The Clean Coder by Robert C. Martin: Book Notes

Behavioral Professionalism

- Professionalism
  - Do no harm
    - Don't lookout for yourself, lookout for the project and the customers
    - We must not create bugs
      - Impossible?
        - Human body is more complex and doctors take an oath to do no harm
      - QA Should Find Nothing!
      - All code should be covered with automated unit tests
        - 100% code coverage
          - Why write code and never test that it works
          - Professionally proves the system works
    - Delivering function at the expense of structure is a fool's errand
      - You must be able to make changes without exorbitant costs
      - Merciless refactoring
        - Make every class you touch slightly better
        - Always be improving
          - Difficulty to improve is bad design, fix it
  - Work Ethic
    - 168 Hours per week
      - 40 hours to solve your employers problems
      - 20 hours on your career
        - 3 hours per day
        - Lunch hour to read
        - Podcasts while traveling
      - 56 hours for sleep
      - 52 hours for everything else
    - Minimal list Software Professional should be conversant with
      - Design patterns
        - all 24 in the GOF book
      - Design principles
        - SOLID principles
      - Methods
        - XP, Scrum, Lean, Kanban, Waterfall, Structured Analysis and Structured Design
      - Disciplines
        - TDD, Object-Oriented design, Structured Programming, Continuous Integration and Pair Programming
      - Artificats
        - UML, DFDs, Structure Charts, Petri Nets, State Transition Diagrams and Tables, flow charts and decision tables
    - Continuous Learning
    - Practice
      - Kata
    - Mentoring
    - Know your domain
      - Read a couple books on the new topic
      - Spend some time with the experts and try to understand their principles and values
    - Identify with your employer
- Saying No
  - Professionals speak truth to power, and have the courage to say no to their managers
  - Being a Team Player
    - Telling the truth, committing to realistic estimates and not backing down is best for everyone
      - Don't allow anyone to lie to try to make the team sound better, its not beneficial to anyone, the project, the company or the customer
  - Trying
    - Admitting you can try harder implies you have not been trying hard
      - It applies you and your team to a new commitment that no one really thinks is plausible
        - This will inevitably lead to failure and put the project in a bad spot
          - Just say no
    - Don't accept you need to try to meet a date that is not agreed upon by the team
      - A professional is already trying, there is no magic pixie dust to make us work better on demand
      - Say no to trying
    - Response: I could try to levitate, but I promise you I won't
      - I could try to change lead in to gold
        - I could try to swim across the Atlantic
    - Don't stop by washing your hands and saying no
      - If your boss is not taking 'no' seriously, make it clear his/her boss needs to be told the truth, or you will tell them
        - This is for the good of the company, not yourself, not your boss, for everyone, this is your professional responsibility and right
- Saying Yes
- Time Management
  - Meetings
    - Truths
      - Necessary
      - Huge Time Wasters
      - ~$200/hr/attendee
    - Declining
      - need to use your time wisely
      - don't accept meetings unless it is a meeting for which your participation is immediately and significantly necessary to the job your doing now
        - You are the only one who can manager your time, your current project is highest priority, it is your responsibility to weigh which is more important
      - Your management should help keep you out of unnecessary meetings
    - Leaving
      - When the meeting gets boring, leave
        - politely exit that meeting
      - remaining in a meeting that has become a waste of time for you and you can no longer significantly contribute is unprofessional
    - Have an Agenda and a Goal
      - When invited, try to get the details of agenda and how much time is alloted for theam
        - if you can't get this information politely decline
      - When meeting gets side tracked ask the new topic be tabled and the agenda be followed
        - if this doesn't happen you should politely leave when possible
  - Arguments/Disagreements
    - Data is the best way to argue, not relentless arguing
      - Run experiments or do some simulation or modeling
    - Sometimes flipping a coin is best
      - If one path works out great, otherwise you can go down the other path
      - Pick a time frame in which to decide if path's need to change
    - Passive aggressive behavior is unprofessional
      - 'This is how he wants it so this is how he will get it'
    - If an argument must truly be settled, ask each of the arguers to present their case to the team in 5 minutes
      - have the team vote
        - whole meeting will take less than 15 min
  - Focus-Mana
    - Good nights sleep
    - Moderate caffeine
    - Recharging
      - long walk
      - conversation w/ friends
      - meditate
      - power nap
      - listen to podcast
    - Muscle focus
      - bike riding
      - carpentry
      - building models
      - gardening
  - Pomodoro Technique
    - 25 minute timer
      - politely decline all interruptions to get back to them within 25 min
  - Avoidance
    - Priority Inversion
      - Convincing yourself something else is more important to avoid doing the real work
      - Unprofessional - Don't do!
