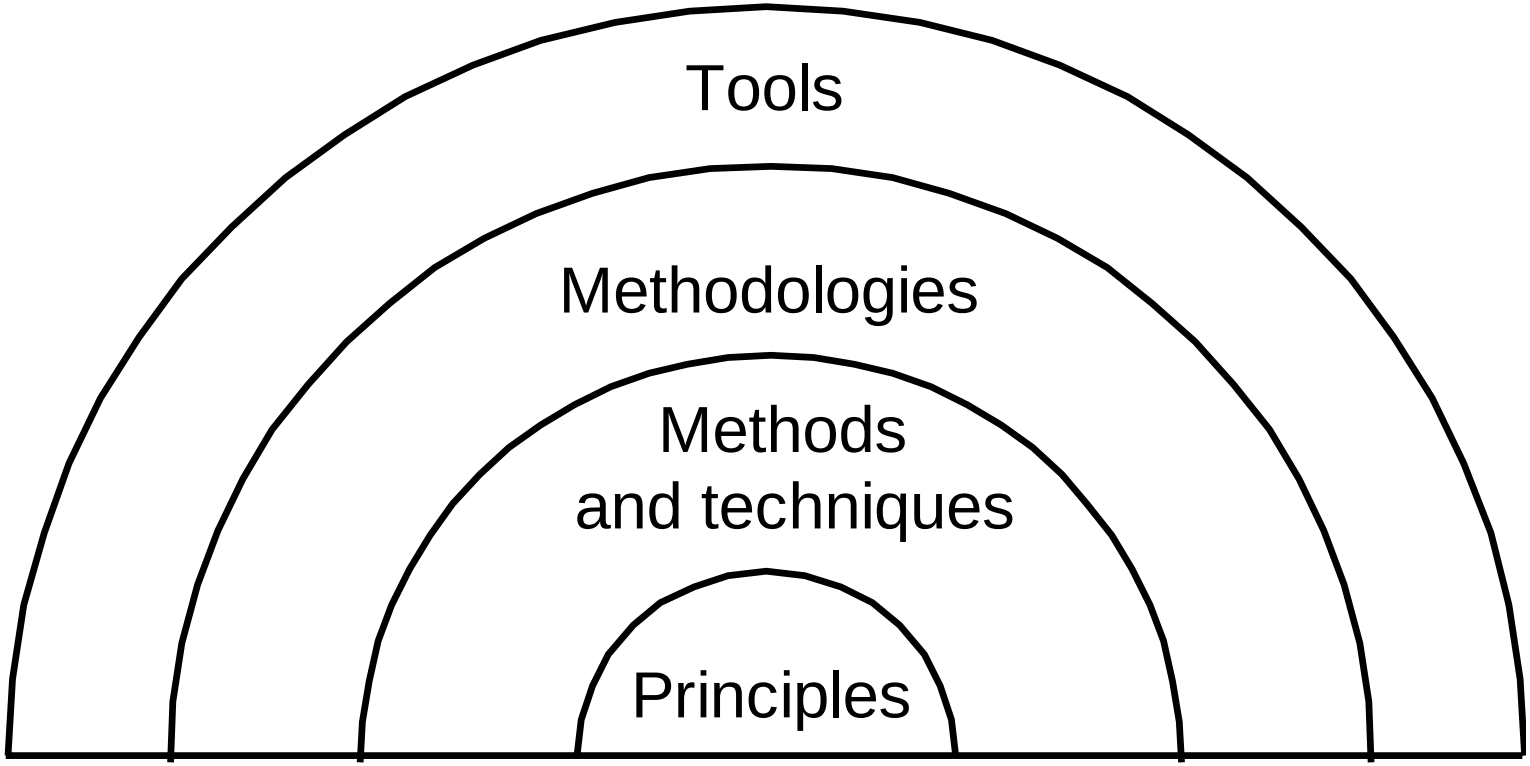
5SENG007W
Software Engineering Principles and Practice

Week 5 Lecture

Software Engineering Principles: Modules, interfaces, separation of concerns, and programming patterns. Practical aspects of abstractions, invariants.

From Principles to Tools

Principles form the basis of methods, techniques, methodologies and tools



INTRODUCING THE ISSUES

- 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
- **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

