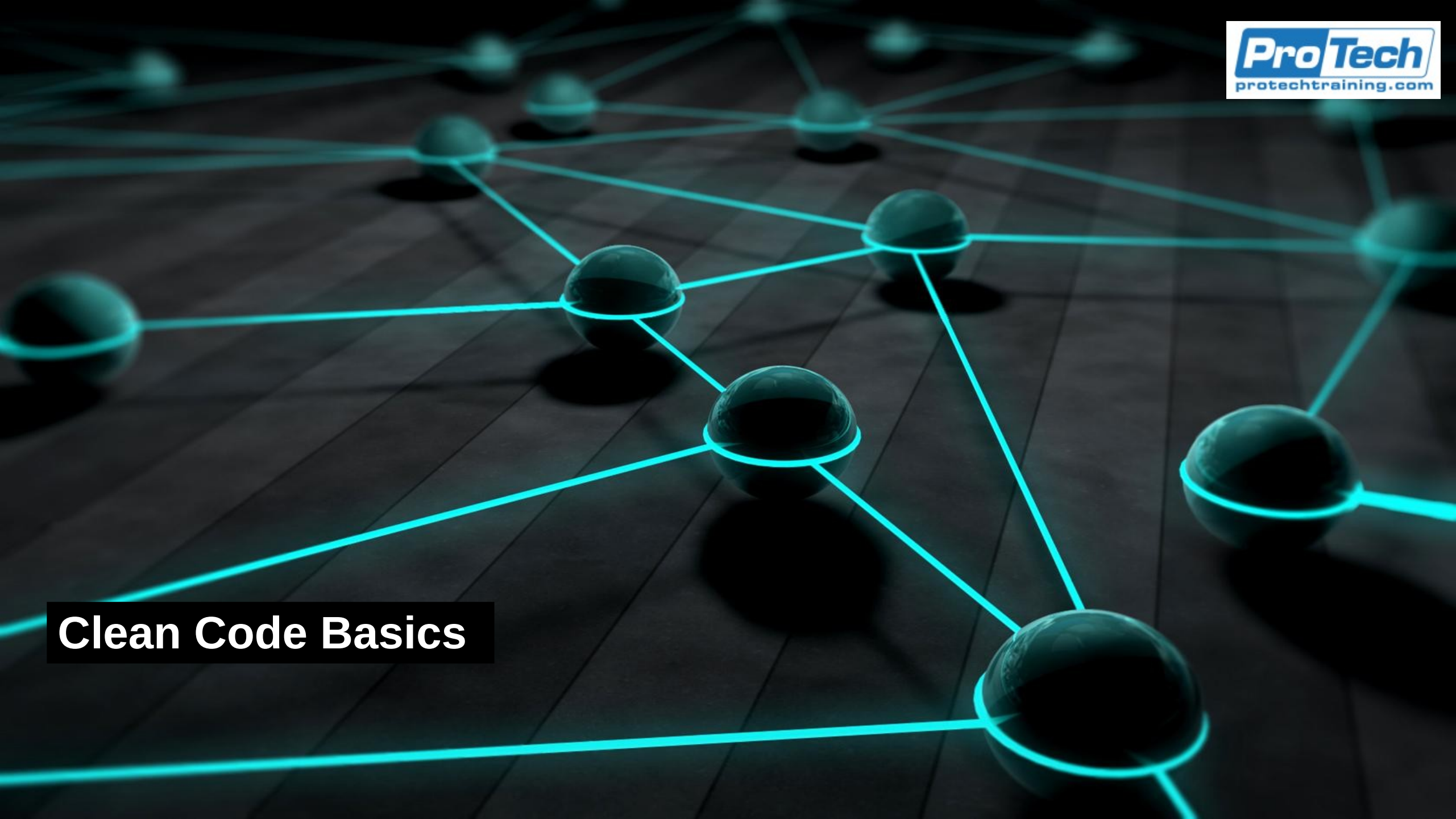


Clean Code Basics

Introduction

- Software Craftsmanship
  - A set of principles that guide the low actual writing of code.
  - Includes issues like readability and maintainability of the physical code
- Based on the principle that:
  - *“Good programmers write code that computers can understand, but great programmers write code that other programmers can understand.” (attributed to Kent Beck)*
  - Also expressed by Bjarne Stroustrup as *“Programming is a collaborative activity, forget that and you are lost*”
- This led to the idea of clean code – code that is written to be more easily understood by other programmers
  - Note that this is independent of program design or architecture
  - It applies to all code written in any language with some minor variations