  - Blind Alleys
    - technical pathways that will lead no where
    - The Rule of Holes: When you are in one, stop digging
- Estimation
  - What Is an Estimate?
    - Business sees them as Commitment
    - Developers sees them as Guesses
      - Estimate
    - A Guess
      - no commitment is implied
        - no promise is made
        - Professional needs to be careful not to make any implied commitments with estimates
      - Missing an estimate is no dishonorable
      - An Estimate is a probability distribution
        - most likely will get done at our estimate forced
        - can get done sooner if things work out
        - can take much longer, twice, three times as long if things go poorly
  - PERT
    - Trivariate analysis
      - O - Optimistic Estimate
      - N - Nominal Estimate
      - P - Pessimistic Estimate
    - Expected duration
      - Mue= (O + 4N + P)/6
    - Standard deviation
      - (P - O)/6
    - Example
      - O - 1
      - N - 3
      - P - 12
      - Expected: 4.2 days
      - Standard Deviation: 1.8 days
    - Sequence Calculations
      - Sequence expected
        - Summation of each expected
      - Sequence deviation
        - sqrt(sum of squares of deviations)
  - Program Evaluation and Review Technique
  - Wideband Delphi
    - Team of people assemble, discuss a task, estimate the task, and iterate the discussion and estimation until they reach agreement
      - Flying Fingers
        - Fingers below table and show on 3, if agreement or close move on
      - Planning Poker
        - see scrum planning poker
      - Affinity Estimation
        - Cards with the tasks written on them are silently ordered by the team fromo smaller to larger left to right
        - Once most cards settle, discuss disagreements
        - Assign bucket points to the cards
      - Trivariate Estimates
        - Get three estimates, pessimistic, nominal and optimistic
        - Can use any of the three approaches still
  - Law of large numbers
    - break up tasks into smaller tasks and estimate them independently
- Pressure
  - Avoid Pressures
    - analogy
      - Don't want your surgeon to stop best practices while under 'dead'lines
    - Commitments
      - avoid committing to deadlines we can't meet
      - help the business meet goals set by others but don't take ownership of the commitment
        - If we can't find a way to meet the promises made by the business, then the people who made the promises must accept the responsibility
    - Stay Clean
      - "quick and dirty" is an oxymoron
        - Dirty always means slow
    - Crisis Discipline
      - How you behave in crisis shows what you truly believe in
        - If you don't do TDD in crisis then you don't truly believe it's helpful
        - If you make messes during crunch time, then you don't truly believe messes slow you down
      - Choose the methods you feel comfortable following in a crisis, then follow them all the time.
  - Handling Pressure
    - Don't Panic
      - Don't stay up all night, sit and frett, or rush, none of these things will help solve the problems
      - Slow down and think the problem through
    - Communicate
      - Let your team and superiors know you are in trouble
      - Discuss the best plans to get out of trouble, ask for their input
      - Avoid creating surprises for anyone
    - Rely on your disciplines
      - TDD more
      - Refactor harder
    - Get Help
      - Pair Program!
- Language of Commitment
  - Recognizing Lack of Commitment
    - "I will... by..."
      - I will finish this by Tuesday
  - How to Commit
    - Bad Excuses
      - It wouldn't work because I rely on person X to get this done
      - It wouldn't work because I don't really know if it can be done
      - It wouldn't work because sometimes I just won't make it
        - Sometimes this is life, but you need to raise a red flag as soon as possible to whoever you committed to
          - You can still commit to different parts, and you should make those apparent and clear
            - I can sit with Gary for an hour and understand the dependencies
            - I can create an interface that abstracts the module's dependency
            - I can meet three times this week w/ the build guy to make sure your changes work well in the build system
            - I can create an integration test for the module
          - You can still commit to actions. Instead to committing to fixing all 25 remaining bugs for a release:
            - I can go through all 25 bugs and try to recreate them
            - I can sit down with the QA who found each bug to see a repro of that bug
            - I can spend all the time I have this week trying to fix each bug
          - Going to be late for a meeting, as soon as you know, call your colleague and find a solution, maybe postpone or change locations
        - You can work to change the commitment in the best way possible
          - A task/bug is more difficult than anticipated, tell the team and come to a conclusion on how to move forward. Maybe fix specific pieces that you have discovered.
