CMPS1134
Fundamentals of Computing

Software Engineering 1

Computer Science: An Overview
Eleventh Edition
J. Glenn Brookshear
Chapter 7

Chapter 7: Software Engineering

- **The Software Engineering Discipline**
- **The Software Life Cycle**
- **Software Engineering Methodologies**
- **Modularity**
- **Coupling versus Cohesion**
- **Information Hiding and Components**

- Tools of the Trade
- Quality Assurance
- The Human Machine Interface
- Software Ownership and Liability

The Software Engineering Discipline

- Distinct from other engineering fields
  - **Prefabricated components**: SE lags behind – many systems are built from scratch
  - **Metrics**: Difficult to measure software properties in a quantitative manner
- Practitioners versus Theoreticians
  - **Practitioners** develop techniques for immediate applications
  - **Theoreticians** search for underlying principles and theories on which stable techniques may be constructed
- CASE Tools and IDEs have helped to streamline and simplify the software development process
- Professional Organizations: ACM, IEEE, etc.
  - Codes of professional ethics
  - Standards

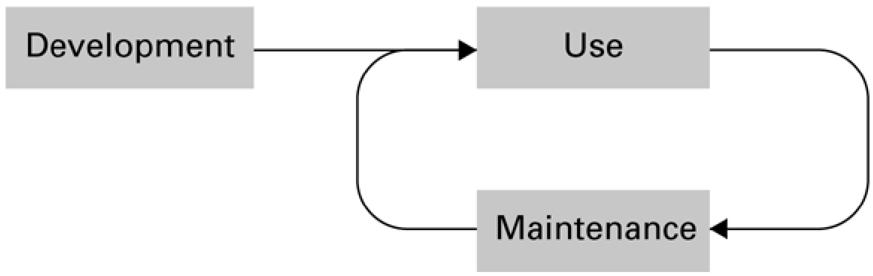
The Software Engineering Discipline

Computer Aided Software Engineering (CASE) tools

- Project planning
- Project management
- Documentation
- Prototyping and simulation
- Interface design
- Programming (IDEs)

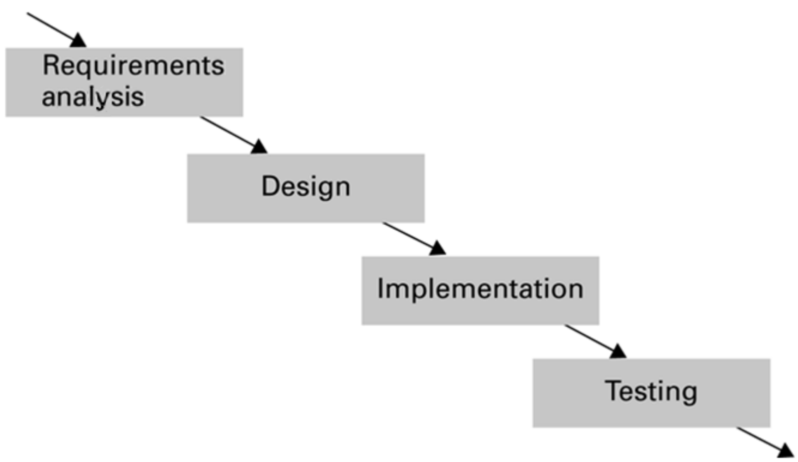
The Software Life Cycle

The software life cycle (Fig 7.1)



The Software Life Cycle

The development phase of the software life cycle (Fig 7.2)



The Software Life Cycle

Requirements Analysis Stage

Goal is to:
- Specify what services the proposed system will provide
- Identify any conditions (time constraints, security, etc.) on those services
- How the outside world will interact with the system

- Requirements analysis process
  - Compile and analyze software users needs
  - Negotiate with stakeholders' trade-offs (wants, needs, cost, and feasibility)
  - Determine requirements that identify the features and services the finished software system must have
- Software requirements specification
  - Based on requirements analysis
  - Output document that satisfies the goal

The Software Life Cycle

Design Stage

Involves creating a plan for the construction of the proposed system.

- Flawed perspective (layperson) is that **Requirements Analysis** is about "**what**" the system will do while **Design** is about "**how**" the system will do it.
- In actuality Requirements Analysis considers more of "how" it will be done and Design considers more of "what" will be done
- Methodologies and tools (discussed later)
  - Various diagramming and modeling methodologies in designing the software
- Human interface (psychology and ergonomics) a key component in Design

The Software Life Cycle

Implementation Stage

- Create system from design
  - Write programs
  - Create data files
  - Develop databases
- Role of “software analyst” versus “programmer”
  - **Software Analyst** is involved with the entire development process with emphasis on the Requirements Analysis and Design steps
  - **Programmer** is involved primarily with the implementation step
  - Note: The use of the above terminology is the common usage but may be interchanged in some circumstances

