**Clean Code**

**CHAPTER 1**

**There Will Be Code**

• Code always sticks around because it shows what we need.

• Programming means telling machines what to do.

• Code is like a set of instructions.

• Any detailed instructions in a specific language count as code.

• Making a machine that perfectly understands us is impossible.

• Well-written instructions are as clear as code.

• Code is how we talk about what we want.

**Bad Code**

• Good instructions matter because bad ones slow us down.

• Struggling through messy instructions is frustrating.

• Bad instructions hold us back.

• Hurrying through instructions is a shared experience.

• Fixing rushed instructions later is almost never easy.

**The Total Cost of Owning a Mess**

• Starting fast might slow us down later.

• Making a mess makes us less productive over time.

• Pressure to work faster can lead to more messes.

**The Grand Redesign in the Sky**

• Starting over with a big project.

• Keeping instructions neat is vital for our careers.

**Attitude**

• We're partly to blame for making good instructions messy.

• We take some responsibility when things go wrong.

• Our job is to defend our instructions.

• It's not professional to listen to bosses who don't understand the risks of messes.

**The Primal Conundrum**

• Going fast means keeping instructions clean.

**The Art of Clean Code?**

• Writing neat instructions doesn't help if we don't know what it means.

• Making neat instructions needs lots of practice and a sense of cleanliness.

• Knowing if instructions are good or bad is key.

• Our sense of neatness helps us choose the best way to write instructions.

**What is Clean Code?**

* According to Bjarne Stroustrup, inventor of C++
  + Elegant and efficient
  + Simple logic
  + Instructions do one thing well.
  + Complete error handling
* According to Bjarne, neat instructions are nice to read.
* Messy instructions make more mess.
* Pragmatic Dave Thomas and Andy Hunt: "A building with broken windows looks like nobody cares about it."
* Messy instructions try to do too much.

• According to Grady Booch, author of Object-Oriented Analysis and Design with Applications:

* Simple and clear
* Like well-written stories
* Never hiding what the designer meant
* Full of clear ideas and simple lines
* Neat instructions show the problems we're solving.
* Instructions should be clear, not just guesses.

• According to "Bid" Dave Thomas, founder of OTI:

o Other people can read and improve them.

o They have tests.

o They have clear names.

o They don't depend on too many other things.

o They have clear rules to follow.

o They're easy to understand.

• Smaller is better.

• Neat instructions should be easy for people to read.

• According to Michael Feathers, author of Working Effectively with Legacy Code:

o Looks like they were written by someone who cares.

o There's nothing obvious to make better.

• According to Ron Jeffries, author of Extreme Programming Installed and Extreme Programming Adventures in C#

o They pass all the tests.

o They don't repeat themselves.

o They show all the ideas in the system.

o They don't have too many parts, like classes or methods.

o They focus on not repeating.

o They have clear names.

o They're clear and express ideas well.

o They split up into smaller parts when needed.

o When we want something from a group, we put it in a clearer way.

• According to Ward Cunningham, inventor of Wiki:

o Each step matches what we expect.

o They make the language fit the problem.

o They're easy, clear, and make sense.

**Schools of Thought**

We are Authors

• Being right in one way doesn't mean other ways are wrong.

• When we write instructions, we're like authors, and people will judge our work.

• We spend a lot more time reading than writing.

• Making instructions easy to read makes them easier to write.

**The Boy Scout Rule**

• Writing neat instructions isn't enough; we have to keep them neat over time.

• "Leave things cleaner than you found them."

• Cleaning up doesn't have to be hard.

**Prequel and Principles**

• Sometimes we talk about different ideas in design.

CHAPTER 2

**Use Intention-Revealing Names**

• Selecting appropriate names demands time but ultimately yields time savings.

• Exercise caution with names and consider replacing them with better alternatives when feasible.

• Naming should address fundamental inquiries: why, what, and how.

**Avoid Disinformation**

• Appropriately naming concepts significantly enhances code quality.

• Avoid introducing misleading clues that obscure code meaning.

• Avoid words with meanings that deviate from the intended purpose.

• Be wary of names that differ only slightly.

**Make Meaningful Distinctions**

• Writing code to satisfy compilers or interpreters often leads to complications.

• Number-series naming lacks informativeness.

• Noise words are unnecessary.

**Use Pronounceable Names**

• A name's pronouncability correlates with ease of discussion.

**Use Searchable Names**

• Single-letter names and numeric constants pose particular challenges in text searches.

• Single-letter names are suitable only for short methods.

• Name length should reflect scope size.

**Avoid Encodings**

• Use search-friendly names for variables or constants appearing in multiple places.

• Encoding type or scope information into names adds unnecessary complexity.

• Encoded names are often difficult to pronounce and prone to typos.

**Hungarian Notation**

• Once important, Hungarian Notation and similar type encodings now hinder comprehension.

• Previously useful when compilers didn't check types, now more likely to mislead readers.

**Member Prefixes**

• Modern environments make member prefixes unnecessary, especially with features highlighting or colorizing members.

**Interfaces and Implementations**

• An Abstract Factory, such as for creating shapes, will serve as an interface implemented by a concrete class.

**Avoid Mental Mapping**

• Names should be clear without requiring mental translation.

• Single-letter names are generally unsuitable.

• Professional programmers prioritize clarity over cleverness.

**Class Names**

• Classes and objects should bear noun or noun phrase names, not verbs.

**Method Names**

• Method names should comprise verbs or verb phrases.

• Accessors, mutators, and predicates should reflect their value and be prefixed accordingly.

• Use static factory methods when constructors are overloaded.

**Don’t Be Cute**

• Overly clever names are memorable only to those who share the author's humor.

• Prioritize clarity over entertainment value.

**Pick One Word per Concept**

• Consistently use one word for each abstract concept.

• Function names should be clear and independent.

• A consistent lexicon aids programmers in understanding code.

**Don’t Pun**

• Avoid using the same word for multiple purposes.

**Use Solution Domain Names**

• Remember that code readers are typically programmers.

• Opt for technical names where appropriate.

**Use Problem Domain Names**

• Programmers maintaining your code can consult domain experts for clarification.

• Distinguishing between solution and problem domain concepts is essential for effective programming and design.

**Add Meaningful Context**

• Enclose names in well-named classes, functions, or namespaces to provide context.

• Use prefixes to enhance context.

**Don’t Add Gratuitous Context**

• Prefer shorter names over longer ones.

• Avoid excessive context in names.