Why Clean Code?

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

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.

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

Visibility 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 lots of code duplication: 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 reignite 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.

Dependancy 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.

Package Cohesion
Release Reuse Equivalency Principle (RREP)
The granularity of reuse is the granularity of release. Examples re-use only actual parts of a class.

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)
Abstraction increases with stability.

Clean Code

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
C#, Java, Javascript, XML, HTML, XAML, German... (red line).

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 overpower 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 Arthritic
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 you will handle 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 symmetrical 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 Black<T>.

Clean Code Cheat Sheet

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 Entry
The method 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 accessor and mutator of some other object to manipulate the data within that object, then it envisions 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.

Transitive Navigation
Aka Law of Demeter, writing shy code.

A module should know only its direct dependences.

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 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
Fields → parameters → local → 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.

Encoding in Names
No prefixes, no type/scope information

Software Services

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 is dependent on 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 easily changed or replaced by 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 risks and costs. 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.

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 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).
<table>
<thead>
<tr>
<th>Clean Code Cheat Sheet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Methods</strong></td>
</tr>
<tr>
<td>Methods Should Do One Thing</td>
</tr>
<tr>
<td>Loops, exception handling, encapsulate in sub-methods.</td>
</tr>
<tr>
<td>Methods Should Descend 1 Level of Abstraction</td>
</tr>
<tr>
<td>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.</td>
</tr>
<tr>
<td><strong>Methods with Too Many Arguments</strong></td>
</tr>
<tr>
<td>Prefer fewer arguments. Maybe functionality can be outsourced to a dedicated class that holds the information in fields.</td>
</tr>
<tr>
<td><strong>Methods with Out/Ref Arguments</strong></td>
</tr>
<tr>
<td>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.</td>
</tr>
<tr>
<td><strong>Selector / Flag Arguments</strong></td>
</tr>
<tr>
<td>public int Foo(bool flag)</td>
</tr>
<tr>
<td><strong>Inappropriate Static</strong></td>
</tr>
<tr>
<td>Static method that should be an instance method</td>
</tr>
<tr>
<td><strong>Source Code Structure</strong></td>
</tr>
<tr>
<td>Vertical Separation</td>
</tr>
<tr>
<td>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.</td>
</tr>
<tr>
<td><strong>Nesting</strong></td>
</tr>
<tr>
<td>Nested code should be more specific or handle less probable scenarios than unrestricted code.</td>
</tr>
<tr>
<td><strong>Structure Code into Namespaces by Feature</strong></td>
</tr>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

| **Conditional** |
| **Ensure Conditionals** |
| if (this.ShouldBeDeleted(timer)) is preferable to if (timer.HasExpired & & timer.IsRecurrent). |
| **Positive Conditionals** |
| Positive conditionals are easier to read than negative conditionals. |
| **Useless Stubs** |
| 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 kind of system: product backlog, source control. Use code comments for technical notes only. |
| **Maintainability Killers** |
| Multiple exceptions, many throw statements. This makes it hard to understand and leverage. |
| **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** |
| Anonymous enum, use named enum instead. |
| **Conditionals** |
| 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 Exploratory Variables** |
| Use locals to give stats in algorithms names. |
| **Encapsulate Boundary Conditions** |
| Boundary conditions are hard to keep track of. Put the processing for them in one place, e.g. postexit = exit + 1; |
| **Preferred Dedicated Value Objects to Primitive Types** |
| Instead of passing 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 formed, not correct grammar/spelling. |
| **Obfuscated Intent** |
| Too dense algorithms that lose all expressiveness. |
| **Obvious Behavior 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 Deletetitem 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. |
| **Methods with Too Many Arguments** |
| Prefer fewer arguments. Maybe functionality can be outsourced to a dedicated class that holds the information in fields. |
| **Methods 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) |
| **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 unrestricted 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. |

| **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, simulations). |
| 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** |
| Refactor code 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 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. |
Clean
Incorrect Behaviour at Boundaries

Just working is not enough, make sure you understand why it works. Understand the Algorithm

Don't Assume

Understand the Algorithm

Just working is not enough, make sure you understand why it works. Understand the Algorithm

Make your tests mock the real production environment. Use fakes, stubs, and spies to make your tests as close to the real world as possible. This will help you to understand the algorithm better.

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 abstractions 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 + "Tests".

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 method or base 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

Names reflect the testee. E.g. FeatureWhenScenarioThenBehaviour.

Single Scenario per Test

One test checks one scenario only.

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 sut). Clearly identifies the SUT, robust against refactoring.

Naming Result Values

Give the variable holding the result of the tested method always the same name result.

Anonymous Variables

Always use the same name for variables holding uninteresting arguments to tested methods (e.g. anonymousTest).

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 ...)

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 re-use).

Mixing Stubbing and Expectation Declaration

Make sure that you use test methods (e.g. assert, expect) syntax when using mocks. Don't mix setup and setup up (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 mocks or mock setup, then consider splitting the testee into several classes or provide an additional abstraction between your testee and its dependencies.

Unit Test Principles

Fast

Unit tests have to be fast in order to be executed often. Fast means much smaller than seconds.

Isolated

Clear where the failure happened. 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!

Tinny

Tests are written at the right time (TDD, DOT, POUTing)

Unit Test Smells

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 noise makes it difficult to use 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 problematic 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 test – probably once used 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 clear by looking at the test method only – no initialisation or asserts somewhere else.

Boated Construction

The construction of dependencies and arguments used in calls to testee makes test surprisingly unreadable. 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.

Grease Test

Sometimes passes, sometimes fails due to left overs or environment.

TDD Process Smells

A 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.

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.

Test Domain Specific Language

Use test DSLs to simplify reading tests: helper methods, classes.

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 Extent 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 ("If You Make It")

Return a constant to get first test running. Refactor later.

Triangulate – Drive Abstraction

Write tests 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 it 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.

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, 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.
**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**
- Automatically build an installer as often as possible to test software on a test system (for manual tests, or tests with real hardware).

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

**ATDD/TDD Cheat Sheet**

**ATDD, TDD cycle**

**Pre-Commit Check**
- Write acceptance criteria for user story
- Define examples
- Write acceptance test skeleton

**Post-Commit Check**
- Define examples
- Write acceptance test skeleton
- Make an initial design
- Refactor
- Automatically build an installer as often as possible to test software on a test system (for manual tests, or tests with real hardware).

**Communicate Failed Integration to Whole Team**
- Refactor existing code to simplify introduction of new functionality. Run all tests to keep code working.

**Build Staging**
- Make initial or update class design
- Make error reason obvious
- Run acceptance test

**Automatically Build an Installer for Test System**
- Implement a Spike to get the acceptance test running so that you get an initial design.

**Continuous Deployment**
- Implement a Spike to gather enough knowledge so you can design a possible solution.

**Test Pyramid**
- Add arrange, act and assert parts to the acceptance test skeleton (Given, When, Then or Establish, Because, It...)
- Automate build an installer as often as possible to test software on a test system (for manual tests, or tests with real hardware).

**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

**Legend:**

<table>
<thead>
<tr>
<th>DO</th>
<th>DON'T</th>
</tr>
</thead>
</table>

---

**Bibliography**

**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

---

**Legend:**

<table>
<thead>
<tr>
<th>DO</th>
<th>DON'T</th>
</tr>
</thead>
</table>

---

**Clean ATDD/TDD Cheat Sheet**

Urs Enzler  May 2013  V2

---