 9

The Software Life Cycle

Testing Stage

- Validation testing
  - Confirm that system meets specifications
- Defect testing
  - Find bugs

 10

Software Engineering Methodologies

- **Waterfall Model** (flow in one direction)
- **Incremental Model**
  - Constructed in increments (extending)
  - **Evolutionary Prototyping**
- **Iterative Model**
  - Refining each version and may incrementally add features
  - Significant example: IBM's **Rational Unified Process (RUP)**/ Non-commercial Unified Process
  - **Throwaway Prototyping** (e.g. Rapid Prototyping)
- **Open-source Development**
  - Less formal (initial version of software is modified and corrected by multiple authors)
  - Used widely for open-source development (e.g. Linux)
- **Agile Methods**
  - Early/ quick implementation on an incremental basis, responsiveness to changing requirements, and reduced emphasis on rigorous requirements analysis
  - Example: Extreme Programming

 11

Modularity

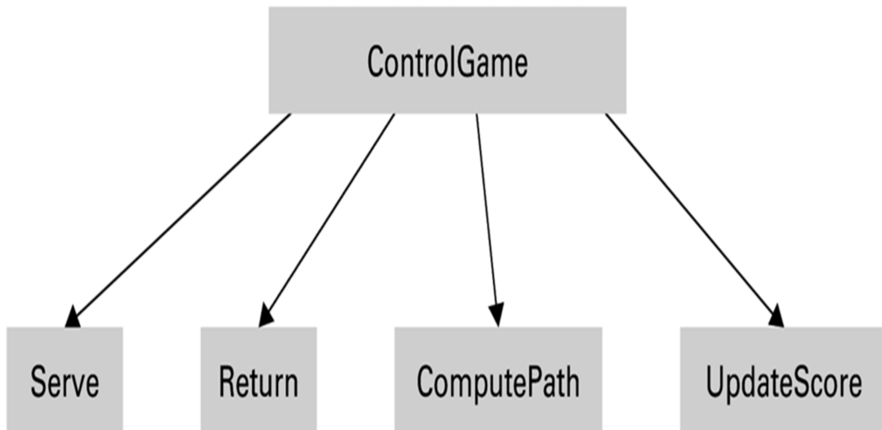
Modules depending on the context may appear as procedures or as objects.

- Procedures -- Imperative paradigm
  - **Structure chart**
    - Does not indicate how each procedure will perform its task
    - Identifies procedures and indicates dependencies among procedures
- Objects -- Object-oriented paradigm
  - **Template** for class
    - Defines the methods and attributes associated with each object
  - **Collaboration diagram**
    - Presents the communication between objects

 12

Modularity

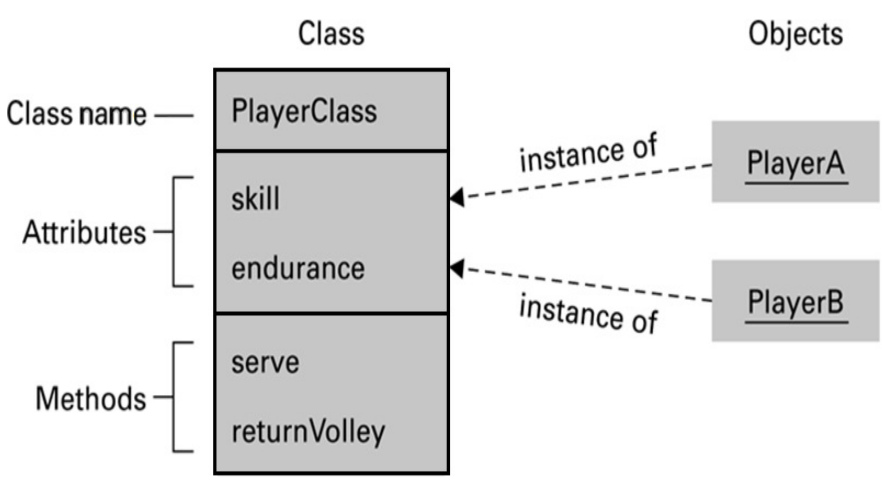
A simple structure chart (Fig 7.3)



 13

Modularity

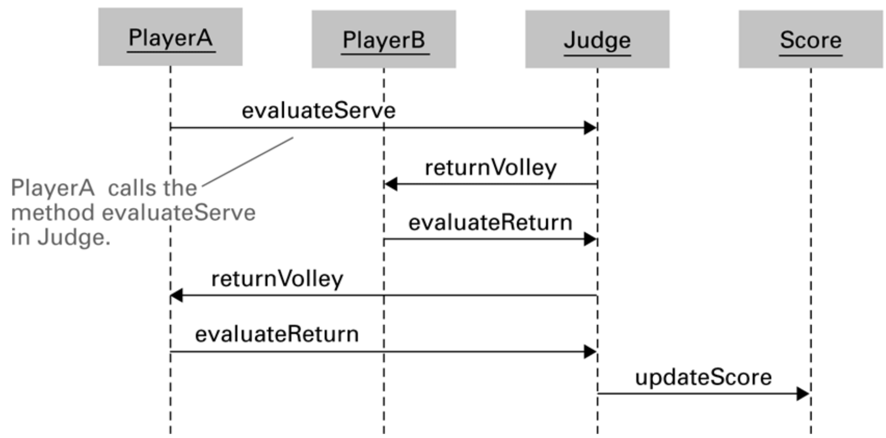
The structure of PlayerClass and its instances (Fig 7.4)



