific as possible. Catch only the exceptions for which you can react in a meaningful manner.
Catch Where You Can React in a Meaningful Way +	Only catch exceptions when you can react in a meaningful way. Otherwise, let someone up in the call stack react to it.
Use Exceptions instead of Return Codes or null +	In an exceptional case, throw an exception when your method cannot do its job. Don't accept or return null. Don't return error codes.
Fail Fast +	Exceptions should be thrown as early as possible after detecting an exceptional case. This helps to pinpoint the exact location of the problem by looking at the stack trace of the exception.
Using Exceptions for Control Flow -	Using exceptions for control flow: has bad performance, is hard to understand and 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.

From Legacy Code to Clean Code	
Always have a Running System +	Change your system in small steps, from a running state to a running state.
1) Identify Features +	Identify the existing features in your code and prioritise them according to how relevant they are for future development (likelihood and risk of change).
2) Introduce Boundary Interfaces for Testability +	Refactor the boundaries of your system to interfaces so that you can simulate the environment with test doubles (fakes, mocks, stubs).
3) Write Feature Acceptance Tests +	Cover a feature with Acceptance Tests to establish a safety net for refactoring.
4) Identify Components +	Within a feature, identify the components used to provide the feature. Prioritise components according to relevance for future development (likelihood and risk of change).
5) Refactor Interfaces between Components +	Refactor (or introduce) interfaces between components so that each component can be tested in isolation of its environment.
6) Write Component Acceptance Tests +	Cover the features provided by a component with Acceptance Tests.
7) Decide for Each Component: Refactor, Reengineer, Keep +	Decide for each component whether to refactor, reengineer or keep it.
8a) Refactor Component +	Redesign classes within the component and refactor step by step (see Refactoring Patterns). Add unit tests for each newly designed class.
8b) Reengineer Component +	Use ATDD and TDD (see Clean ATDD/TDD cheat sheet) to re-implement the component.
8c) Keep Component +	If you anticipate only few future changes to a component and the component had few defects in the past, consider keeping it as it is.
Refactoring Patterns	
Reconcile Differences – Unify Similar Code +	Change both pieces of code stepwise until they are identical. Then extract.
Isolate Change +	First, isolate the code to be refactored from the rest. Then refactor. Finally, undo isolation.
Migrate Data +	Move from one representation to another by temporary duplication of data structures.
Temporary Parallel Implementation +	Refactor by introducing a temporary parallel implementation of an algorithm. Switch one caller after the other. Remove old solution when no longer needed. This way you can refactor with only one red test at a time.
Demilitarized Zone for Components +	Introduce an internal component boundary and push everything unwanted outside of the internal boundary into the demilitarized zone between component interface and internal boundary. Then refactor the component interface to match the internal boundary and eliminate the demilitarized zone.
Refactor before adding Functionality	Refactor the existing code before adding new functionality in a way so that the change can easily be made.
Small Refactorings	Only refactor in small steps with working code in-between so that you can keep all loose ends in your head. Otherwise, defects sneak in.

How to Learn Clean Code	
Pair Programming +	Two developers solving a problem together at a single workstation. One is the driver, the other is the navigator. The driver is responsible for writing the code. The navigator is responsible for keeping the solution aligned with the architecture, the coding guidelines and looks at where to go next (e.g. which test to write next). Both challenge their ideas and approaches to solutions.
Commit Reviews +	A developer walks a peer developer through all code changes prior to committing (or pushing) the changes to the version control system. The peer developer checks the code against clean code guidelines and design guidelines.
Coding Dojo +	In a Coding Dojo, a group of developers come together to exercise their skills. Two developers solve a problem (kata) in pair programming. The rest observe. After 10 minutes, the group rotates to build a new pair. The observers may critique the current solution, but only when all tests are green.
Bibliography	
	Clean Code: A Handbook of Agile Software Craftsmanship by Robert Martin
	Code Simplicity by Max Kanat-Alexander

Legend:

DO

DON'T



This work by Urs Enzler is licensed under a Creative Commons Attribution 4.0 International License.

Urs Enzler www.bbv.ch February 2015 V2.5