Technical Debt

- Technical debt refers to the long-term costs incurred by taking shortcuts in software development.
  - Just like financial debt, you “borrow” time now by using a quick, easy, but sub-optimal solution instead of a cleaner or most maintainable one.
  - Over time, the “interest” payback accumulates
  - Messy code, poor design, lack of tests, or missing documentation make the system harder to understand, extend, modify, test and maintain.
- Often, future enhancements and support starts to stall when code is not understandable
  - Delayed by attempting to understand the code base
  - Or rewriting poorly structured code so that enhancement can be made.
  - Eventually, paying off that debt requires refactoring, rewriting, or adding missing work.

Types of Technical Debt

- Deliberate / Intentional Debt
  - Teams knowingly take shortcuts to meet deadlines or deliver quickly.
  - Like taking a loan: you accept the cost now, with a plan to pay it back later.
  - Acknowledges that skipped work will have to be done at a later date
  - Example: Skipping unit tests to release a hotfix quickly for an urgent security exploit
- Accidental / Unintentional Debt
  - Caused by a lack of knowledge, inexperience, or misunderstanding.
  - Usually, developers didn’t know there was a better approach at the time.
  - Example: Using an inefficient algorithm because the team was unaware of more effective one.

Types of Technical Debt

- Bit Rot (Entropy Debt)
  - Code structure naturally degrades over time as features are added without refactoring.
  - The design decays, making changes harder.
  - Example: A class that started small slowly turned into a 2,000-line “mega class” as a series of small changes over time add to the blob of code
- Outdated Design / Environmental Debt
  - Caused by changes in technology, business needs, or environment.
  - Code that was once well desgined no longer fits current requirements.
  - Example: Legacy system written before cloud-native architectures, now expensive to maintain.
- Unavoidable Debt
  - Sometimes debt is simply inevitable due to evolving requirements.
  - Early-stage projects often accumulate it because you can’t anticipate future needs.
  - Example: Startups building MVPs often accept debt to learn from user feedback.

Software Craftsmanship

- Intended to provide guidance to avoid technical debt
  - All code will eventually have to be modified
  - Write the code in such a way that future programmers can clearly understand what the code does
  - Organize the code in such a way that future changes will require less effort
  - Structure the code so that future changes can be localized in a module rather making a series of changes across multiple modules (referred to as “shotgun surgery”)
- The rationale is to avoid
  - Badly written code that produces unintentional debt by making code hard to understand
  - Badly written code that is harder to update and is prone to bit rot
  - Badly written code that is difficult to refactor when trying to accommodate design changes and the requirements of evolving systems

