## Why Clean Code

Cleaning 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 the cost of change is typically 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. But 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 minimizes the risk of introducing defects by making the code as easy to understand as possible.

### Principles

<table>
<thead>
<tr>
<th>Loose Coupling</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>High Cohesion</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

### Change is Local

When a software system has to be maintained, extended and changed for a long time, keeping change local requires 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

<table>
<thead>
<tr>
<th>Rigidity</th>
</tr>
</thead>
<tbody>
<tr>
<td>The software is difficult to change. A small change causes a cascade of subsequent changes.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fragility</th>
</tr>
</thead>
<tbody>
<tr>
<td>The software breaks in many places due to a single change.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Immobility</th>
</tr>
</thead>
<tbody>
<tr>
<td>You cannot reuse parts of the code in other projects because of involved risks and high effort.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Viscosity of Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taking a shortcut and introducing technical debt requires less effort than doing it right.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Viscosity of Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building, testing and other tasks take a long time. Therefore, these activities are not executed properly by everyone and technical debt is introduced.</td>
</tr>
</tbody>
</table>

### Needless Repetition

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

## Code at Wrong Level of Abstraction

- **Functionality** is at wrong level of abstraction, e.g. a `PercentAgregateFactor` property on a generic `Ticket` class.

### 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 a module depends upon another, that dependency should be physical and not just logical. Don’t make assumptions.

### Singletons / Service Locator

Use dependency injection. Singleton 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 classes they depend on, and the variables and functions of other classes. Using accessors and mutators of some other object to manipulate its data, is enyong 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

At a Law of Demeter, writing the 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 descriptive.

Choose Names at Appropriate Level of Abstraction

Choose names that reflect the level of abstraction of the class or method you are working on.

### 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 fulfills the functionality provided by its interface(s), e.g. `MemoryStream : IStream`

### Name Methods After What They Do

The name of the method should describe what is done, not how it is done.

### Use Long Names for Long Scopes

Fields `parameters` to `locals` to `loop variables` long `short loop variables`.

### 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.
Clean Code Cheat Sheet

Understandability

Consistency
If you do something a certain way, do all similar things in the same way.

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. public static boundary().

Prefer Dedicated Value Objects to Primitive Types
Instead of using primitive types like strings and integers, use dedicated primitive types: e.g. AbsolutePath instead of string.

Poorly Written Comment
Comment does not add any value (redundant to code), is not well formatted, does not correctly 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 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, etc. 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 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

Source Code Structure

Vertical Separation
Variables and methods should be defined close to where they are used.

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 (this.IsMyFavorite(timer)) return Value.A; else return Value.B;

Positive Conditionals
Postive conditionals are easier to read than negative conditionals.

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.

Mantainability 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.

Fast Fail
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
Use 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. Priorise 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 ATOD and TDD (see Clean ATOD/TDD cheat sheet) to re-implement the component.

8c) Keep Component
If you anticipate only few future changes to a component and the component has no defects in the past, consider keeping it as 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 changes.

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.

