The essence of "Clean Coder"

A heavily paraphrased summary of the book

Robert C. Martin: Clean Coder: A Code of Conduct for Professional Programmers, Prentice Hall 2011, 210 pages

(Lutz Prechelt, 2014)

Ch. 1: Professionalism

Being a professional means taking full responsibility for one's actions. "What would happen if you allowed a bug to slip through a module, and it cost your company $10,000? The nonprofessional would shrug his shoulders, say 'stuff happens', and start writing the next module. The professional would write the company a check for $10,000!" [real disaster story about field-failure of a routine each test of which took hours]

The first rule of professionalism for a software developer is not doing harm to the function nor the structure of the software. You will always make occasional mistakes, but you must learn from each. Promptly.

You should be *certain* about all code you release and firmly expect QA to find nothing wrong with it. Test it. Test it again. Automate your tests. Demand 100% coverage. Design your code to be easy to test.

You should follow the Boy Scout rule and always leave a module a little cleaner than you found it so that it becomes easier to change over time, not harder. Suitable automated tests can allow you to not be afraid to change the code and continually changing it makes sure it stays that way.

Your career is your responsibility, not your employer's. Spend 20 hours a week beyond your normal work to improve your knowledge and skills. Read, experiment, practice (kata), talk to others, collaborate, look over the fence, mentor. It should be fun.

Also, know your domain, identify with your customer (no "us vs. them", ever), be aware of the arrogance inherent in programming and learn to be humble, too.

Ch. 2: Saying No

[real disaster story about premature deployment of a totally immature distributed system]

"Professionals speak truth to power. Professionals have the courage to say no to their managers."

Managers and developers have roles that are often adversarial, because on the short term, their goals tend to conflict. The manager will defend her objectives, but will also expect you to defend yours for the best overall outcome (which is: reaching the largest goal that you and the manager share, which may be tricky to determine [fictious examples of how not and how to do it]).

The higher the stakes, the more valuable a "no" becomes – and the harder to say. [fictious example of how a tough instance may look]

Good teams will successfully work towards a yes – but only a right yes, that will later work out in practice. Saying "no" is often a prerequisite for getting to that right yes.

[real disaster story (from a blog post) about a developer who failed to say no]

Ch. 3: Saying Yes

There are three parts to making a commitment:

1. You *say* you'll do it.
2. You *mean* it.
3. You actually *do* it.

It is OK to clearly(!) stop before step 1 (see previous chapter), but a professional will not stop after step 1 or step 2. We need to learn to recognize it when others do (watch out for terms such as *need, should, hope, try, we, let's*).

Unconditional commitment always takes a form equivalent to *"I will achieve goal X until time Y"*. Commitment means taking full responsibility. Most results depend on conditions you cannot fully control, so you will often only commit to actions (not results) or commit only conditionally.

Your commitments must respect the limits of what you expect (based on your experience) you can and cannot do. If you recognize you will probably not be able to meet a commitment, you need to raise a red flag *immediately*.

[fictious examples of negotiation with and without proper commitment]

Ch. 4: Coding

Coding "requires a level of concentration and focus that few other disciplines require." A clean coder codes only if s/he can guarantee enough focus. Distractions (personal, environmental, or whatever) are a problem. Overtime is a problem.

[story where code written at 3 am created huge problems for a long time.]

Flow ("the Zone") is not as good as people think: You will be locally productive, but will often lose sight of the bigger picture and possibly produce not-so-good designs.

Music may be a distraction (even if you don't think so).

Interruptions are bad distractions. Pair programming is helpful to cope with them. TDD helps to make the pre-interruption context reproducible.

If you have writer's block, start pair programming.

Make sure you take in enough creative input, e.g. reading fiction books. Find out what works for you.

You have to find ways to minimize the time spent debugging. The only one I know is TDD.

Coding is a marathon, not a sprint, so conserve your energy and creativity. Go home when it's time, even in the middle of something important. Showers and cars are problem-solving resources, too!

Continuously re-estimate your best/likely/worst completion time and speak up as soon as you recognize you will likely be late. Do not allow anyone to rush you (see above). Consider overtime only for a short stretch (2 weeks max.) and only if there is a fallback plan as well. Use a proper definition of "done", with sufficiently high quality requirements.

Programming is too hard for anyone, so get help and provide help to others, in particular (but not only) in mentoring style. Don't protect your turf, don't shy away from asking, don't shove away others who ask.

Ch. 5: Test-driven Development

1. You are not allowed to write any production code until you have first written a failing unit test.
2. You are not allowed to write more of a unit test than is sufficient to fail—and not compiling is failing.
3. You are not allowed to write more production code than is sufficient to pass the currently failing unit test.

The cycle is only about 30 seconds long.

It

- provides certainty not having broken anything when making changes,
- reduces defect injection rates often 2-10x,
- provides courage for cleaning up messy code,
- documents how code is to be used, and
- makes you create designs with low coupling.

TDD is not a cure-all and is impractical or inappropriate in some (rare) cases.

Ch. 6: Practicing

Professionals practice: Musicians, football players, doctors, lawyers, soldiers. We should, too.

Our computers have become so powerful that we can have negligible turnaround times: We do things quickly (rather than sit down and think carefully about them) and that requires practice to make enough things unconscious, as in martial arts training.

What to do in a martial arts training room ("Dojo"):

Kata: A programming Kata is a precise set of choreographed keystrokes and mouse movements that simulates the solving of some programming problem. You aren’t actually solving the problem because you already know the solution. Rather, you are practicing the movements and decisions involved in solving the problem: IDE usage, TDD, CI. (See examples on http://katas.softwarecraftsmanship.org)

Wasa: A two-person Kata, done in ping-pong pair-programming style (TDD with rapid actor changes).