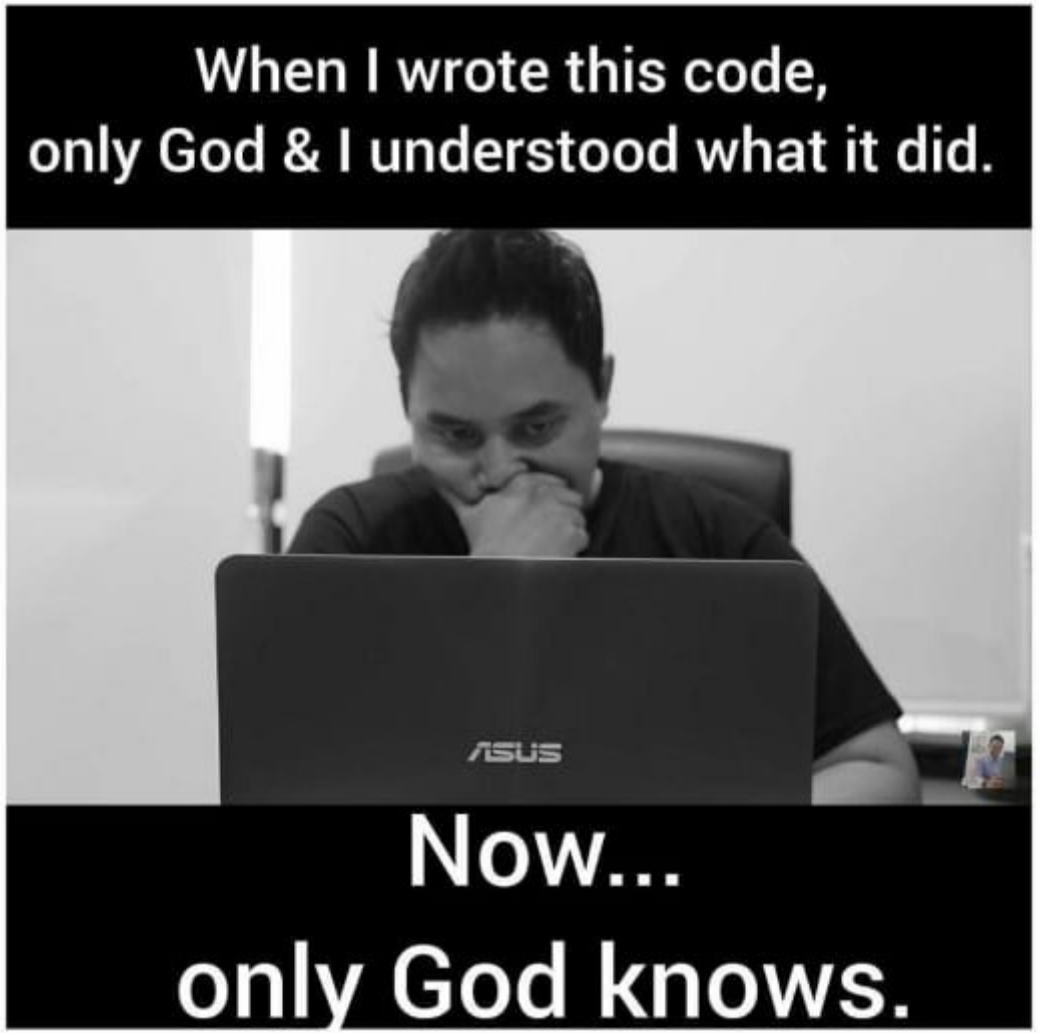
Code Amnesia

Refers to forgetting why code was written the way it was

- Hard to read an opaque
- Poor documentation (“This is obvious” or “I’ll remember why I did this”)
- Can be the result of doing something complicated that is “cool”

Old programming axioms

- Code you wrote more than a year ago may as well have been written by someone else.
- Write your code like the person who will be maintaining it is a homicidal maniac who knows where you live.