Comment 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

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

Urs Enzler  www.bbv.ch  October 2014  V2.4
Clean
Always unit test boundaries. Do not assume behaviour.
Understand the Algorithm
Don’t Assume
Always use the same name for variables holding uninteresting arguments to
Anonymous Variables
Give the variable holding the result of a
Resource Files
Use a pattern that
Test Namespace
Create a test assembly for each production assembly and name it after the
Abstraction Layers at System Boundary
Use abstraction layers at system boundaries (database, file system, web services, …) that simplify unit testing by enabling the usage of fakes.
Structure
Arrange – Act – Assert
Structure the tests always by AAA. Never mix these three blocks.
Test Assemblies
Create a test assembly for each production assembly and name it as the
Setup / TearDown for Infrastructure Only
Use the Setup / TearDown methods only for infrastructure that your unit tests do not need. Do not use it for anything that is under test.
Test Method Naming
Use a pattern that reflects behaviour of tested code, e.g. Behaviour / (Trigger) / When(Condition) with [] as optional parts.
Resource Files
Test and resource are together: FooTest.cs, FooTest.resx
Naming
Naming SUT Test Variables
Give the variable holding the System Under Test always the same name (e.g. testee or test). Clear naming the SUT, robust against refactoring.
Naming Result Values
Give the variable holding the result of the tested method always the same name (e.g. result).
Anonymous Variables
Always use a single name for variables holding uninteresting arguments to
Don’t Assume
Understand the Algorithm
Just working is not enough, make sure you understand why it works.
Incorrect Behaviour at Boundaries
Always unit test boundaries. Do not assume behaviour.
Faking (Stubs, Fakes, Spies, Mocks, Test Doubles …)
Isolation from environment
Use fakes to simulate all dependencies of the testee.
Faking Framework
Use a dynamic fake framework for fakes that show different behaviour in
different test scenarios (little behaviour reuse).
Manually Written Fakes
Use manually written fakes when they can be used in several tests and they have only little changed behaviour in these scenarios (behaviour reuse).
Mixing Stubbing and Expectation Declaration
Make sure that you follow the AAA (Arrange, act, assert) syntax when using fakes. Done by setting up stubs (so that the testee can run) with expectations on what the testee should do in the same code block.
Checking Fakes instead of Testee
Tests that do not check the testee but values returned by fakes. Normally due to excessive fake usage.
Excessive Fake Usage
If your test needs a lot of fakes or fake setup, then consider splitting the testee into several classes or provide an additional abstraction between your testee and its dependencies.
Unit Test Principle
Fast
Unit tests have to be fast in order to be executed often. Fast means much smaller than seconds.
Isolated
Isolated testee: Clear where the failure happened. Isolated test: No dependency between tests (random order).
Repeatable
No assumed initial state, nothing left behind, no dependency on external services that might be unavailable (databases, file system ...).
Self-Validating
No manual test interpretation or intervention. Red or green!
Timely
Tests are written at the right time (TDD, DOT, POUTing)
Testable
Test Not Testing Anything
Passing test that at first sight appears valid but does not test the testee.
Test Needing Exceptional Setup
A test that needs dozens of lines of code to set up its environment. This test uses a set of assertions to what is really tested.
Too Large Test / Assertions for Multiple Scenarios
A valid test that is, however, too large. Reasons can be that this test checks for more than one feature or the testee does more than one thing (violation of Single Responsibility Principle).
Checking Internals
A test that accesses internals (private/protected members) of the testee directly (Reflection). This is a refactoring killer.
Test Only Running on Developer’s Machine
A test that is dependent on the development environment and fails elsewhere. Use continuous integration to catch them as soon as possible.
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.
Irrelevant Information
Test contains information that is not relevant to understand it.
Chatty Test
A test that fills the console with text – probably used once to manually check for something.
Test Swallowing Exceptions
A test that catches exceptions and lets the test pass.
Test Not Belonging in Host Test Fixture
A test that tests a completely different testee than all other tests in the fixture.
Obsolete Test
A test that checks something no longer required in the system. May even prevent clean-up of production code because it is still referenced.
Hidden Test Functionality
Test functionality hidden in either the SetUp method, base class or helper class. The test should be clearly by looking at the test method only – no installation or asserts somewhere else.
Bloatd Construction
The construction of dependencies and arguments used in calls to testee makes test hard to read. Extract helper to methods that can be reused.
Unclear Fail Reason
Split test or use assertion messages.
Conditional Test Logic
Tests should not have any conditional test logic because it’s hard to read.
Erratic Test
Sometimes passes, sometimes fails due to left overs or environment.
TDD Principles
A Test Checks One Feature
A test checks exactly one feature of the testee. That means it tests all things included in this feature but not more. This includes probably more than one call to the testee. This way, the tests serve as samples and documentation of the usage of the testee.
Tiny Steps
Make tiny little steps. Add only a little code in test before writing the required production code. Then repeat. Add only one assert per step.
Keep Tests Simple
Whenever a test gets complicated, check whether you can split the testee into several classes (Single Responsibility Principle).
Prefer State Verification to Behaviour Verification
Use behaviour verification only if there is no state to verify. Refactoring is easier due to less coupling to implementation.
Test Domain Specific Language
Use test DSLs to simplify reading tests: builders to create test data using fluent API, assertion helpers for concise assertions.
TDD Process Smells
Using Code Coverage as a Goal
Use code coverage to find missing tests but don’t use it as a driving tool. Otherwise, the result could be tests that increase code coverage but not correctness.
No Green Bar in the last ~10 Minutes
Make small steps to get feedback as fast and frequent as possible.
Not Running Test Before Writing Production Code
If the test fails, then new code is required. Additionally, if the test surprisingly does not fail then make sure the test is correct.
Not Spending Enough Time on Refactoring
Refactoring is an investment in the future. Readability, changeability and extensibility will pay back.
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.
Organising Tests Around Methods, Not Behaviour
These tests are not test setters and getters in isolation, test the scenario they are used in.
Red Bar Patterns
One Step Test
Pick a test you are confident you can implement and which maximises learning effect (e.g. impact on design).
Partial Test
Write a test that does not fully check the required behaviour, but brings you a step closer to it. Then use Extend Test below.
Extend Test
Extend an existing test to better match real-world scenarios.
Another Test
If you think of new tests, then write them on the TDD DO list and don’t lose focus on current test.
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 just get first test running. Refactor later.
Triangulate – Drive Abstraction
Write test with at least two sets of sample data. Abstract implementation on these.
Obvious Implementation
If the implementation is obvious then just implement it and see if tests run. 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
Use Acceptance Tests to Drive Your TDD tests
Acceptance tests check for the required functionality. Let them guide your TDD.
User Feature Test
An acceptance test is a test for a complete user feature from top to bottom that provides business value.
Automated ATDD
Use automated Acceptance Test Driven Development for regression testing and executable specifications.
Component Acceptance Tests
Write acceptance tests for individual components or subsystems so that these parts can be combined freely without losing test coverage.
Simulate System Boundaries
Simulate system boundaries like the user interface, database, file system and external services to speed up your acceptance tests and to be able to check exceptional cases (e.g. a full hard disk). Use system tests to check the boundaries.
Acceptance Test Spree
Do not write acceptance tests for every possibility. Write acceptance tests only for real scenarios. The exceptional and theoretical cases can be covered more easily with unit tests.
ATDD/TDD Cheat Sheet

