5SENG007C
Software Engineering Principles and Practice

Week 5 Lecture
Software Engineering "Principles"

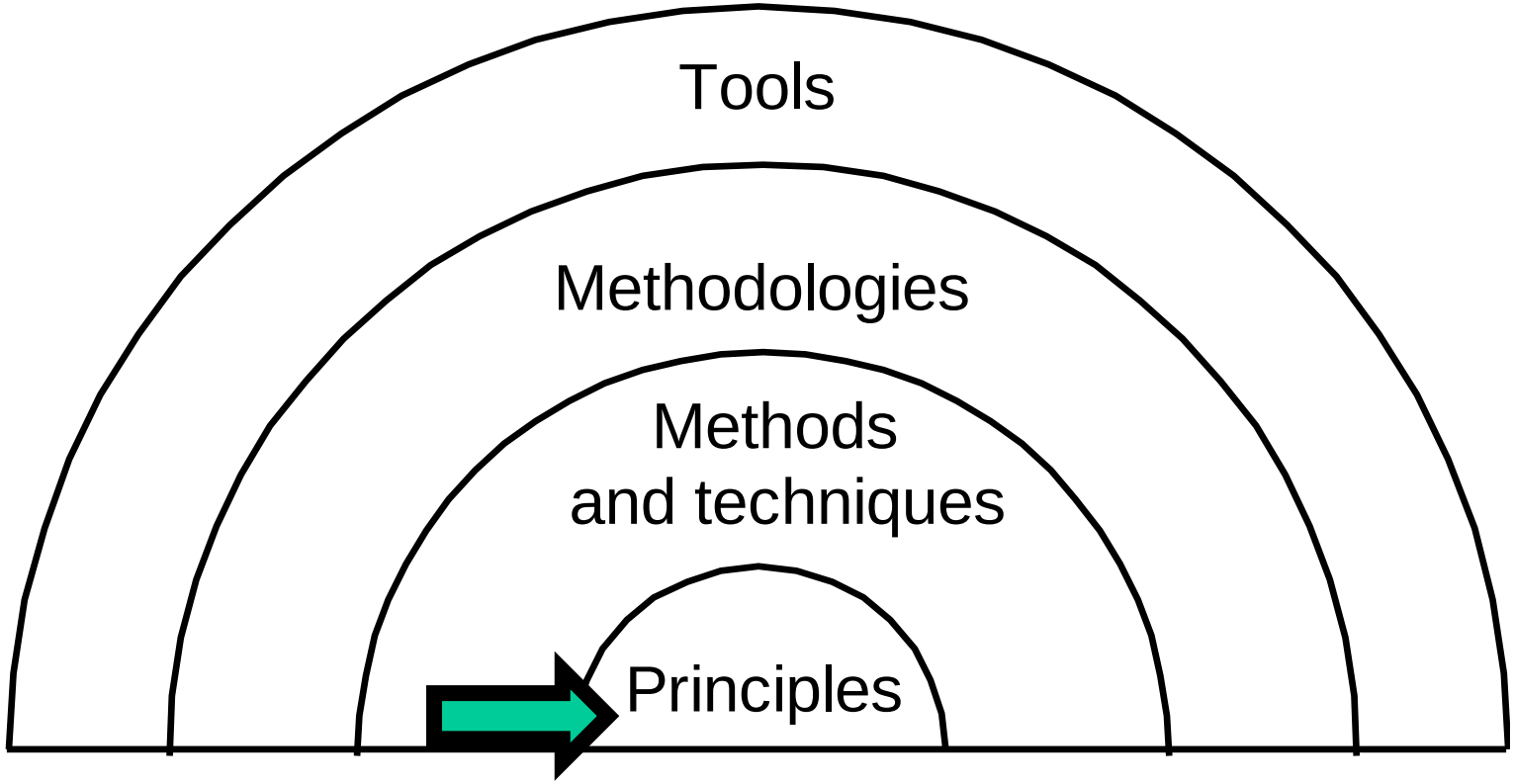
Modules, interfaces, separation of concerns, and programming patterns.