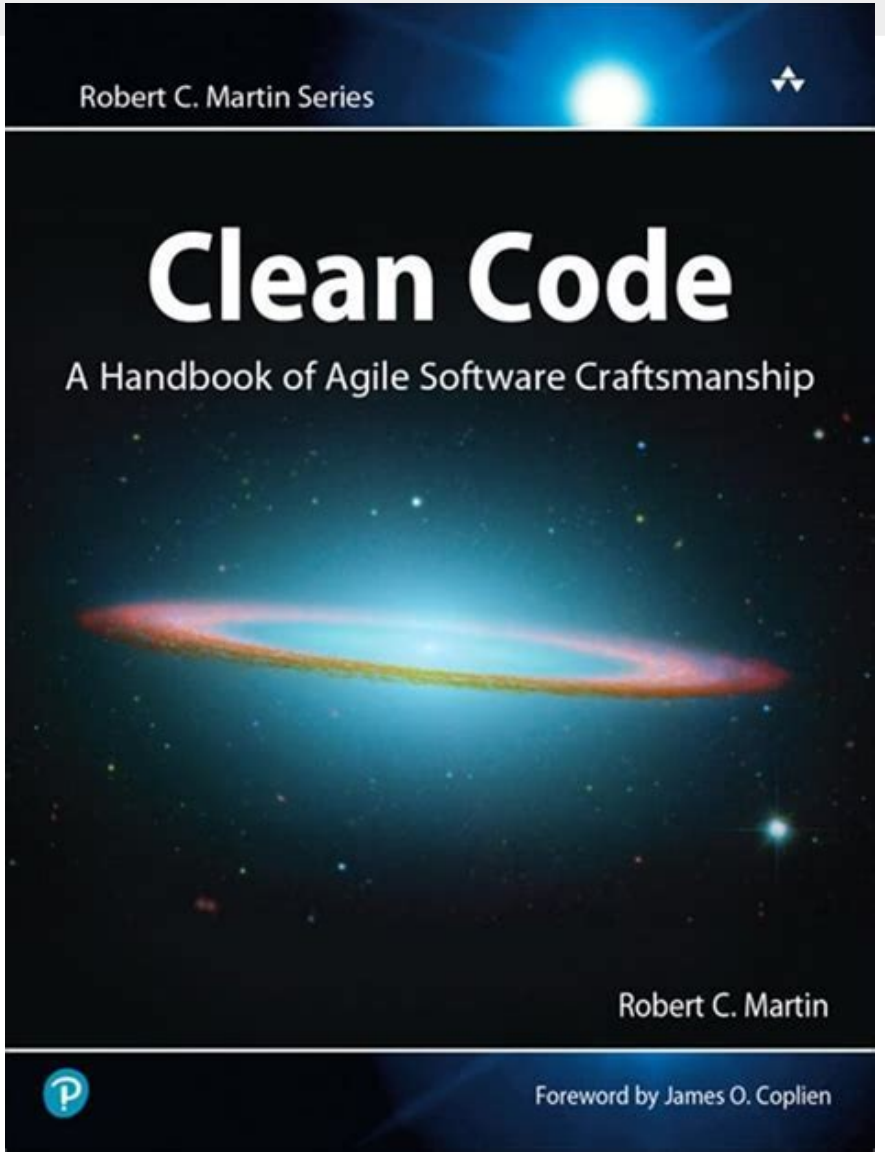


Clean Code

Software Craftsmanship is a movement to set guidelines for writing clean or understandable code

- Came out of the Agile movement
- Agile focused on process, not coding technique
- Software craftsmanship evolved to fill the gap
- A set of guidelines to implement craftsmanship in the process of writing and designing code
- Not intended to replace the sort of engineering design principles covered earlier, but to supplement them

Standard reference is Robert Martin's book.



Defining Well Crafted

According to Robert Martin (Uncle Bob)

- Clean code is not written by following a set of rules.
- You don’t become a software crafts man by learning a list of heuristics.
- Professionalism and craftsmanship come from values that drive disciplines.
- The idea was to use the analogy of a craftsman in other areas to writing code
- Influenced by the it is possible to see evidence of style and technique in code written by expert programmers
- Craftsmanship tried to quantify what gave that perception

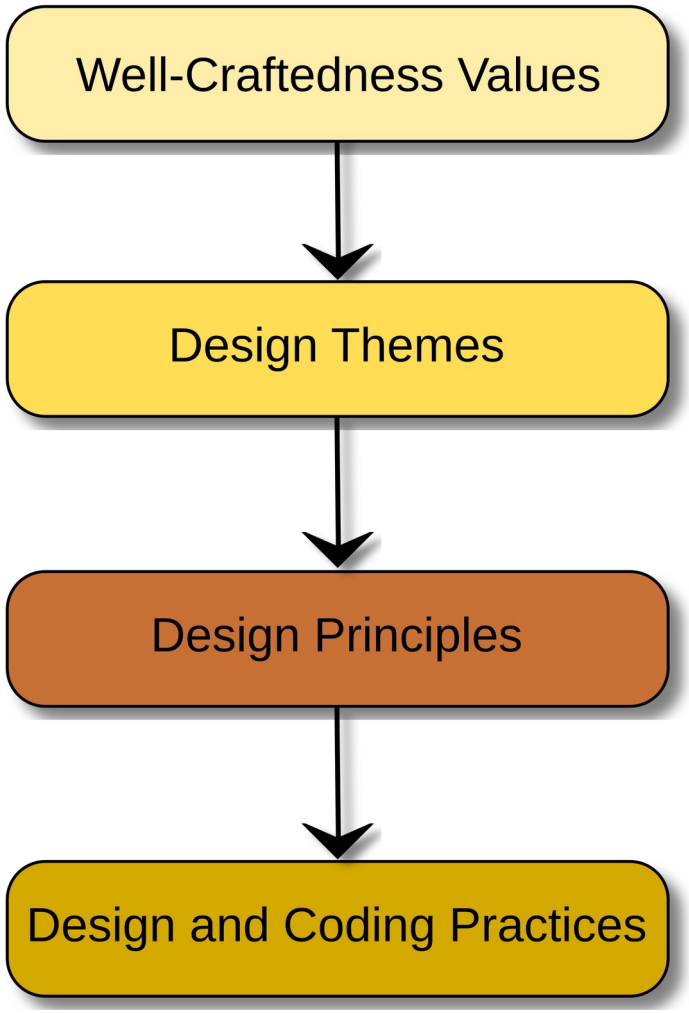
Craftsmanship Themes

Well-crafted software exhibits properties or values that are consistent with other well-crafted products.

