The essence of "Clean Code"

A heavily paraphrased summary of the book

Robert C. Martin: Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall 2008, 431 pages

(Lutz Prechelt, 2013-2014)

Ch. 1: Clean Code

We will always develop on the code level because all the details matter.

Good, clean code matters: Bad code eventually brings a product down, because during further development, productivity gradually approaches zero.

Programmers must stand up for clean code just like managers stand up for requirements and schedules. But managers rely on programmers, not vice versa. And in order to go fast, we *must* have clean code.

Definitions of clean code by Bjarne Stroustrup (→C++), Grady Booch (→UML), Dave Thomas, Michael Feathers, Ron Jeffries (→XP), Ward Cunningham (→XP, Wiki, Design Patterns): Clean code is elegant, simple, efficient, straightforward, crisp, clear, literate, readable by others, unsurprising, has minimal and explicit dependencies, has automated tests, minimizes the number of classes and methods, expresses its design ideas, handles errors, has nothing obvious that one could do to make it better, looks like the author has cared.

Code gets read a lot (at least whenever someone is writing more code), so any school of clean code should emphasize readability. Cleaning up a little wherever you go is required to keep code clean.

Ch. 2: Meaningful Names

Use meaningful, intention-revealing, pronounceable names

Avoid disinformation (accidental similarities with something else entirely or too-subtle name differences) and puns

Larger scopes require longer names (for successful searching)

Class Names should be noun phrases, method names verb phrases

Use the same word for a concept consistently

Use problem domain names for solution domain concepts and technical terms for solution domain concepts.

Don't be afraid to globally change bad names (including their uses, of course).

Ch. 3: Functions

Functions (methods) should be small and do only one thing. All statements should be exactly one level of abstraction below the concept represented by the function, that is, should be worth mentioning in a summary of the implementation. This implies avoiding nested control structures, switch statements, and most if-else-if chains.

Order the functions thus broken down in depth-first order, so that the overall code can be read top-to-bottom. It is hard to overestimate the importance of descriptive and consistent names and of the absence of surprising side-effects.

Parameters make functions harder to grasp and often are on a lower level of abstraction, so avoid them as best you can, e.g. by introducing conceptual helper classes or turning arguments into member variables.

Avoid duplication.

All this describes a good end result. Initially, you may well have long, ill-named, complex, parameter-rich functions that do many things. This is no problem, as long as you then go and refactor, refactor, refactor.

Ch. 4: Comments

"The proper use of comments is to compensate for our failure to express ourself in code." Comments do not make up for bad code, rather, we should express ourselves in the code.

Types of good comments are: legal comments, informative comments, explanations of intent, warning of consequences, TODO, marking as important, documenting public API.

Types of bad comments are: unclear mumbling, redundant comments, misleading comments, mandated comments, changelog comments, a comment instead of putting code into a separate function, banners, closing-brace (etc.) comments, attributions

and by-lines, commented-out code, HTMLified comments, nonlocal information, too much or irrelevant information, comments needing explanation, documenting non-public API.

Ch. 5: Formatting

Adequate and uniform is required if you intend to communicate orderliness to your code's readers and to provide readability. Use a formatting tool.

Avoid too-long files; ~200 lines is fine.

Good files are like newspaper articles, with a heading, the important stuff first, and details later.

Use blank lines to separate separate stuff and no blank lines to group related stuff. Keep somewhat-related stuff nearby in the file; functions below their calls. Affinity should produce nearness.

Don't let lines get too long (80 or 120 is OK). Use horizontal whitespace to indicate relatedness and separateness (but aligning columns emphasizes the wrong things). Indent properly and line-break even short constructs.

Use team-wide formatting rules.

Ch. 6: Objects and Data Structures

Decide consciously what to hide in your objects. It depends on what changes are expected (and sometimes bare public data structures will be just fine).

Preferably, call only methods of your own class, of objects you have just created, of parameters, and of instance variables, not further methods reachable through these objects (Law of Demeter).

Ch. 7: Error Handling

Error handling is important and there is often a lot of it, but it must not obscure the main intentions of the code, so use exceptions (not return codes) and treat try-catch blocks like transactions.

Your exceptions should provide intent, context and error type detail. Classify exceptions by how they are caught and handled. Wrap third-party APIs to remap their exceptions as needed.

Null checks are cumbersome, just like return codes; use exceptions or do-nothing objects rather than returning or accepting null.

Ch. 8: Boundaries

Keep boundaries clean between code originating from different teams, e.g.

- do not widely pass around (in your code) over-flexible or change-prone objects of third-party libraries;
- learn, document, and change-control third-party libraries by writing learning tests for them.

Ch. 9: Unit Tests

Automated tests should cover every detail of our code's functionality, should accompany the code in the archive, and be easy to execute.

They should be built with Test-Driven-Development (TDD):

- You may not write production code until you have written a failing unit test.
- You may not write more of a unit test than is sufficient to fail (not-compiling is failing).
- You may not write more production code than is sufficient to pass the currently failing test.

This style produces a cycle of maybe 30 seconds in which we develop all our production code. The tests must be as clean as the code, as they will have to change, too, so always refactor both as needed. Each test should check a single concept.

