Khulna University
Computer Science & Engineering Discipline

SK Alamgir Hossain

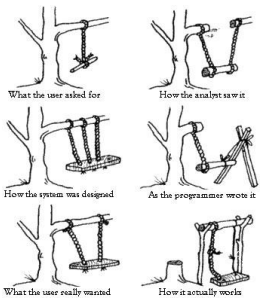
Assistant Professor
Computer Science & Engineering Discipline
Khulna University, Khulna, Bangladesh

http://alamgirhossain.com

CSE3203 - Software Engineering and Information System Design

SOFTWARE ENGINEERING

Khulna University



Chapter 2

Software Engineering Principles

Chapter Outline

- Principles from the basis of methods, techniques, methodologies and tools
- Seven important principles that may be used in all phases of software development
- Modularity is the cornerstone principle supporting software design
- Case studies

Lecture 2

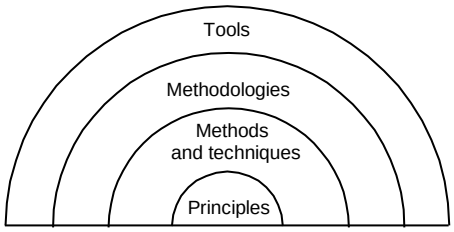
Software Engineering Principles

- Principles from the basis of methods, techniques, methodologies and tools
- Seven important principles that may be used in all phases of software development (four will be discussed)
- Modularity is the cornerstone principle supporting software design
- Case studies

Application of principles

- Principles/Rule apply to process and product
- Principles become practice through methods and techniques
  - often methods and techniques are packaged in a *methodology*
  - methodologies can be enforced by *tools*

A visual representation



Key principles

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

1. Rigor and formality

- Consider the ubiquitous traffic light system. Order at the traffic light junction comes about because of one reason: Drivers obey the rules - green to go, amber to get ready to stop and red to stop. When traffic lights go out of order, this rule becomes invalid; the intersection should be treated as an all way stop. Each driver is left on his/her own to decide when it is their turn and safe to cross the junction. The situation at the junction becomes informal and difficult, resulting in chaos.
- Software systems development is very much like the situation at a traffic light junction. Many people of differing skill-sets and interests are involved in the process. Without set rules, each developer imposes his/her own interest on the project. When problems occur, they become difficult to resolve.
- The *rigor* of a software development project is achieved by setting rules into the process. Every person involved in the project has to observe the rules. With rigor, a project can carry on smoothly. However, when projects are lacking in rigor, they are doomed to run into problems and fail, resulting in unreliable products, high costs, and time overrun.
- There are various degrees of rigor. The highest level of rigor is *formality* – a situation where software systems can be verified by mathematical laws.

Rigor and formality

- Software engineering is a creative design activity, BUT
  It must be practiced systematically
- Rigor increases the confidence level in our developments
- Formality is rigor at the highest degree
  - software process driven and evaluated by mathematical laws

Examples: product

- Mathematical (formal) analysis of program correctness
- Systematic (rigorous) test data derivation

Example: process

- Rigorous documentation of development steps helps project management and assessment of timeliness

2. Separation of concerns

- "Divide and Conquer" is a phrase you might have heard before. This phrase (in Latin *divide et impera*) reflects an approach practiced by the ancient Romans in their conquest of other nations – divide and isolate the nations, then conquer them one by one. When this approach is applied to software development, the same happens – divide a larger problem into multiple smaller sub-problems; solve the sub-problems individually and the larger problem is said to be solved.
- "Separation of concerns" is similar to "Divide and Conquer" in that it allows us to deal with different aspects of a problem and focus on each separately. There are many areas of concerns in software development.
  - Examples include software functionalities, user interface design, hardware configuration, software applications, space and time efficiency, team organization and structure, design strategies, control procedures, error handling, and budgetary matters. By separating the multiple concerns and focusing on them individually, the inherent complexity of a large-scale software project can be greatly reduced and better managed.