General design themes are re-usable patterns used to build software that displays well-crafted values.

Design principles are axioms or rules of thumb that software developers follow as a way of creating software consistent with design themes.

Practices are tangible rules and specific techniques that developers use to implement a design principle,usually at the level of writing code.



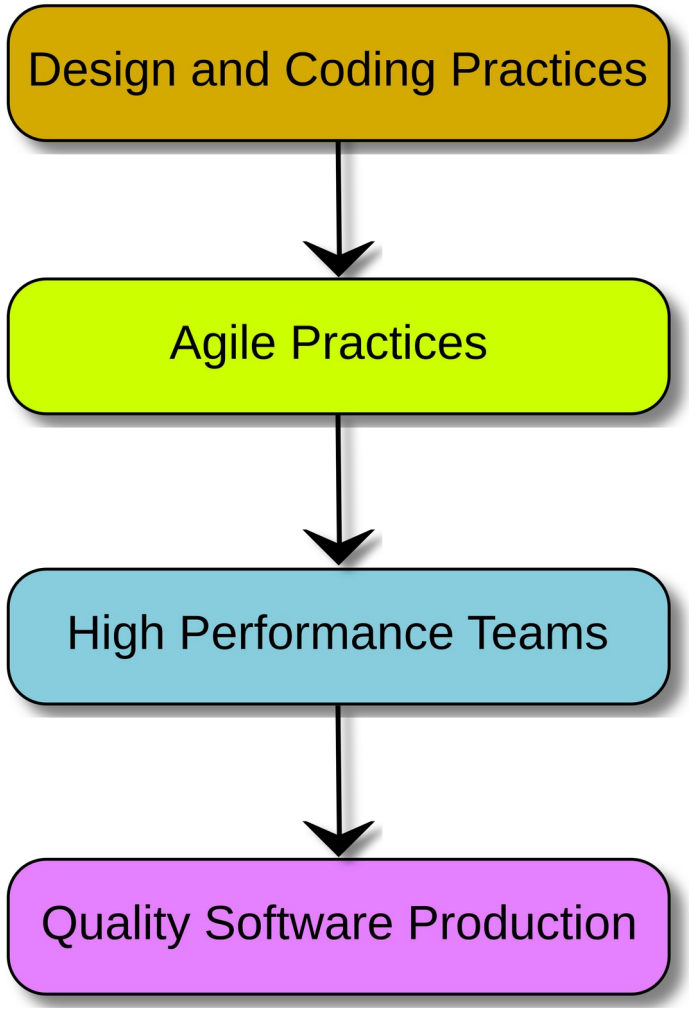
Craftsmanship Themes

Specific Agile practices support, promote and enhance craftsmanship.

Practices like Acceptance Test Driven Development, Test Driven Development, Domain Driven Design, shared code stewardship etc.

Agile allows for integrated teams of craftsmen from different specializations to work together smoothly – programmers, data designers, testers, etc.

This means consistently producing high quality software AND having a high quality production process for software development (i.e. efficient, effective and economical)



The Values

- Generally, the values of being well crafted apply across a wide range of disciplines
  - Notions like elegance, easy to use, etc.
- The next slide shows a summary of the types of values that have been identified as important in software by top software engineer
  - This is not a definitive list but are representative of experts on the subject
  - The values provide a context for the clean code sections that follow

Well Crafted: The Values

Correct:

- Always does the right thing.

Robust:

- Hard to break, handles bad data and environments gracefully

Maintainable:

- Easy to change, modify and upgrade

Usable:

- Easy to understand and use, has an intuitive feel to it.

Skillfully Made:

- Clearly made by someone who knew what they were doing.