F.I.R.S.T. rule: Tests should be fast, independent of each other, repeatable, self-validating, and timely (i.e. written just before the corresponding code).

Ch. 10: Classes

Ordering: Constants before variables before methods (within each: public before private, but private methods used only once follow right after their usage).

Keep variables and utility methods private unless that gets in the way of testing; then use protected or package.

Classes should be small: Have only one responsibility (Single Responsibility Principle (SRP): have only one reason to change). If a 25-word description of the class responsibilities uses the term "and", be wary.

Smaller classes do not increase the number of concepts relevant for understanding: Responsibilities. There will be more classes,

but their purpose will be clearer and the need to wade through irrelevant aspects of a class smaller.

Cohesion: A method that accesses more of the class's variables is more cohesive to the class. Overall-low cohesion (e.g. from promoting local variables to instance variables when extracting sub-methods) tends to be bad and may indicate the class should be split.

p.141-146: Example of splitting a long one-method class PrintPrimes into three classes; explains the responsibilities.

Splitting classes also tends to support the Open-Closed Principle (OCP) of avoiding to modify existing classes when extending the program's functionality.

Dependency Inversion Principle (DIP): Rather than hard-coding calls to dependent services, rely on an abstraction (interface) only and pass a concrete service (object) in as a parameter.

Ch. 11: Systems

Obey the Separation of Concerns principle. Never let convenient idioms lead to modularity breakdown, e.g. by hard-coding dependencies; the startup process is a major concern. Use factories and Dependency Injection (DI), which applies the Inversion of Control (IoC) principle: Delegate the creation of dependencies to objects that are specialized to that task (either explicitly or, preferably, via suitable constructor parameters or setter methods). This also supports the Single Responsibility Principle.

Proper Separation of Concerns will allow to grow even the architectural structure of a system. It is most difficult for Cross-Cutting Concerns. Those can sometimes be handled by Aspect-Oriented Programming (AOP). Spring is a pure-Java AOP framework that relies on nested decorators. AspectJ can help in those few cases where Spring is insufficient.

Full decoupling (with mostly technology-free POJOs) will allow architectural changes (e.g. exchanging persistence and communication technologies) easily. It also simplifies decentralizing or postponing decisions.

DSLs help to keep application logic concise, readable, modifiable, and technology-free.

Never forget to use the simplest thing that can possibly work.

Ch. 12: Emergence

Good designs can be produced by letting them emerge from the use of Kent Becks four rules of Simple Design at any time. A simple design

(1) runs all the tests, i.e. everything is being tested (realistic with SRP-style, DIP'ed classes only) and nothing fails,

(2) contains no duplication (realistic with stubborn refactoring only),

(3) expresses the intent of the programmer (realistic with small-scale, straightforward, unit-tested code with long, convention-obeying names plus stubborn refactoring only), and

(4) minimizes the number of classes and methods (requires not overdoing SRP, de-duplication, etc, and avoiding element-generating dogmas in general).

Ch. 13: Concurrency

Concurrency decouples what is done from when it is done and can improve or complicate the structure, understandability, and efficiency of a system. Program state evolution, however, becomes much more complicated. Correct concurrency is complex, even for simple problems.

Strictly obey the SRP: Keep concurrency management separate from other code.

Severely limit access to data. Prefer copies over sharing and message passing style over threads using copies or sharing.

Know your library: thread-safe vs. non-thread-safe, blocking vs. non-blocking, executor framework, synchronization helpers.

Know basic concepts: bound resources, mutual exclusion, starvation, deadlock, livelock.

Know basic programming models: producer-consumer, reader-writer, dining philosophers situations.

Keep locked sections small.

Graceful shutdown can be difficult (deadlocks).

Test with variations ("jiggling": #threads, speed, yielding, #processors, problem sizes, OSs, etc.) and track down each and every failure. Consider AOP-based instrumentation or tools for jiggling. Get non-threaded code working reliably first.

Ch. 14: Successive Refinements

Design style example *Args* (commandline-argument parsing).

Presents a well-designed solution (unfortunately without showing StringArrayArgumentMarshaler, the reason why Marshalers are set with Iterator arguments)

States that this cannot be written in one pass, but by drafting and then successively cleaning and improving.

Presents a messy earlier version of *Args* that has everything in one class (p.201+).

Presents a still earlier version that only handles Boolean args and is still reasonably tidy (p.206+) and a subsequent version that handles Boolean+String (p.208+) and gets visibly messy. Explains why he started refactoring after the Boolean+String+Integer version (p.212) and how he recognized the concept of ArgumentMarshaler class then [whose getValue method is static to allow for different result types: ugly!].

Explains many of the refactoring steps, made in TDD must-never-break mini-step style (p.213-242, based on a test suite shown only in one form on p.242-245). Good discussion.

"Nothing has a more profound and long-term degrading effect upon a development project than bad code. [...] the solution is to continuously keep your code as clean and simple as it can be. Never let the rot get started."

Ch. 15: JUnit Internals

Presents the test suite for the ComparisonCompactor class from JUnit, then the class itself, then critiques it.

Ch. 16: Refactoring SerialDate

Ch. 17: Smells and Heuristics

A: Concurrency II