    - Don't allow vagueness
      - Never say or accept you will try to meet a goal
      - Don't hope for a goal you don't think is a certainty. Promise what you honestly need for a task..
        - This will force an awkward conversation to make a change. Maybe you reduce features to meet a desired deadline. Maybe the deadline gets moved back.
        - Potentially being responsibly creative to meet a deadline can be okay, just don't be the only one sacrificing
          - I will work 4 hours on Saturday and work with Willy Monday, but I will take Tuesday off because I know I will be useless without a break.
  - Making a Commitment
    - ❶ Say you'll do it
    - ❷ Mean it
    - ❸ Actually do it
  - Most communicated commitments are never real commitments
    - "Yeah, we really need to get some new routers"
    - Ask a team members to run tests -> "Sure. I hope to get to it by the end of the day"
      - But we know we will have to check with him the next day
    - Management to the team -> "We have to move faster"
      - Which means the team needs to move faster but management will not do anything toward that goal
  - Signs of non commitment
    - "Need/Should"
      - We need to get this done
      - I need to lose weight
      - Someone should make that happen
    - "Hope/Wish"
      - I hope to get this done by tomorrow
      - I hope we can meet again some day
      - I wish I had time for that
      - I wish this computer was faster
    - "Let's" (not followed by "I")
      - Let's meet sometime
      - Let's finish this thing

Development

- Coding Professionalism
  - Your code must work
    - Avoid the 'zone'
      - True the zone will allow you to write more code however that's not an indicator your more productive
      - The zone causes tunnel vision and causes you to lose sight of the big picture
        - This will cause you to revisit this code a lot which defeats the purpose of the increase in code
      - Pair programming makes it near impossible to enter the zone
      - Avoid listening to music and headphones - this doesn't usually help you code, it helps you stay in the zone which is undesired
      - The zone causes you to be rude to interruptions - extremely unprofessional to be rude
        - Pairing helps avoid or deal with interruptions to allow the other to be productive and help you get back into context after the interruption
        - TDD helps you stay in context as well, having a failing test to gain context, this will help with interruptions
    - Never write 3am code: It's crap even if it works, and that crap will be copied and pasted everywhere
    - Never write code when distracted with personal thoughts
      - Same analogy for driving, driving needs your focus, don't drive when distracted
      - Learn how to manage your time with those problems. Spend some blocks of time throughout the day working on both to ease your mind and allow you to code
    - Writers Block
      - Pair Program
      - Reduce debugging time to a minimum
        - Do this with TDD
        - Layers constantly retrying cases, surgeons re-opening up patients are unprofessional
          - constantly having to debug your code is unprofessional
    - No False Delivery
      - Worst unprofessionalism of all
  - Your code must solve the problem set for you by the customer
    - Not the requirements given, it is up to you to negotiate with the customer the true solution to their problem
  - Your code needs to follow solid engineering principles
  - Your code must be readable by other programmers (clean code)
- Help
  - your responsibility to be available to help each other
  - Help Others
    - unprofessional to sequester yourself in a cubicle and refuse the queries of others
      - Your work is not so important that you cannot lend someone of your time to help others
    - If needed - set office hours
    - Offer to help, don't wait to be asked
  - Accept Help
  - Mentor junior developers personally
  - New node
- Test Driven Development
  - It just works!
    - Developers don't have to defend TDD
    - Surgeons don't have to defend hand-washing
  - 3 Laws
    - You are not allowed to write any production code until you have first written a failing unit test
    - You are not allowed to write more of a unit test than is sufficient to fail -- and not compiling is failing.