 14

Modularity

The interaction between objects resulting from Player A's serve (Fig 7.5)



 15

Coupling versus Cohesion

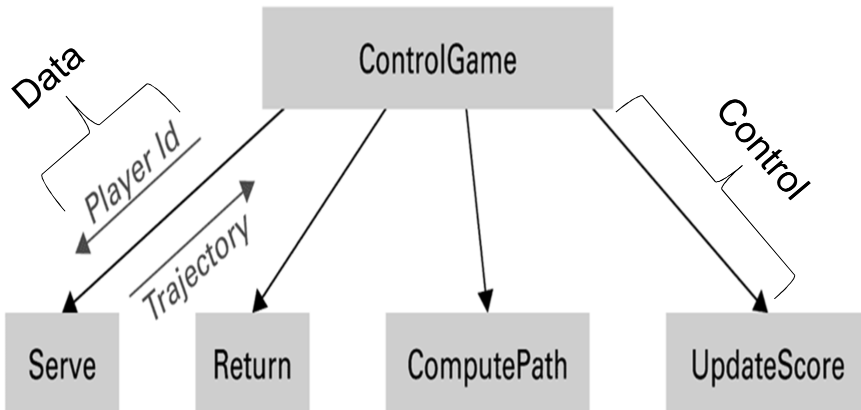
Modular systems should maximize independence among modules/ minimize <u>inter-module coupling</u>.

- **Coupling** (minimize)
  Linkage between modules
  - **Control coupling** – module passes control of execution to another module (fig. 7.3 & 7.5)
  - **Data coupling** – sharing of data between modules (fig. 7.6)
- **Cohesion** (maximize)
  Maximize the internal bindings within each module (degree of relatedness of a module's internal parts).
  - **Logical cohesion** – internal elements of a module perform activities logically similar in nature
  - **Functional cohesion** – all the parts of a module are focused on the performance of a single activity

 16

Coupling

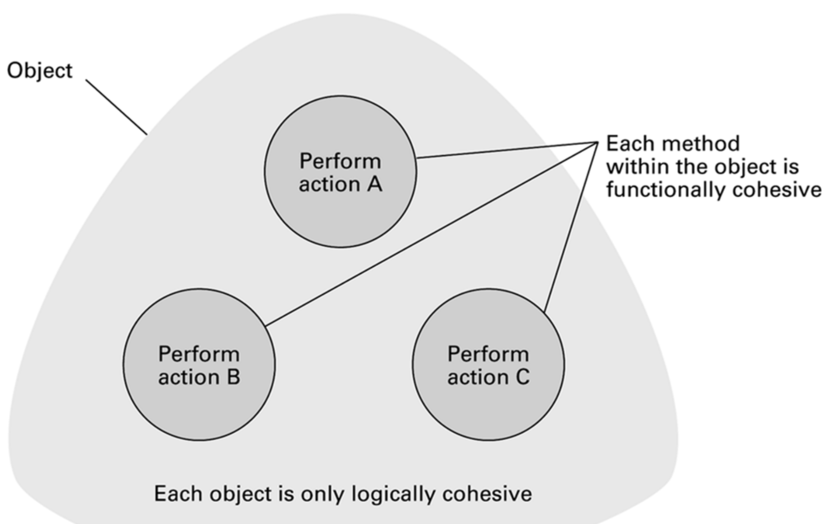
Structure chart including data coupling (Fig 7.6)



 17

Coupling

Logical and functional cohesion within an object (Fig 7.7)



 18

Information Hiding and Components

- **Information Hiding**
  - Central to abstraction "black box" concept
  - Restricts information to specific portions of a software system
  - Internal data/ structure of a module is restricted from access by other modules (it is "hidden")
  - Avoids corruption/ malfunction
- **Components**
  Re-usable units of software.
  - **Component architecture** – traditional role of a programmer is replaced by a **component assembler** that constructs software systems from prefabricated components (usually icons in a graphical interface)
  - Minimizes internal programming of components and maximizes the ease of creation of software through the integration of predefined components
  - Example: smartphone systems that utilize collaborating components to overcome resource constraints of the devices

Chapter 7: Topics Covered

- The Software Engineering Discipline
- The Software Life Cycle
- Software Engineering Methodologies
- Modularity
- Coupling versus Cohesion
- Information Hiding and Components