Specialized Techniques:

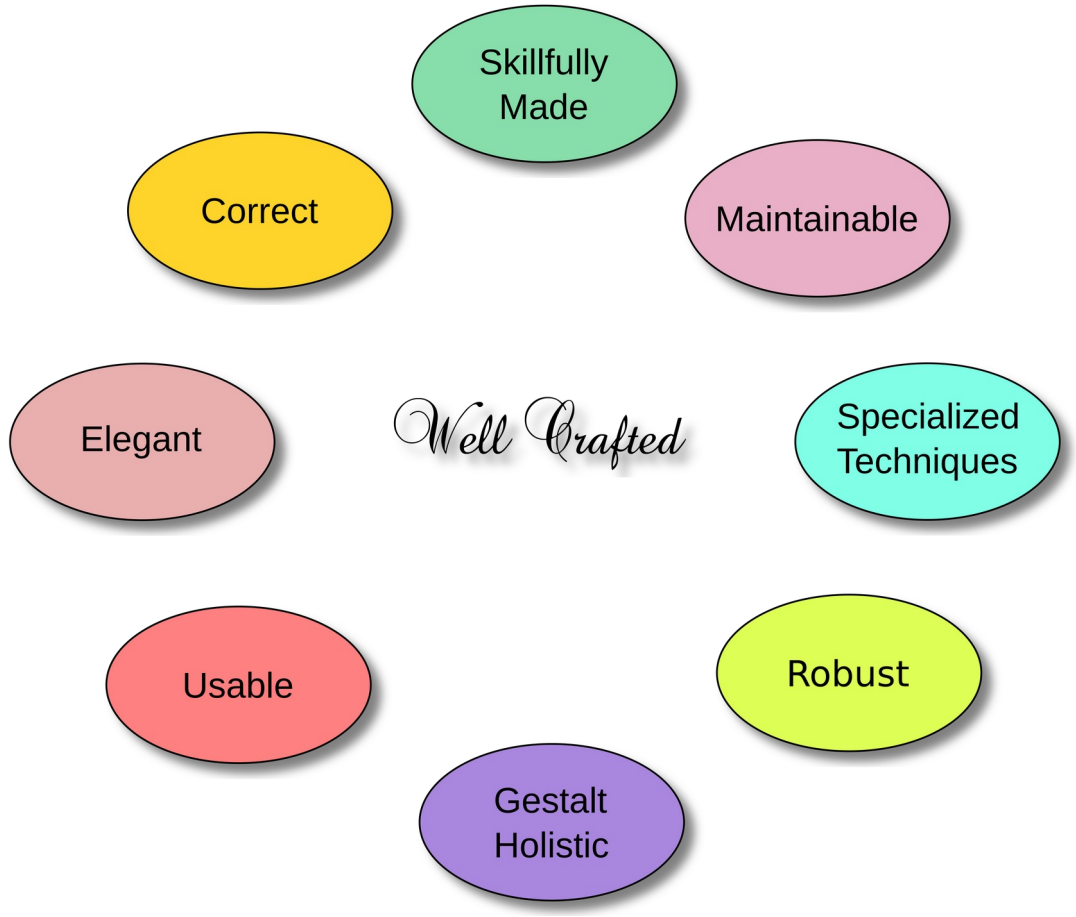
- Construction follows optimal best practices

Elegant:

- It is simple and beautiful

Holistic/Gestalt:

- Everything hangs together in a unified way – the whole is greater than the sum of the parts.



Overview of the Practices

- Clean code provides a set of practices that help implement the values
  - The book was written when most of the programming being done was Java
  - There is a bias to OO programming
    - For example, it talks about naming classes and using exceptions rather then errors
    - Some of this is paradigm specific
- However, most of the recommendations are still quite valid because
  - The emphasis is on readability and organizing code according to the engineering design principles defined in a previous section
    - *Code is read 10x more often than it is written so there should be a bias to making code readable*
  - Messy code slows teams and increases costs no matter what the programming paradigm is or architectural design or development process type chosen
  - Technical debt is universal: quick fixes now, higher costs later applies everywhere

Overview of the Guidelines

- Meaningful Names
  - Intention-revealing and searchable
  - Avoid noise words and abbreviations
  - This is often expressed as self documenting code – the names of methods, classes and variables describe their function.
  - Example: getActiveAccount() vs gAAct()
- Functions
  - Small, do one thing and are highly cohesive: restatement of a design principles
  - Descriptive names and layout that reveal the flow of logic in a method
  - Avoid side effects: promotes loose coupling and suppleness
  - All data should be passed through the interface: promotes modularization