    - You are not allowed to write more production code that is sufficient to pass the currently failing unit test
  - Benefits
    - Certainty
      - tens of unit tests written a day makes me certain the code works when I wrote and still works today
    - Defect Injection Rate
      - Proven to reduce the amount of bugs dramatically
    - Courage
      - Bad code is scary to fix without unit tests so often it doesn't get fixed
        - TDD helps give us courage and the confidant ability to refactor and fix code
      - When developers lose their fear of cleaning, they clean
    - Documentatin
      - Each unit test defines how the system should be used
      - Implicit, constantly updating, constantly added documentation
    - Unit test displays:
      - how to create every object
      - every way that object can be created
      - how to call every function in the system
      - every way that those functions can meaningfully be called
      - Anything you need to know how to do will be a unit test
    - Design
      - Following the laws forces you to be able to test your code
        - This forces you to consider and implement good design qualities
      - If tests are written after it's written defensively and will force the implementation to work
        - Tests written first are written offensively and will design the objects in the best way possible so it's easiest to use, forcing good design
  - TDD is NOT
    - Religion
    - Magic Formula
    - Does not guarantee the benefits and you can still write bad code with it
      - You can even write bad tests
    - Always be practical or appropriate
      - professionals should never follow a discipline when that discipline does more harm than good
- Kata
  - a precise set of choreographed movements that stimulates one side of a combat
  - Practicing over and over how to solve a particular problem
    - You already know the solution, but practicing the problem solving motions of TDD is the goal
      - Drive common problem/solution pairs into your subconcious
      - Attempt to practice so much to reach perfection
  - Examples
    - The Bowling Game
    - Integer Square Root
    - Prime Factors
    - Word Wrap
      - AWESOME Post
- Practicing
  - Wasa
    - Two-man kata
      - Ping-pong pair programming pattern
        - First person writes a failing test
        - Second person passes it
        - Second person writes a failing test
        - First person passes it
      - When you both really know the kata
        - Critique each other's keyboarding and mousing techniques.
    - Switch it up and add constraints in the tests
      - speed & memory
      - Make it competitive and fun
  - Randori
    - free-form combat
    - Wasa with more than 2 people
      - People rotate through while the code is projected on the wall
      - Can go in order, people can skip, move around, etc no specific rules just with more people
  - Broaden Your Experience
    - Open Source projects
    - Practice on your own time, its not your employers responsibility to keep you sharp
- Acceptance Testing
  - Premature Precesion
    - Business wants to know exactly what they are going to get, programmers want to know exactly what they are supposed to deliver
    - Uncertainty principle
      - requirements when implemented often aren't really what is desired in the end
        - observer effect
    - Estimation anxiety
    - Late Ambiguity
  - Cost
    - Testing ALWAYS gets cut first
    - Rerunning manual tests are very expensive
    - Having a product that the customer doesn't want is very expensive for everyone
    - Automated acceptance testing is cheap
      - Can maybe appear more expensive/work but it's the only way to prove it's working and continues to work
  - Automated
    - Always
      - why?
        - Ideally - stakeholders and QA
  - Who writes them?
    - realistically - business analysts and QA
      - analysts test happy paths
      - QA tests unhappy paths
    - If developers are forced to write them
      - Should not be the same developer who implemented it
  - When to write?