Practical aspects of abstractions, invariants.

(notes are adopted from UoW ML - Dr Alexander Bolotov)
Please read the detailed slides with extra notes on BB

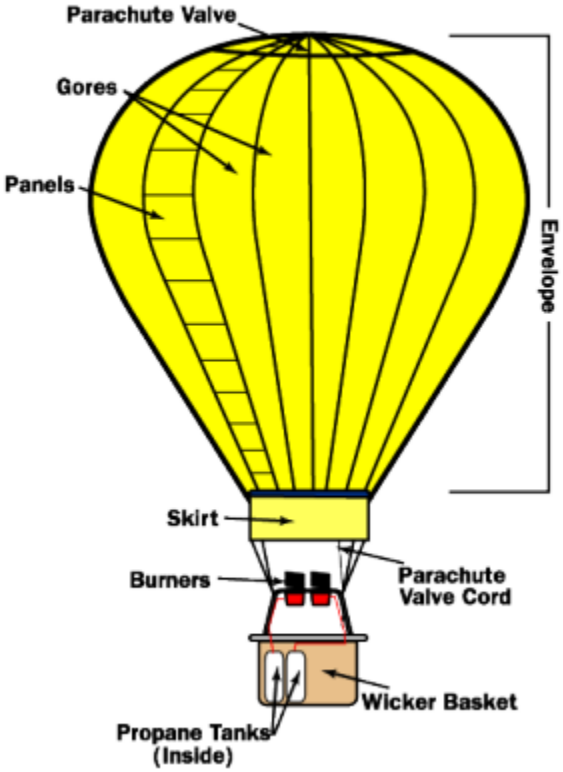
From Principles to Tools

Principles form the basis of methods, techniques, methodologies and tools. We explore "Principals" today.



Principles

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



- Hot air is lighter than cold air
- Buoyancy (lift-up force) > Gravity (drag-down force)

Principles

- Seven important principles that may be used in all phases of software development

  - **Rigor and formality**
  - Separation of concerns
  - Modularity
  - Abstraction
  - Anticipation of change
  - Generality
  - Incremental Development

- Modularity is the cornerstone principle supporting software design

Main principles for Software Development

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- Generality
- Incremental Development

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

Source: http://www.csce.uark.edu/~mqhuang/courses/3513/s2010/lectures/SE_Lecture_3.pdf

- Rigor and formality
- **Separation of concerns**
- Modularity
- Abstraction
- Anticipation of change
- Generality
- Incremental Development

Separation of concerns

- To dominate the complexity, separate the issues to concentrate on one at a time
- "Divide & conquer" (*divide et impera*)
- Supports parallelization/synchronisations of efforts and separation of responsibilities

Separation in Product

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

What to Separate

- Time
  - Life cycle models
- Qualities (qualitative analysis of software)
- Views
  - activity versus control
- Problem Domain from Implementation Domain

- Rigor and formality
- Separation of concerns
- **Modularity**
- Abstraction
- Anticipation of change
- Generality
- Incremental Development

Modularity

- A complex system (S) may be divided into simpler subsystems called *modules* (M).
- A system (S) that is composed of modules (M1 … Mn) is called *modular.*
- Modularity supports separation of concerns