Overview of the Guidelines

- Comments
  - Good comments explain WHY, not WHAT
  - If the code is self documenting, comments should not have to explain what the code is doing
  - Comments should provide context, like why a particular code structure was chosen
  - Comments contain background that will help maintainers understand the choices made by the original developer
  - Don’t try to comment your way out of bad code, rewrite it instead.
  - Use comments sparingly.
- Formatting
  - Consistency > personal preference
    - Create and use a coding standard that the whole team follows rigorously
  - Vertical spacing allows the easier understanding of the separation of parts of the code
  - Horizontal alignment aids clarity, it helps delineate hierarchical structure

Overview of the Guidelines

- Objects vs Data Structures
  - Objects hide implementation while data structures expose data
    - This is a very OO style guide and doesn't apply to modern languages like GO that are data driven
    - But it can be restated as using an interface to access data (dependency inversion principle) via a data service
    - Which allows the implementation of the service to vary without to modify the code that uses the data
  - Law of Demeter: avoid chaining method calls
    - This law states that modules should only interact with their immediate friends and not with strangers.
    - This approach minimizes dependencies between modules, making the software easier to maintain and adapt.
    - If this can't be done, it suggests that the code functionality is scattered and is lacking cohesion
- Error Handling
  - Use exceptions, not return codes
    - This is again very OO since many modem languages like GO and Rust do not use exceptions
    - For example, in Rust, functions often return a Result construct that contains either the return value or an error object
  - Handle errors gracefully
    - If the error is unrecoverable, ensure there is a graceful shutdown process

Overview of the Guidelines

- Boundaries
  - Encapsulate third-party libraries
    - Don’t call them directly, but create adapters/wrappers which are interfaces to the library
    - If the third party library changes, the wrapper is updated, not the codebase.
  - Protects our codebase from external churn when external dependencies are modified
- Unit Tests (we deal with this in week 4)
  - Unit tests should be validated by QA
  - They should be valid, accurate, reliable and comprehensive
  - The unit testing process should be automated
  - F.I.R.S.T principles for tests: Fast, Independent, Repeatable, Self-Validating, Timely

Overview of the Guidelines

- Classes
  - Small and cohesive, but also applies to whatever the basic program units are
  - Single Responsibility Principle: one reason to change
  - This still applies to other types of programming where we replace the idea of class with a function in a functional programming language or a subroutine in a structured language
- Systems
  - Separate construction from use
    - Construction = the process of creating and wiring up objects (e.g., instantiating classes, setting dependencies, configuring resources).
    - Use = the process of actually running business logic (e.g., calling methods, executing workflows).
    - If construction and use are mixed together in code, it creates tight coupling:
    - The code that performs business logic is also responsible for knowing how to construct all its dependencies.
    - That makes it harder to change, test, or reuse because the logic and setup are entangled.

Overview of the Guidelines

- Separate construction from use (cont)
  - Dependency injection reduces coupling
  - Architecture should support flexibility
    - The different subsystems should be loosely coupled, highly cohesive modules that only interact through interfaces
    - Allows a simple and quick way to replace or reorganize modules in a modification of the architecture
- Emergence: Kent Beck's 4 rules of simple design
  - Run tests while writing code to ensure it is correct
  - No duplication of code or functionality; single responsibility
  - Expresses intent
    - The code clearly communicates what it is doing to another developer (or to your future self).
    - Someone reading it should quickly understand the purpose, not just the mechanics.
    - The why behind the code should be obvious, not hidden in obscure names or tangled logic

