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 made up quite soon. Especially if you keep in mind that the cost has to be paid during maintenance of the software. Unclen 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 minimizes 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.

- High Cohesion
  - Cohesion is the degree to which elements of a whole belong together.
  - Methods and fields in a single class and components of a class 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.

Legend:

- DO
- DON'T

Fields Not Defining State

- Fields holding data that do 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.

Dependability

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

- Singleton / 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. When a method uses attributes 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.

Navigational

- Akka 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, such as Stream.

- Name Classes After How They Implement Their Interfaces
  - The name of a class should reflect how it fulfills the functionality provided by its interface(s), such as MemoryStream : Stream

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

- Long names make a method easier to understand.

Name Describes 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

### From Legacy Code to Clean Code

Always have a Running System

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, simulators).

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 decide only a few future changes to a component and the component had few 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.

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.

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.

### 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.

Code 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

Legend:

- **DO**
- **DON'T**
Clean ATDD/TDD

Kinds of Automated Tests
- ATDD – Acceptance Test Driven Development
  Specify a feature first with a test, then implement.
- TDD – Test Driven Development
- DDT – Defect Defect Testing
  Write a unit test that reproduces the defect – Fix code – Test will succeed – Defect will never occur.
- POUTing – Plain Old Unit Testing
  Akka after. Write unit tests to check existing code. You cannot and probably do not want to test drive everything. Use POUT to increase sanity.

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.

Abstraction Layers at System Boundary
- Use abstraction layers at system boundaries (database, file system, web services, COM interfaces …) that simplify unit testing by enabling the usage of mocks.

Structure
- Arrange – Act – Assert
  Structure the tests always by AAA. Never mix these three blocks.

Test Assemblies (Net)
- Create a test assembly for each production assembly and name it as the production assembly e.g. “Test”.
- Test Namespace
  Put the tests in the same namespace as their associated testee.

Unit Test Methods Show Whole Truth
- Unit test methods show all parts needed for the test. Do not use SetUp methods or classes to perform actions on testee or dependencies.

Setup / TearDown for Infrastructure Only
- Use the SetUp / TearDown methods only for infrastructure that your unit test needs. Do not use it for anything that is under test.

Test Method Naming
- Use descriptive names, e.g. FeatureWhenScenarioThenBehaviour
- Single Scenario per Test
  One test checks one scenario only.

Resource Files
- Test and resource are together: FooTest.cs, FooTest.resx
- Test Name
  Name software tests. Test Not Testing Anything.
- Pass Testing that at first sight appears valid but does not test the testee.
- Needing Excessive Setup
  A test that needs dozens of lines of code to set up its environment. This also makes it hard to see what is really tested.
- 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.
- 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.
- 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 tests are involved or checking for item order in unordered collections.
- Irrelevant Information
  Test contains information that is not relevant to understand it.
- Chatty Test
  Test output is noisy and makes it hard to understand what is going on.
- Swallowing Exceptions
  A test that catches exceptions and lets the test pass.
- 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 by looking at the test method only – no installation or asserts somewhere else.
- Bloat Construction
  The construction of dependencies and arguments used in calls to testee makes test code hard to read. Extract to helper 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.
- Test Logic in Production Code
  Tests depend on special logic in production code.
- Erratic Test
  Sometimes passes, sometimes fails due to left over errors or environment.
- TDD Principle
  Test Checks One Feature
  A test checks exactly one feature of the testee. That means that 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.

Simulate System Boundaries
- Partial Test
  Tests are written at the right time (TDD, DDT, POUTing) but may not be finished.
- Isolated
  Clear where the failure happened. No dependency between tests (random order).
- Repeatability
  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).

Technical Smells
- Write test with at least two sets of sample data. Abstract implementation of Single Responsibility Principle).
- Obvious Implementation
  Use test DSLs to simplify reading tests: helper methods, classes.
- Extend Test
  An acceptance test is a test for a complete user feature from top to bottom that provides business value.
- One to Many – Drive Collection Operations
  First, implement operation for a single element. Then, step to several elements.
- Acceptance Test Smell
  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 can be combined freely without losing test coverage.

Simulate System Boundaries
- Simulate system boundaries like the user interface, databases, 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.

Legend:
- DO
- DONT
**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 on a test system (for manual tests, or tests with real hardware).

**Continuous Deployment**
Install the system to a test environment on every commit or manual request. Deployment to production environment is automated, too.

---

**Test Pyramid**

- Unit Tests
  - tests not practical to automate
  - automatically executed
- System and Constraint Tests
  - manually executed: find gaps, regression, drive development
- Acceptance Tests
  - Acceptance Test = Test for non-functional requirements.

---

**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

---

**Clean ATDD/TDD Cheat Sheet**

**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 classes and It 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 ...)

**Succeeded and all acceptance tests implemented yet**

You have all knowledge to implement the acceptance test.

**Run acceptance test**

You have no class design idea

**Failed**

Make error reason obvious

The failing test should state what went wrong so you don’t have to debug the code.

**Succeeded, not all acceptance tests implemented yet**

You have a class design idea

**Succeeded, code clean, TO DO list empty**

Succeeded, code clean, TO DO list empty

**TO DO list**

- Add missing test when you think of one
- Remove test when written

We write the TO DO list into the same file as the unit test with /// TODO:

**Write a test**

Add a minimal test or make a minimal change to an existing test (< 10 minutes).

**Run test**

Failed

Make error reason obvious

The failing test should state what went wrong so you don’t have to debug the code.

**Succeeded**

You have enough knowledge to implement the acceptance test.

**Write test**

Write a test

**Succeeded, code not clean**

Apply clean code guidelines. Redesign classes as needed. (< 10 minutes).

**Rewrite code**

Write as little code as possible to make the test pass.

**Succeeded, code clean**

Apply clean code guidelines. Redesign classes as needed. (< 10 minutes).

**Run all tests**

Succeeded

**Clean up code**

Apply clean code guidelines. Redesign classes as needed. (< 10 minutes).

---

**Legend:**

- **DO**
- **DON'T**