Separation of concerns

- To dominate complexity, separate the issues to concentrate on one at a time
- "Divide & conquer"
- Supports parallelization of efforts and separation of responsibilities

Example: process

- Go through phases one after the other
  - Does separation of concerns by separating activities with respect to time

Example: product

- Keep product requirements separate
  - functionality
  - performance
  - user interface and usability

3. Modularity

- A complex system may be divided into simpler pieces called *modules*
- A system that is composed of modules is called *modular*
- Supports application of separation of concerns
  - when dealing with a module we can ignore details of other modules

4. Abstraction

- General users of television sets, a useful abstraction would be a description of the functionalities and how the functionalities can be achieved via the available buttons. However, to a television repairman, a useful abstraction would be a box he can dismantle for examination and repair. In other words, for the same reality, there can be different abstractions, each providing a view of the reality and serving some specific purposes.
- Abstractions have long been applied in software development. Even a programming language is a form of abstraction. A high-level programming language hides the bits and the manipulation of the bits via the registers from the programmer. It does so by providing constructs to achieve the functionalities. Even the comment at the beginning of a module describing what the module does is a form of abstraction. It describes the module without revealing how the module implements its function.

Lecture 3

Software Engineering Principles

- Principles from the basis of methods, techniques, methodologies and tools
- Seven important principles that may be used in all phases of software development (cont...)
- Modularity is the cornerstone principle supporting software design
- Case studies

5. Cohesion and coupling

Khulna University

- **Cohesion** refers to what the class (or module) will do.
  - Low cohesion would mean that the class does a great variety of actions and is not focused on what it should do.
  - High cohesion would then mean that the class is focused on what it should be doing, i.e. only methods relating to the intention of the class.

Example of Low Cohesion:

```
| Staff            |
|------------------|
| checkEmail()     |
| sendEmail()      |
| emailValidate()  |
| PrintLetter()    |
```

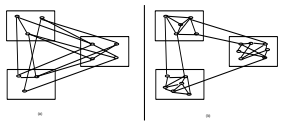
Example of High Cohesion:

```
| Staff                    |
|--------------------------|
| -salary                  |
| -emailAddr               |
|--------------------------|
| setSalary(newSalary)     |
| getSalary()              |
| setEmailAddr(newEmail)   |
| getEmailAddr()           |
```


5. Cohesion and coupling

Khulna University

- As for **coupling**, it refers to how related are two classes / modules and how dependent they are on each other.
  - Low coupling would mean that changing something major in one class should not affect the other.
  - High coupling would make your code difficult to make changes as well as to maintain it, as classes are coupled closely together, making a change could mean an entire system revamp.



high coupling low coupling

5. Cohesion and coupling

- Each module should be *highly cohesive*
  - module understandable as a meaningful unit
  - Components of a module are closely related to one another
- Modules should exhibit *low coupling*
  - modules have low interactions with others
  - understandable separately

6. Generality (Anticipation of change)

- While solving a problem, try to discover if it is an instance of a more general problem whose solution can be reused in other cases
- Carefully balance generality against performance and cost
- Sometimes a general problem is easier to solve than a special case

7. Incrementally

- Process proceeds in a stepwise fashion (*increments*)
- Examples (process)
  - deliver subsets of a system early to get early feedback from expected users, then add new features incrementally
  - deal first with functionality, then turn to performance
  - deliver a first prototype and then incrementally add effort to turn prototype into product

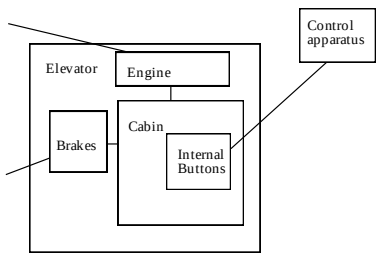
Case study : elevator system

- Rigor and formality
  - Define requirements
    - must be able to carry up to 400 Kg. (safety alarm and no operation if overloaded)
    - emergency brakes must be able to stop elevator within 1 m. and 2 sec. in case of cable failures
  - Later, verify their fulfillment

Separation of concerns

- Try to separate
  - safety
  - performance
  - usability (e.g, button illumination)
  - cost
- although some are strongly related
  - cost reduction by using cheap material can make solution unsafe

A modular structure

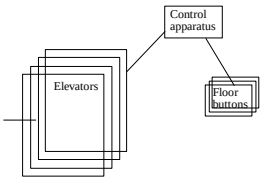


Abstraction

- The modular view we provided does not specify the behavior of the mechanical and electrical components
  - they are abstracted away

Anticipation of change, generality

- Make the project parametric wrt the number of elevators (and floor buttons)



Class Test 1

- Will be on Chapter 1 and 2
- Total Marks will be: 15
- Total Time will be: 40min
- On next class

THANK YOU