Chapter 15: Implementation

- Good Design
  - Half the implementation effort
  - Rigor
    - Ensures all requirements are addressed
  - Separation of concerns
    - Modularity
      - Allows work in isolation because components are independent form each other
    - Abstraction
      - Allows work in isolation because interfaces guarantee that components will work together
  - Anticipation of change
    - Allows changes to be absorbed seamlessly
  - Generality
    - Allows components to be reused throughout the system
  - Incrementality
    - Allows software to be developed with intermediate working results
- Bad Design
  - Lack of rigor leads to missing functionality
  - Lack of modularity leads to conflicts among developers
  - Lack of abstraction leads to massive integration problems (and headaches)
  - Lack of anticipation of changes leads to re-design and re-implementation
  - lack of generality leads to "code bloat": long, slow, duplicated work
- From Design to Code
  - Implementation is the process of translating the detailed design into code executable by a machine
  - Design/Implementation interaction
    - Sometimes changes have to be made
- Implementation Workflow
  - Aim
    - Implement the product
  - A large product is partitioned into subsystems
    - Implemented in parallel by coding in teams

- Subsystems consist of components or code artifacts
- Once the programmer has implemented an artifact, he or she unit tests it
- Then the module is passed
- Choose a **suitable** implementation language
    - The language is usually specified in the contract
    - But what if the contract specifies that the product is to be implemented in the "**most suitable**" language?
        - Language Generations
            - 1st Generation: Machine Language
            - 2nd Generation: Assembly Language
                - Machine specific
            - 3rd Generation: Functional, procedural, object-oriented
                - High-level language compiled to machine code
                - Java, C++, C, COBOL, Fortran
            - 4th Generation
                - Databases, Visual Basic, Forms
                - Domain Specific Languages
                    - Described using domain concepts, no code
                    - Code is automatically generated from models
        - Notable Languages
            - FORTRAN
            - Lisp
            - BASIC
            - Pascal
            - Ada
                - Used primarily by the Department of Defense
            - Smalltalk
                - First fully object oriented language
            - Prolog
            - C
            - C++
            - Java
                - Runs on the Java Virtual Machine
            - C#
        - Why so many Languages?
            - Evolution

        - What constitutes a good or a bad programming construct
        - Early 70s: structured programming in which goto-based control flow was replaced by high-level constructs such as while loops and case statements
        - Late 80s: nested block structure gave way to object-oriented structures

    - Special Purposes
        - Many languages were designed for a specific problem domain. For Example:
            - Scientific Applications
            - Business Applications
            - Artificial Intelligence
            - Systems Programming
            - Internet Programming

    - Personal Preference
        - The strength and variety of personal preference makes it unlikely that anyone will ever develop a universally accepted programming language

- Establish code conventions
    - Good Programming Practices
        - Use of *consistent* and *meaningful* variable names
            - Meaningful to future maintenance programmers
            - Consistent to aid future maintenance programmers

        - A code artifact includes the variable names
        - Use the same abbreviation if you are going abbreviates names

    - Naming
        - Avoid confusing characters
            - 1, l, o, 0, O, S, G, 6
            - Avoid misleading names
            - Avoid names with similar meaning
            - Use capitalization wisely and consistently

    - Code Layout
        - White space/ blank lines
        - Grouping
        - Alignment
        - Indentation
            - Tabs vs. Spaces

        - Parentheses
            - Ex. Parentheses around compound logical operations can increase clarity

        - Comments
            - Prologue Comment
                - At the beginning of every code artifact (class or method) every variable name must be explained in the comment prologue
                - Other programmers and maintenance programmer will quickly understand what each variable represents

            - Other Comments
                - Inline comments should be inserted to assist maintenance programmers in understanding what that code does
                - Comments are essential whenever the code is written in a non-obvious way, or makes use of some subtle aspect of the language
                - If code is too confusing, recode it in a clearer way
                    - Never promote/excuse poor programming

        - Programming Standards
            - Standards can be both a blessing and a curse
            - Modules of coincidental cohesion arise from rules like
                - "Every module will consist of between 35 and 50 executable statements"

            - Better
                - "Programmers should consult their managers before constructing a module with fewer than 35 or more than 50 executable statements"

            - No standard can be universally acceptable
            - Standards imposed from above will be ignored
            - Ideally, standard must be checkable by machine
            - The aim of standards is to make maintenance easier

    - Divide work effort
        - Real-life products are generally too large to be implemented by a single programmer
        - Assign different modules to different developers
            - Assignments can be incremental
            - Assignments change
                - ex. Illness of programmers

            - New employee
            - Employees who quit
            - Schedule adjustments
            - Star Programmers

        - Interfaces are tremendously important "contracts" among modules

    - Implement

- Code
  - The approach up to now:
    - Implementation followed by integration
    - This is a poor approach
  - Better:
    - Combine implementation and integration methodically
  - Integrated Development Environments
    - Developers use different tools during the software life span
      - Online interface checker, builder, text editor
    - Tools can be combined into a workbench that supports 1 or 2 activities
      - Configuration control, coding
    - An environment provides computer-aided support for most, if not all, the software process
    - IDEs integrate workbenches and tools in a common, uniform user interface
      - Same look and feel
      - Tools communicate via the same data format
    - Ideally, an IDE should be a process integration, an environment to support a specific software process
    - Features of an IDE
      - Source code editor
      - Compiler, interpreter
      - Build automation
      - Debugger
        - Methodical process of finding and reducing the number of defects in a program at run-time to make it behave as expected
        - A debugger simulates the execution of the code to be examined by running the code and be able to halt the execution when specific conditions are encountered
        - Some debugger features
          - Stepping
            - Step-by-step execution to animate program
          - Breaking
            - Pausing to examine current state
        - Methods
          - Print Statements
            - Output state information or track control flow at run-time

- Assertions Check
  - A specific condition at run-time and abort program if fails
- Exceptions
  - Detect logical errors or corner cases
  - Throw/raise exception when error occurs
  - Catch/Handle exception to correct error
  - Post-handling: continue execution, abort execution, or propagate (recursively) exception to encapsulation module
- Tracing
  - Stack trace tracks the history of execution of the program
- Interactive debugger

- Debugging Effort
  - Time required to diagnose the symptom and determine the cause takes much more time than:
  - Time Required to correct the error and conduct regression test
- Consequence of Bugs
  - Categories
    - Bugs
- Symptoms and Causes
  - Symptom and cause may be separated
  - Symptom may disappear whenever another problem is fixed
  - Cause may be due to a combination of non-errors
  - Cause may be due to system or compiler error
  - Cause may be due to assumptions that everyone believes
  - Symptom may be intermittent
- Final Thoughts
  - Think about the symptom before attempting to fix
  - Use tools
  - Get Help
  - Conduct regression tests when you do "fix" the bug

- Tests

- Integrate