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.
- Communicate Failed Integration to Whole Team
  Whenever a stage on the continuous integration server fails, notify whole team in order to get blocking situation resolved as soon as possible.
- Build Staging
  Split the complete continuous integration workflow into individual stages to reduce feedback time.
- Automatically Build an Installer for Test System
  Automatically build an installer as often as possible to test software deployment to production environment is automated to prevent manual mistakes.
- Continuous Deployment
  Install the system to a test environment on every commit/push and on manual request. Deployment to production environment is automated to prevent manual mistakes, too.

Test Pyramid
- Constraint Test = Test for non-functional requirements.
- Exploratory Testing
 -manually executed
- System and Constraint Tests
  automatically executed
- Other Tests
  tests not practical to automate
- Acceptance Tests
- Actively driven development
- Acceptance Tests
- Manually executed

ATDD/TDD Cycle
- Write acceptance criteria for user story
  The whole team defines acceptance criteria for user stories.
- Define examples
  The whole team defines examples for acceptance criteria used to show that code works.
- Write acceptance test skeleton
  Map the examples into an empty specification/test in your acceptance test framework (Gherkin, MSpec/expect classes and statements ...).
- Explore design
  Implement a Spike to gather enough knowledge so you can design a possible solution.
- Make an initial design
  Roughly design how you want to implement the new functionality, especially the interface for your acceptance test (how to call and verify functionality).
- Refactor
  Refactor existing code to simplify introduction of new functionality. Run all tests to keep code working.
- Write an acceptance test
  Add arrange, act and assert parts to the acceptance test skeleton (Given, When, Then or Establish, Because, It ...).
- Run all acceptance tests
  Succeeded and all examples tested
- Run acceptance test
  You have no class design idea
  The failing test should state what went wrong so you don't have to debug the code.
- Build Staging
  Succeeded, not all acceptance tests implemented yet
- Clean up code
  Apply clean code guidelines. Redesign classes as needed. (< 10 minutes).
- Make error reason obvious
  You have a class design idea
  Succeeded, updated acceptance tests.

Legend:
- DO
- DON'T

Bibliography
- Test Driven Development: By Example by Kent Beck
- ATDD by Example: A Practical Guide to Acceptance Test-Driven Development by Markus Gärtner
- The Art of Unit testing by Roy Osherove
- xUnit Test Patterns: Refactoring Test Code by Gerard Meszaros

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