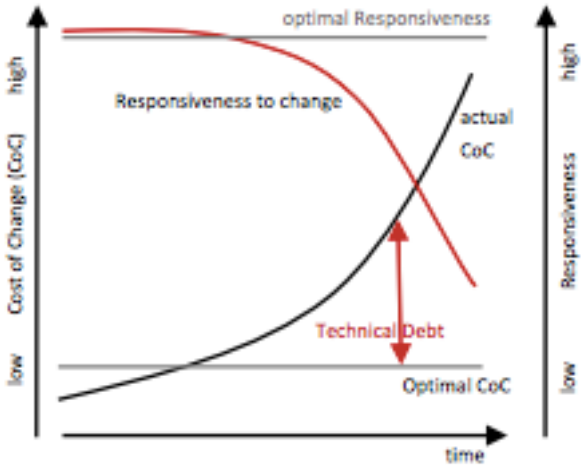
Why Clean Code

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).

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.	+
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.	+
Smells	
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.	-

General	
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 …	-
Symmetry / Analogy	
Favour symmetric designs (e.g. Load – Save) and designs that follow analogies (e.g. same design as found in .NET framework).	+
Dependencies	
Base Classes Depending On Their Derivatives	
Base classes should work with any derived class.	-
Too Much Information	
Minimise interface to minimise coupling	-
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.	+
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.	+
Methods	
Methods Should Do One Thing	
Loops, exception handling, … encapsulate in sub-methods	+
Method with Too Many Arguments	
Prefer fewer arguments. Maybe functionality can be outsourced to a dedicated class that holds the information in fields.	-

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.	+
Conditionals	
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.	-
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.	-
Exception Handling	
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.	+
From Legacy Code to Clean Code	
Write Feature Acceptance Tests	
Cover a feature with Acceptance Tests to establish a safety net for refactoring.	+
Refactoring Patterns	
Isolate Change	
First, isolate the code to be refactored from the rest. Then refactor. Finally, undo isolation.	+

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.	+
Kinds of Automated Tests	
DDT – Defect Driven Testing	
Write a unit test that reproduces the defect – Fix code – Test will succeed – Defect will never return.	+
Design for Testability	
Constructor – Simplicity	
Objects have to be easily creatable. Otherwise, easy and fast testing is not possible.	+
Constructor – Lifetime	
Pass dependencies and configuration/parameters into the constructor that have a lifetime equal to or longer than the created object. For other values use methods or properties.	+
Don’t Assume	
Understand the Algorithm	
Just working is not enough, make sure you understand why it works.	+
Faking (Stubs, Fakes, Spies, Mocks, Test Doubles …)	
Isolation from environment	
Use fakes to simulate all dependencies of the testee.	+
Unit Test Principles	
Test Checking More than Necessary	
A test that checks more than it is dedicated to. The test fails whenever something changes that it checks unnecessarily. Especially probable when fakes are involved or checking for item order in unordered collections.	+
TDD Principles	

Skipping Something Too Easy to Test	
Don’t assume, check it. If it is easy, then the test is even easier	-
Skipping Something Too Hard to Test	
Make it simpler, otherwise bugs will hide in there and maintainability will suffer.	-
Red Bar Patterns	
Learning Test	
Write tests against external components to make sure they behave as expected.	+
Green Bar Patterns	
Fake It (‘Til You Make It)	
Return a constant to get first test running. Refactor later.	+
Obvious Implementation	
If the implementation is obvious then just implement it and see if test runs. If not, then step back and just get test running and refactor then.	+
One to Many – Drive Collection Operations	
First, implement operation for a single element. Then, step to several elements (and no element).	+
Acceptance Test Driven Development	
Component Acceptance Tests	
Write acceptance tests for individual components or subsystems so that these parts can be combined freely without losing test coverage.	+
Continuous Integration	
Pre-Commit Check	
Run all unit and acceptance tests covering currently worked on code prior to committing to the source code repository.	+
Post-Commit Check	
Run all unit and acceptance tests on every commit to the version control system on the continuous integration server	+