Software Engineering Principles

Miaoqing Huang
University of Arkansas
Spring 2010
Outline

1. Importance of Principles

2. Key Principles

3. Case Study
   - Compiler Construction
1 Importance of Principles

2 Key Principles

3 Case Study
   - Compiler Construction
Principles primarily deal with the process of software engineering, i.e., the development of software
- The right process will help produce the desired software product
- Principles will guide the adoption of methodologies and the tools
  - Methodologies and tools change more frequently than principles
**Principles, Methodologies, and Tools**

**Example**

- **How can we fly?**
  - Principle: fight against the gravity

- **Hot air is lighter than cold air**
- **Buoyancy (lift-up force) > Gravity (drag-down force)**
How can we fly?

Principle: fight against the gravity

1. Air is diverted down from wings
2. Air pushes back
   - Newton’s third law
3. The airplane is lifted up into the sky
Key Principles

1. Rigor and formality
2. Separation of concerns
3. Modularity
4. Incrementality
5. Abstraction
6. Generality
7. Anticipation of change
Rigor and Formality

- Software Engineering must be practiced systematically
- Rigor is a necessary complement to creativity that increases our confidence in our developments
  - Computer is not the same as human, i.e., it is not able to handle vague commands
- Formality is rigor at the highest degree
  - Software process is driven and evaluated by mathematical laws
- Examples:
  - Mathematical (formal) analysis of program correctness
  - Systematic (rigorous) test data derivation
  - Rigorous documentation of development steps helps project management and assessment of timeliness
Separation of Concerns, Modularity

Separation of Concerns

- Separate the complexities and concentrate on one at a time
  - “Divide and conquer”
  - Example: keep product requirement separate, i.e., functionality, performance, usability
- Support parallelization of efforts and separation of responsibilities

Modularity: *the cornerstone principle*

- A complex system may be divided into simpler pieces called modules
- A system that is composed of modules is called modular
- Support application of separation of concerns
  - When dealing with a module we can ignore details of other modules
Modularity
Cohesion and Coupling

High coupling

Low coupling

Law of Demeter
- Each unit should only talk to its friends; don’t talk to strangers
- The resulting software tends to be more *maintainable* and *
adaptable*
Modularity
Top-down and bottom-up

- **Top-down**
  - Decompose the whole design into modules first and then concentrate on individual module design

- **Bottom-up**
  - Concentrate on modules and then on their composition

- They are two phases of the whole design process
Incrementality

- Process proceeds in a stepwise fashion (increments)
  - Deliver subsets of a system (prototype) early to get early feedback from expected users, then add new features incrementally
  - Deal first with functionality, then turn to performance
    - First step: quick-and-dirty solution
    - Second step: more efficient (i.e., cleaner) design

- Example: Game Design
  - First 2-D release then work on 3-D design
  - First coarse-grained texture then fine-grained texture
Abstraction

- Identify the important aspects of a phenomenon and ignore its details
- Special case of separation of concerns
- The type of abstraction to apply depends on purpose or roles (user or designer)
  - The user of a software concerns the what it does, not how it does
  - The designer in a team concerns what and how of his/her own part, and what of other parts that belong to other team members
    - After a module is fully tested and verified, only the what part is concerned by its user
Generality

- Discover if it is an instance of a more general problem whose solution can be reused in other cases
  - Increase the reusability
- Carefully balance generality against performance and cost
  - The solution to a generalized problem most likely is more expensive than the solution to a special problem
    - A program that can handle the multiplication between two general matrices
    - A program that is specially designed to handle the multiplication between two sparse matrices
Anticipation of Change

- Ability to support software evolution requires anticipating potential future changes
  - Minimize the changes to the existing module
    - Leave the spots in the program for future features

- Basis for software evolvability

- Example
  - A sorting program based on array of fixed length
  - A sorting program based on link of dynamic length
Outline

1. Importance of Principles
2. Key Principles
3. Case Study
   - Compiler Construction
The modular structure of a compiler

Source code → Lexical analysis → Parsing → Code Generation → Object code

Lexical analysis

“Tokenized” code

Lexical diagnostics

Symbol table

Parse tree → Syntax diagnostics

Intermediate code generation

Intermediate code

Machine code generation

Object code

Code Generation
Principles breakdown

- Rigor and formality
  - Compiler design is one of the most serious business in software development

- Separation of concerns, modularity, and incrementality
  - Correctness first, performance next, then ease of use
  - Decompose the whole project into small modules that can be designed and verified in a comparatively easy way
  - Cover only a subset of the source language and omit some features in the first release

- Abstraction and generality
  - Generate intermediate code for an *abstract machine*
    - Java Virtual machine
  - Use command-line arguments to specify the target machine
    - gcc or g++ compiler

- Anticipation of change
  - New instruction set with new processor architecture