Randori: An N-person Wasa solving a new problem or a known Kata problem with new constraints.

Another approach to practicing is to broaden your experience, e.g. by participating in Open Source projects (akin to some pro-bono work of lawyers and physicians).

All professionals practice.

Ch. 7: Acceptance Testing

To avoid "garbage in, garbage out", make sure you understand the requirements – and expect your customer to initially *not* understand them. Creating this understanding means removing ambiguity.

The best way to do this is defining acceptance tests: Ask the customer for all conditions they will plausibly want the software behavior to fulfill and turn them into automated tests. (They often will not want to answer all your questions, so developers or testers will have to guess, in particular for the failure cases, and then validate the result with them.) Success of those tests constitutes the definition of 'Done'.

Code implementation should start only when test implementation is complete. Look out for silly, awkward, or plain incorrect tests and work with the test authors to improve them.

Unlike unit tests (which are for programmers only), the audience of acceptance tests are both business and developers. The prime purpose of both kinds is specification, testing is only secondary.

Test GUIs mostly one level below the actual GUI (on abstractions of the GUI elements) to reduce test volatility.

Run all tests in a continuous integration and *immediately* fix any failures that may occur.

Ch. 8: Testing Strategies

Develop such that the QA people find no problems. Consider them part of the team. They act as specifiers (writing acceptance tests, including the failure cases and corner cases) and perform exploratory testing.

Obey the testing pyramid:

Most tests are unit tests (by programmers, for programmers, in programming language, executing (almost) every statement of any class and asserting its behavior).

Many tests are component or integration tests (by QA or business assisted by programmers, for business and developers, in a component-testing framework, executing all relevant paths through larger combinations of classes). Component tests mock away other parts of the system and assert correct business rules. Integration tests may or may not mock and assert correct choreography of the pieces.

Some tests are automated system tests of the whole, usually at GUI level with the respective tools.

A bit more testing is done manually at system level in creative, exploratory fashion.

Ch. 9: Time Management

SW development, especially in management roles, requires good time management discipline.

Meetings are necessary, but are also often huge time wasters, so do not attend meetings that have no clear benefit (or leave underway) – this is a professional obligation. Meetings must have an agenda and a clear goal; agile stand-up meetings can be an efficient format. Iteration planning should take <5% of the iteration (2 hours for a one-week iteration).

Any argument that can’t be settled in five minutes can’t be settled by arguing, so don't try to; make measurements, flip a coin, or vote.

Concentration (focus) is a scarce resource; use it well when present and recharge with simpler tasks (e.g. meetings) and breaks in between. Sport helps. Creative input helps. The Pomodoro technique helps.

Professionals work on their real tasks, in a sensible priority order, even if they don't like some of them. They admit when they have chosen the wrong path and leave it quickly. They recognize messes (whether their own or others') and never accept them; they clean up. *Nothing* brings down productivity more than a mess.

Ch. 10: Estimation

Estimation is the source of most distrust between business people and developers, because the latter provide estimates which the former treat like commitments – and both are insufficiently aware that the estimate really is a probability distribution, not a fixed number.

The PERT technique computes and uses such distributions based on a best-case, nominal, and worst-case estimate for the project or, better, each task.

Wideband Delphi (e.g. "planning poker" and many other variants) is an estimation procedure where several estimators iteratively work towards agreement. Can be combined with PERT.

Ch. 11: Pressure

The professional developer is calm and decisive under pressure, adhering to his training and disciplines, knowing that they are the best way to meet the pressing deadlines and commitments. [Great story of Bob Martin getting it wrong and then seeing the light.]

Avoid situations that cause pressure, e.g. make only commitments you can fulfil, keep your code clean. Work in such a way that you need not change it when in crisis. Don't panic. Make a plan (and talk with your team). Don't rush. Trust your disciplines. Pair.

Offer pairing to others in crisis.

Ch. 12: Collaboration

Not all but most programmers like working alone. But we need to understand the goals of the people around us, including business folks. This requires communication. Likewise within the development team: Only collective code ownership and pairing produce a good level of communication. Programming is all about communication.

Ch. 13: Teams and Projects

Teams need time (months!) to gel: To really get to know each other and learn to truly work together. Thus, assigning fractional people to projects is a bad idea, as is breaking up a good team at the end of a project. Assigning several projects to one team can work well.

Ch. 14: Mentoring, Apprenticeship, and Craftsmanship

Young programmers (academic education or not) need mentoring. Mentoring can be implicit (e.g. by reading a good manual or observing someone working) or explicit.

Medicine has established a system of apprenticeship for new practitioners (a full year!) in which mentoring is likely to occur – and another 3 to 5 years of apprenticeship are required to become a professional in a medical specialty.

Given that we entrust software with all aspects of our lives, a reasonable period of training and supervised practice would be appropriate. A system of masters, journeymen and apprentices as in the crafts might be suitable. We currently do not impose on the elders a responsibility to teach the young. We are missing the mindset of craftsmanship and so the elders fail to consciously act as the role models that would make the young adopt the craftsmanship attitude as well.

App. A: Tools

Short, subjective, and insightful essays about various categories of tools. Also recommends specific tools.

- Version management: enterprise tools, CVS, SVN, git, branching
- editors/IDEs: vi, Emacs, IntelliJ, Eclipse, Textmate
- Issue tracking: Pivotal tracker, Lighthouse, Wiki, bulletin board, issue dumps
- Continuous build: Jenkins
- Unit testing tools: JUnit, RSpec, NUnit, Midje, CppUTest
- Component testing tools: FitNesse, RobotFX, Green Pepper, Cucumber, JBehave
- Integration testing tools: Selenium, Watir
- UML/MDA: code vs. detail as the main problem