    - Should be written before the implementation begins
    - Mid-sprint should have all the acceptance tests written and failing
  - Negotiate
    - It is your professional responsibility to negotiate with whom has written the test and make a better test when necessary
    - NEVER be passive-aggressive
    - Example:
      - X has to run in under 2 seconds
      - Negotiate on average 2 seconds
      - "not what requirements say"
        - realistically though it's the only way
        - run test 15 times and average is less than 2 seconds
  - NOT unit tests
    - not redundant
      - test things very differently
    - Should never find any bugs
      - But probably still will
  - QA
    - Should be a part of the team
- Testing Strategies
  - Test Automation Pyramid (coverage %)
    - 5% Manual Exploratory
      - NOT automated, nor scripted
      - NOT written test plan of this kind of testing
      - Some teams will have a specialist do this, others will declare a day or two of 'bug hunting' to bang on the system
    - 10% System tests
      - Ultimate integration tests across the entire system
      - NOT test business rules
      - written by system architects and technical leads
    - 20% integration tests
      - choreography tests - how well the assembly of components dances together
      - NOT test business rules
      - Written by system architects or lead designers of the system
      - typically not executed as part of CI
        - Usually apart of nightly/weekly builds
    - 50% component tests
      - Acceptance tests of specific components
      - Written by QA and business with assistance from development
      - apart of continuous integration suite
    - 100% unit tests
      - written by programmers for programmers
      - as close to 100%, generally in the 90s of true coverage
- Collaboration
  - With Employers
    - First responsibility is to meet the needs of your employer
    - understand business goals
      - collaborate with managers, business analysts, testers and other team members to understand
      - understand the business and keep it afloat
  - With Programmers
    - owned code
      - Programmers should never own a piece of code
    - Collective Ownership
      - Break down all walls of code ownership and have the team own the code
    - Pairing
      - Many programmeres dislike the idea
        - odd because most pair in emergencies
      - clearly the most efficient way to solve the problem
        - two heads are better than one
      - Best way to solve problems
      - Professionals Pair
        - Best way to share knowledge with eachother
          - Professionals don't create knowledge silos
        - Best way to review code
  - Cerebellums
    - Need to be close to team members
      - need to hear frustrated mutters, serendipitous communication
    - Need to communicate as a unit
      - Some may work better alone for just themselves but that doesn't mean the team works better when you work alone
        - And its unlikely you work better when you work alone
          - Probably just work more comfortably

Management / Mentorship

- Teams & Projects
  - Does it Blend?
    - No such thing as a half of person
    - Gelled Team
      - takes time for a team to form
      - Magical about a gelled team
        - anticipate each other, cover for each other, support each other, and deman the best from each other
    - Which came first, Team or the Project?
      - Team should not be formed around availability to work on a project
      - Team should be first class decesion
  - Teams are harder to build than projects
- Mentoring
  - Manuals
    - Good books, manuals, api's are great mentors that can reach lots of people
  - Working with someone
    - learning how they work, their tricks, the best that they know of how to do things
  - Hard Knocks
    - learning everything on your own
      - most developers progress this way
      - very slow, very hard, not fun
  - Apprenticeship
    - Masters
      - 10+ years of experience
      - lead of multiple projects
      - coordinate multiple teams
      - Maintain technical role by reading, studying, practicing, doing and teaching
    - Journeymen
      - Trained, competent and energetic
      - Learn how to work well in teams and become team leaders
      - Knowledgeable about current technology but typically lack experience w/ many diverse systems
      - Teachers
        - teach apprentices design principles, design patterns, disciplines, and rituals
    - Apprentices/Interns
      - Graduates start here
      - Closely supervised by journeymen
      - Need intense pair programing
      - Ought to last a year
  - Craftsmanship
    - definition
      - someone who works quickly, but without rushing, who provides reasonoable estimates and meets commitments
      - knows when to say no, but tries hard to say yes
      - a professional
    - Convincing People
      - can't convince people to be crafstmen
      - can't convince them to accept craftmanship
      - Acceptance is not so much rational as much emotional
        - human thing
      - Make craftsmanship observable
        - act as a role model
- What Bob Martin Uses
  - Source Control
    - Open Source tools are usually your best option
    - Enterprise controls usually suck
      - use open source throughout work, then check into enterprise tool at the end of the iteration
    - 2008 switched to GIT
  - IDE
    - VI
      - old and vanishing as a coding tool
    - EMACS
      - Great general purpose tool, still not premier for editing code
    - Eclipse/IntelliJ
      - Bob uses IntelliJ
  - Issue Tracking
    - Start small, grow as needed
    - Bob is using Pivotal Tracker
  - Continuous Builds
    - Jenkins
    - If the build fails, stop the presses and resolve the issue
  - Unit Testing Tools
    - JUnit
    - RSPEC foro Ruby
    - NUnit for .Net
    - Midje for Clojure
    - CppUTest for C(++)
  - Component Testing Tools
    - FitNesse (written by Bob)
    - Others
      - RobotFX
      - Green Pepper
        - based on confluence?
      - Cucumber
      - JBehave
  - UML/MDA
    - Not worth it
    - Assumes the problem is code/design
      - wrong
        - problem is in the details
        - try to UML the details, and you might as well code the thing
    - No great diagramming language today that solves real problems
- Tooling