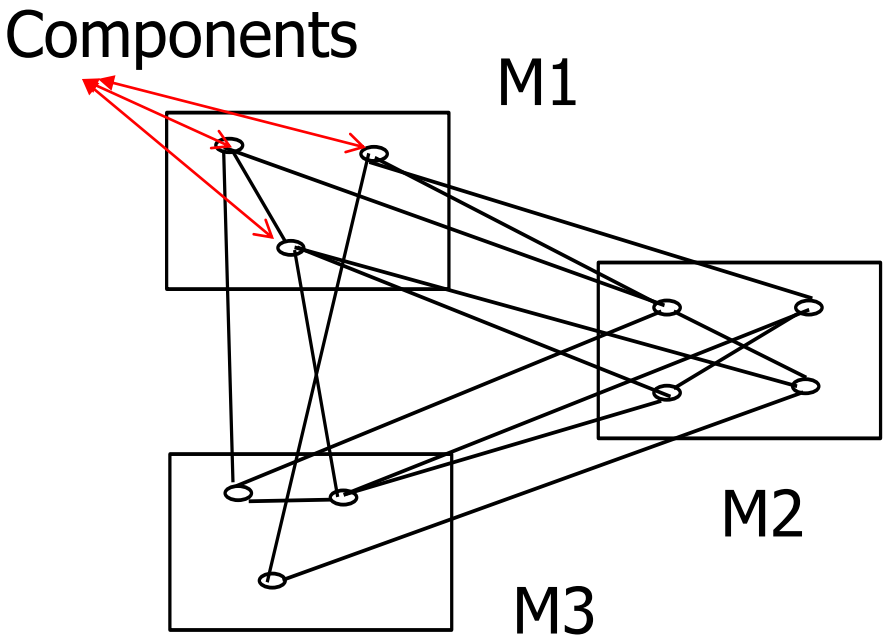
Cohesion and coupling: how to tackle these

- 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

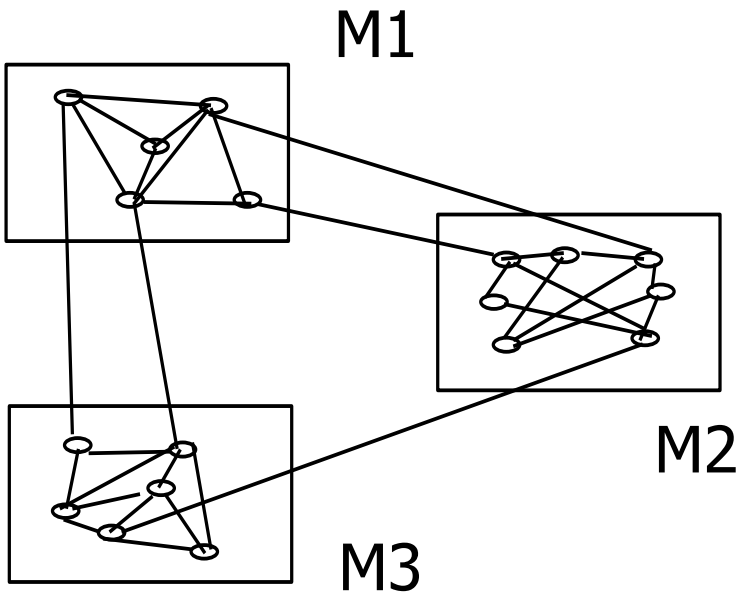
Cohesion and coupling: how to tackle these

- 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



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



(b) 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

LINKING TOGETHER

- Methodology:
  - Increments
    - Development & Testing

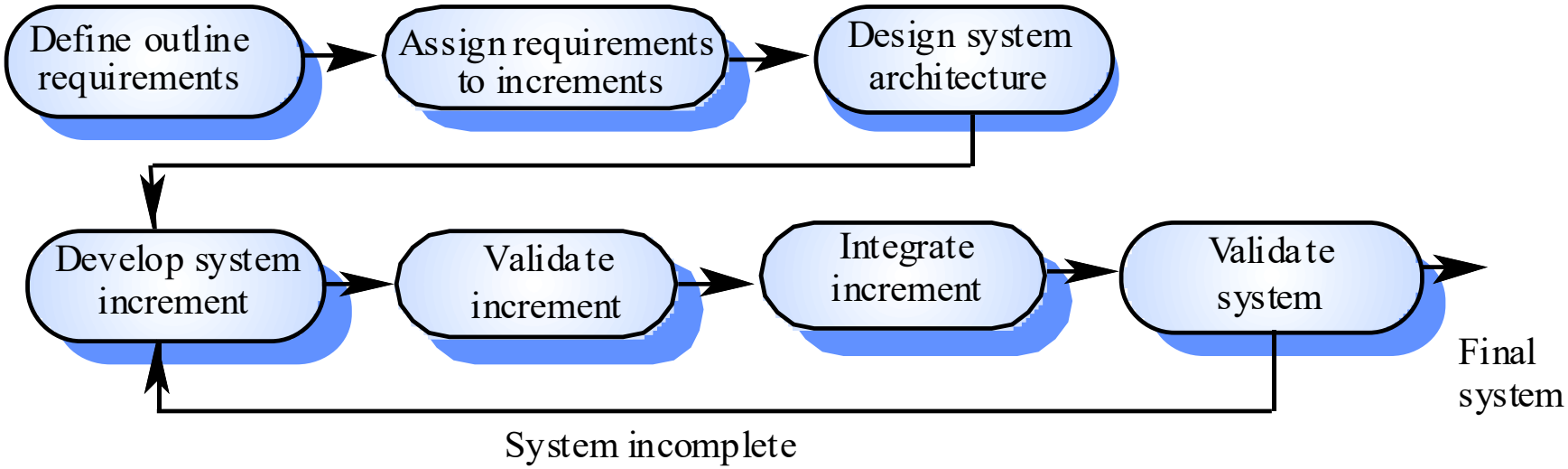
Levels of testing

- 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

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



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

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