Overview of the Guidelines

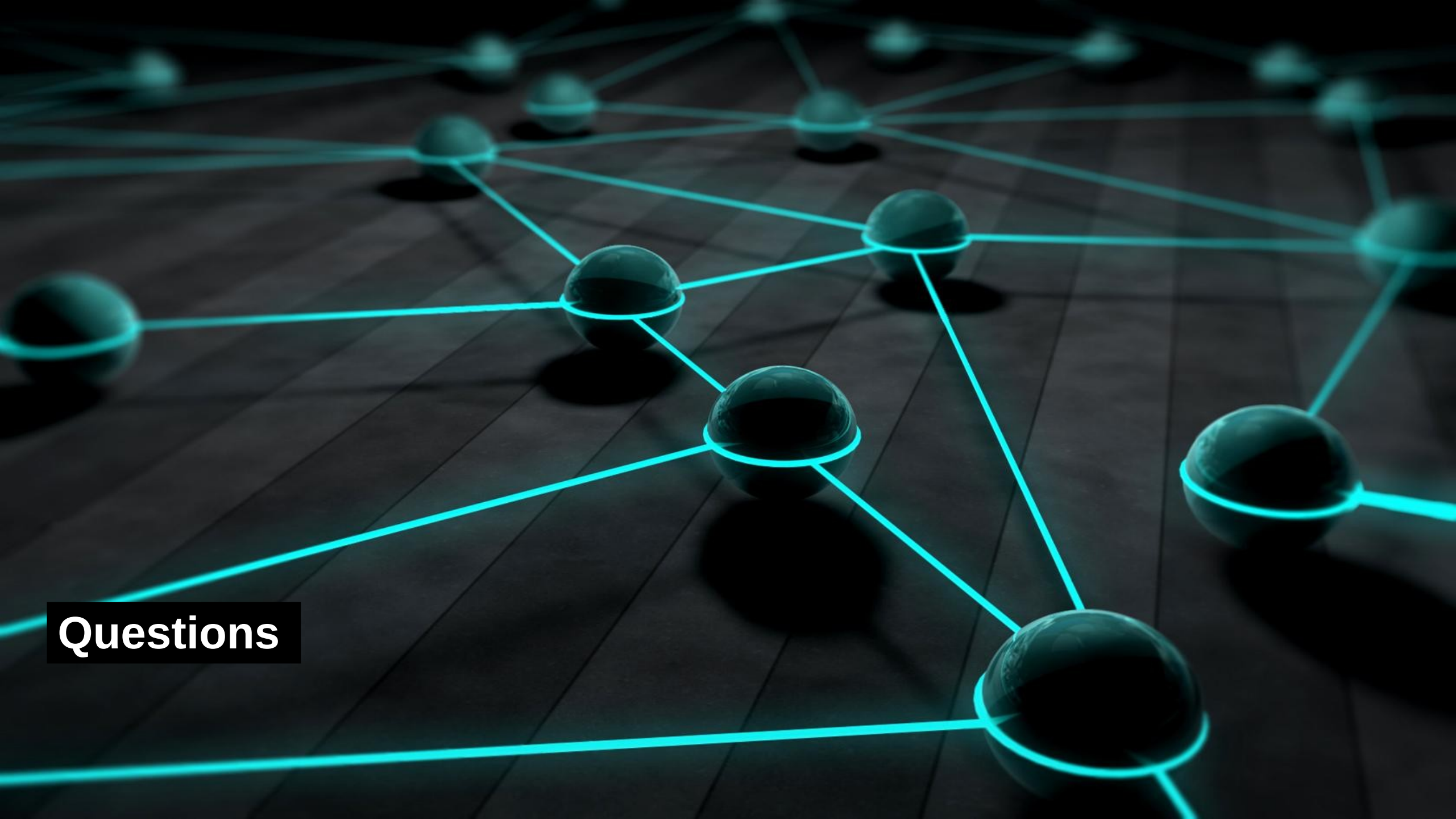
- Minimal entities
  - *Don’t create extra classes, methods, or abstractions unless they are truly needed.*
  - *Each class/method should have a clear purpose.*
  - *Avoid “just in case” design or over-engineering.*
  - *Keep it as simple as possible, but no simpler.*

- Code Smells & Heuristics
  - Code smells are indicators of badly structured working code
    - *Violates the engineering design principles or craftsmanship values*
    - *Examples: long functions, large classes, odd naming*
  - There are standard checklists that can be used to refactor
    - *Most have heuristics, which are recommended changes to eliminate code smells*
  - The idea of smells occurs in other areas
    - *Like architecture smells, design smells, data model smells*
    - *All indicate there are structural deficiencies*

Overview of the Guidelines

- “Smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality" (Kent Beck)
  - *Code smells are usually not bugs because they are not technically incorrect and do not prevent the program from functioning.*
  - *Instead, they indicate weaknesses in design that may slow down development or increase the risk of bugs or failures in the future. Bad code smells can be an indicator of factors that contribute to technical debt. (Kent Beck)*

- Basic Practices
  - Regular code reviews
  - Continuous refactoring to improve code quality once it passes the tests
  - Every code addition is an opportunity to improve the cleanliness of the codebase.


Questions