SE 2AA4 Winter 2007

03 Software Engineering Principles

William M. Farmer

Department of Computing and Software
McMaster University

25 January 2007

Software Engineering Knowledge Units

- A principle is a general concept that is widely applicable in software engineering.
- A method or technique is a specific approach to software engineering.
- A methodology is a coherent collection of methods and techniques.
- A tool is a device that supports the application of a method, technique, or methodology.

Software Engineering Principles

- Used to reduce complexity.
- Form the basis for methods, techniques, methodologies, and tools.
- Can be used in all phases of software development.
- Can be applied to both process and product.
- All of the key software engineering principles are also key principles of mathematics and engineering as a whole!

Key Principles

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

Rigor

- An argument is valid if the conclusion is a logical consequence of the premises.
- Rigor is precise reasoning characterized by:
  - Only unambiguous language is used.
  - There are no hidden assumptions.
  - Care is taken to ensure that all arguments are valid.
- Rigor is achieved through the use of mathematics and logic.
- Rigor should be systematically employed throughout the whole software development process.

Formality

- Formality is reasoning in a formal system consisting of:
  - A language with a formal syntax and a precise semantics.
  - A set of syntactic rules.
- A formal system enables reasoning to be mechanized:
  - Reasoning is performed mechanically with computer assistance.
  - Arguments are machine checked.
  - Parts of the reasoning are automated.
- The use of formality in software development has a high cost:
  - The learning curve is very high.
  - Tool support and knowledge bases are inadequate.
  - The amount of detail involved is often overwhelming.
- Nevertheless, formality is the promise of the future!

Separation of Concerns

- Separation of concerns is the principle that different concerns should be isolated and considered separately.
  - Goal: To reduce a complex problem to a set of simpler problems:
  - Enables parallelization of effort.
- Concerns can be separated in various ways.
  - Different concerns are considered at different times.
  - Software qualities are considered separately.
  - A software system is considered from different views.
  - Parts of a software systems are considered separately.
- Dangers:
  - Opportunities for global optimizations may be lost.
  - Some issues cannot be safely isolated (e.g., security).

Separation of Concerns: Examples

- Separation of requirements from design.
- Separation of design from implementation.
- Decomposition of a system into a set of modules.
- The distinction between a module's interface and its implementation.

Modularity

- A modular system is a complex system that is divided into smaller parts called modules.
- Modularity enables the principle of separation of concerns to be applied to two ways:
  1. Different parts of the system are considered separately.
  2. The parts of the system are considered separately from their composition.
- Modular decomposition is the top-down process of dividing a system into modules.
  - This is the “divide and conquer” approach.
- Modular composition is the bottom-up process of building a system out of modules.
  - This is the “interchangeable parts” approach.

Properties of a Good Module

- To achieve the benefits of modularity, a software engineer must design modules with the two properties:
    1. High cohesion: The components of the module are closely related.
    2. Low coupling: The module does not strongly depend on other modules.
- This allows the modules to be treated in two ways:
    1. As a set of interchangeable parts.
    2. As individuals.

Abstraction

- Abstraction is the process of focusing on what is important while ignoring what is irrelevant.
    - Abstraction is a special case of separation of concerns.
- Abstraction produces a model of an entity in which the irrelevant details of the entity are left out.
    - Many different models of the same entity can be produced by abstraction. The models differ from each other by what is considered important and what is considered irrelevant.
    - Repeated application of abstraction produces a hierarchy of models.
- Refinement is the opposite of abstraction.
- Overabstraction produces models that are difficult to understand because they are missing so many details.

Anticipation of Change

- Anticipation of change is the principle that future change should be anticipated and planned for.
    - Also called design for change.
- Techniques for dealing with change:
    1. Configuration management: Manage the configuration of the software so that it can be easily modified as the software evolves.
    2. Information hiding: Hide the things that are likely to change inside of modules.
    3. Little languages: Create little languages that can be used to solve families of related problems.
- Since software is constantly changing, anticipation of change is crucial for the software development process.

Generality

- The principle of generality is to solve a more general problem than the problem at hand whenever possible.
- Advantages:
  - The more general a solution is the more likely that is can be reused.
  - Sometimes a general problem is easier to solve than a more specific problem.
- Disadvantages:
  - A general solution may be less efficient than a more specific solution.
  - A general problem may cost more to solve than a more specific problem.
- Abstraction is often used to extract a general solution from specific solution.

Incrementality

- The principle of incrementality is to attack a problem by producing successively closer approximations to a solution.
- Enables the development process to receive feedback and the requirements to be adjusted accordingly.
- Techniques for developing software incrementally:
  1. Rapid prototyping. Produce a prototype that is "thrown away" later.
  2. Refinement. A high-level artifact (like a requirements specification or a higher-level design) is incrementally refined into a low-level artifact (like a lower-level design or an implementation).