  - dealing with a module we can ignore details of other modules

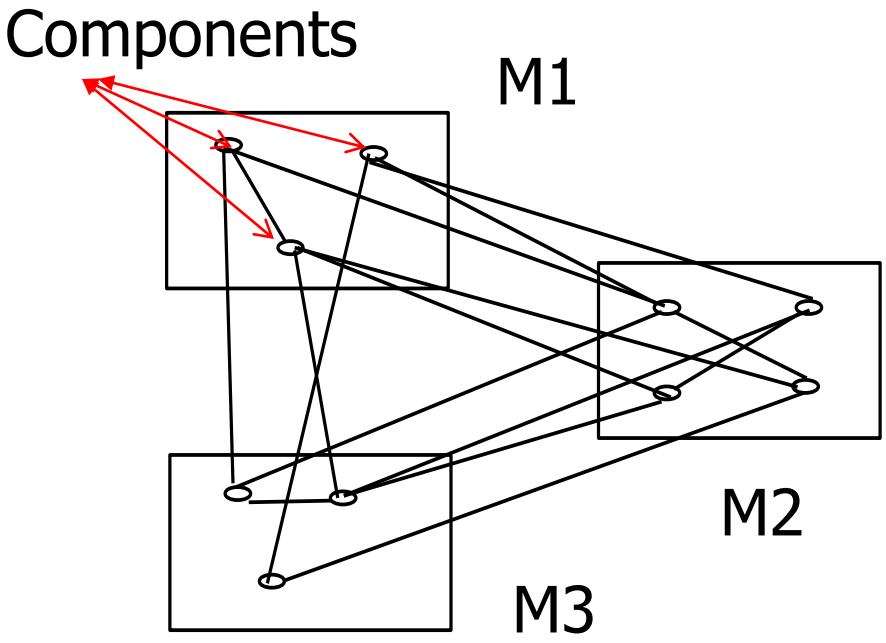
Cohesion and coupling: how to tackle these

- Each module Mi from M1 …. Mn should be *highly cohesive*
  - module Mi is a "self-contained" meaningful unit, a system itself, which is a subsystem of a given system S.
  - Components of a module are closely related to one another
- Modules should exhibit *low coupling*
  - modules have low interactions with others
  - understandable separately

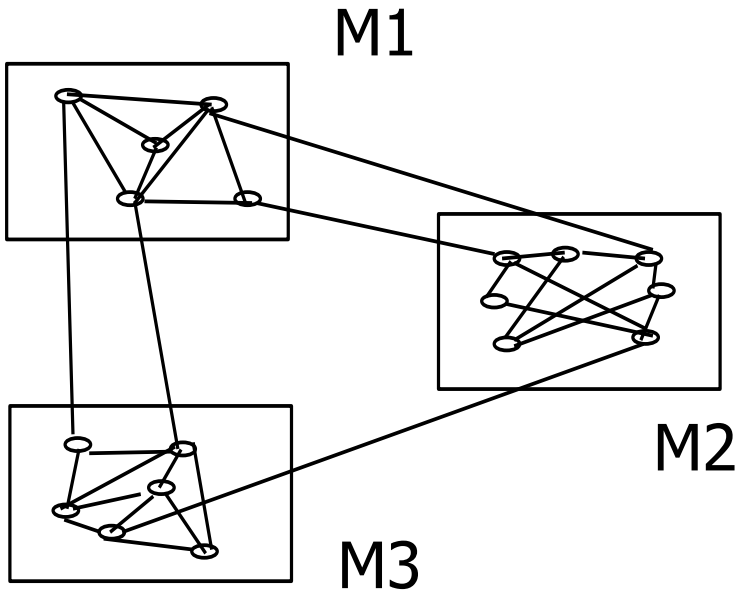
Cohesion and coupling: how to tackle these

- On the contrary,
- Modules should exhibit *low coupling*
  - modules should have low interactions with others (otherwise, they cannot be designed on their own, e.g. abstraction from the other modules and the connectivity would not work)
  - Modules are understandable separately

Cohesion & Coupling: a visual representation



Avoid: high coupling of the modules M1 – M3 and low cohesion within them



Target: high cohesion within modules M1 – M3 and low coupling

Abstraction

- Rigor and formality
- Separation of concerns
- Modularity
- **Abstraction**
- Anticipation of change
- Generality
- Incremental Development

- Identify the important aspects of a phenomenon and ignore its details
- Linked closely to separation of concerns
- Type of abstraction to apply depends on purpose
- Example: the user interface of a watch (its buttons) abstracts from the watch's internals for the purpose of setting time.

Abstraction ignores details

- Abstraction: we have done it already many times!
- You make your own abstraction introducing the context of the system,
- You make your abstractions when prioritise requirements
- You make your abstractions when you define data types, i.e. defining the precision

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- **Anticipation of change**
- Generality
- Incremental Development

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

Source: http://www.csce.uark.edu/~mqhuang/courses/3513/s2010/lectures/SE_Lecture_3.pdf

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- **Generality**
- Incremental Development

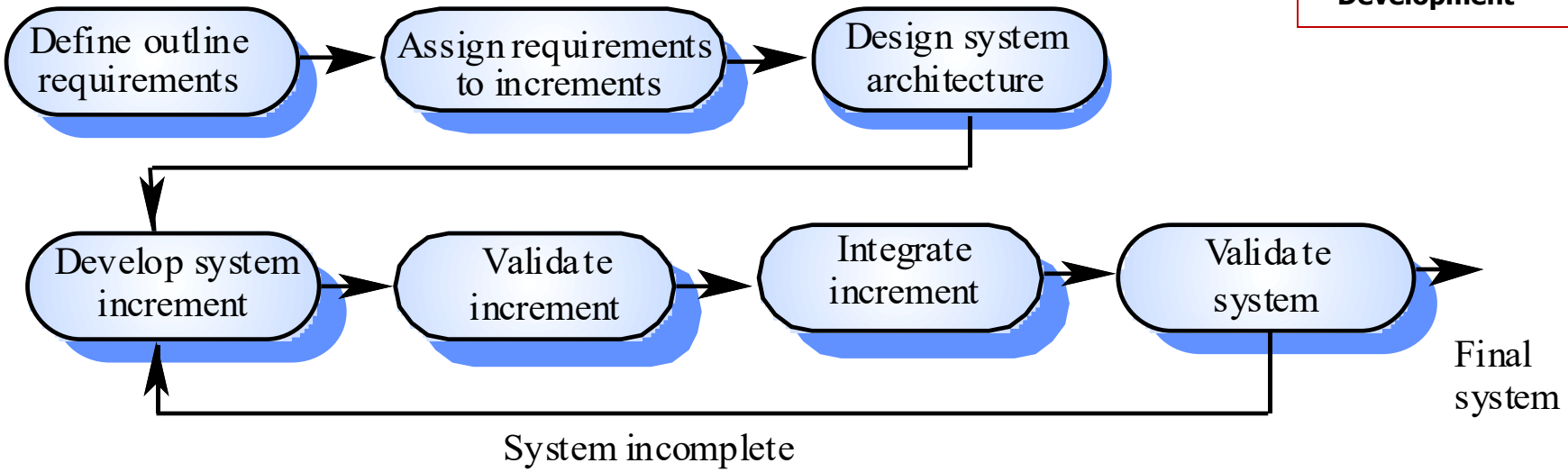
Generality

- 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

Source: http://www.csce.uark.edu/~mqhuang/courses/3513/s2010/lectures/SE_Lecture_3.pdf

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- Generality
- **Incremental Development**

Incremental development



- Rather than deliver the system in a single attempt, **the development and delivery is broken down into increments** with each increment delivering part of the required functionality
- **User requirements are prioritised** and the highest priority requirements are included in early increments
- **Once the development of an increment is started, the requirements are frozen** though requirements for later increments can continue to evolve

Incremental development advantages

- **Customer value** can be delivered with each increment so system functionality is available earlier
- **Early increments** act as a prototype to help elicit requirements for later increments
- Lower risk of overall project failure
- **The highest priority system services** tend to receive the most testing

LINKING TOGETHER

- Methodology:
  - Increments
  - Development & Testing

Levels of testing

- Unit testing:
  functions, member functions etc
- Module testing:
  classes, les etc
- Sub-system testing:
  libraries etc
- System testing
- Integration testing
- Acceptance testing, alpha and beta testing
- Review and Maintenance

How to Sell

- Meet the requirements -> Functionality
- Transparent & Efficient Design
- Usability -> HCI principles
- Evolution -> Easy to upgrade, plug in new modules
- Link to the methodology – increments and modular development
- Link to Evaluation & Testing