Detailed Design

Sommerville: Section 7.1
Tsui: 7.1, 7.3, 8.1-8.4

Detailed Design
Outline:

I. Software Design and Implementation

II. Design Processes
1. Functional Decomposition
2. Relational Database Design
3. Object-oriented Design and UML

III. Design Characteristics and Metrics
1. Halstead Complexity Metric
2. McCabe's Cyclomatic Complexity
3. Coupling and Cohesion
4. Law of Demeter

I. Software Design

- Process of converting the requirements into the design of the system.
- Definition of how the software is to be structured or organized.

- For large systems, this is divided into two parts:
  - **Architectural design** defines main components of the system and how they interact.
  - **Detailed design:** the main components are decomposed and described at a much finer level of detail.

Design and Implementation

- Software **Design**:
  - Creative activity, in which you:
  - Identify software components and their relationships
  - Based on requirements.
- **Implementation** is the process of realizing the design as a program.
- Design may be
  - Documented in UML (or other) models
  - Informal sketches (whiteboard, paper)
  - In the programmer's head.
- How detailed and formal it is depends on the process that is in use.

II. Design Processes

- Functional Decomposition
  - aka: Top down design
- Relational Database Design
- Object-oriented design and UML
  - class diagrams
  - state diagrams
  - etc.
- [User Interface design]

1. Functional Decomposition
Top-Down Design

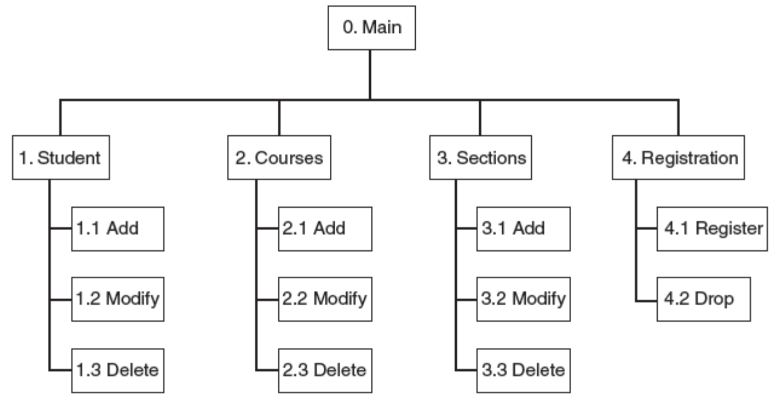
Definition: A software development technique that imposes a hierarchical structure on the design of the program. It starts out by defining the solution at the highest level of functionality and breaking it down further and further into small routines that can be easily documented and coded (The Free Dictionary: top-down programming).

- Used in procedural programming
  - Start with a “main module”
  - Repeatedly decompose into sub-modules.
  - Lowest level modules can be implemented as functions.
- Can be used in Object-oriented design
  - to do initial decomposition of a system (Arch design)
  - to decompose member functions that are particularly hard to implement.

Functional Decomposition: example student registration system

- Design a system for managing course registration and enrollment.
- Requirements: The system shall allow the user to:
  - add, modify and delete students
  - add, modify and delete courses
  - add, modify, and delete sections for a given course
  - register and drop students from a section.
- Main module divided into four submodules (students, courses, sections, registration)
- Decompose each into its tasks (and subtasks).

Functional Decomposition: example student registration system



2. Relational Database Design

- Many software systems must handle large amounts of data
- In a relational database, data is stored in **tables**
  - row corresponds to an object or entity
  - columns correspond to attributes of the entities
  - (basically an array of structs)
- Structured Query Language (SQL) is a set of statements that
  - create the tables
  - add and modify data in the tables
  - retrieve data that match specified criteria

Relational Database Design

- Database design concentrates on
  - how to represent the data of the system in a database, and
  - how to store it efficiently

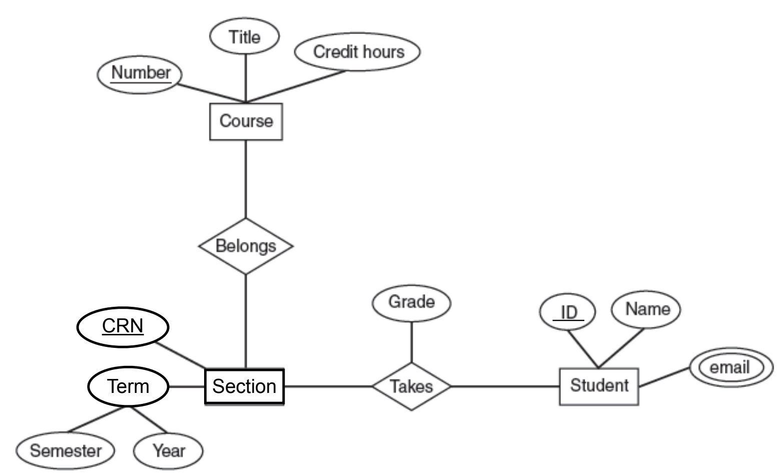
Three phases:

- Data modeling
  - create a model showing the entities with their attributes, and how the entities are related to each other
- Logical database design
  - maps the model to a set of tables
  - relationships are represented via attributes called foreign keys
- Physical database design
  - deciding on types of attributes, how tables are stored, etc.

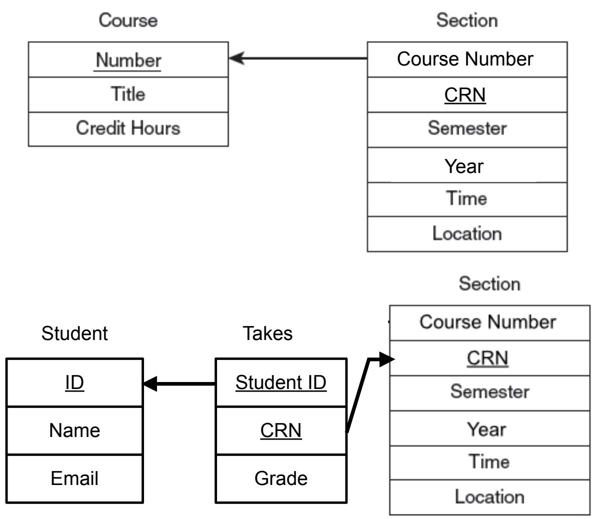
Relational Database Design

- Data modeling: ER diagram
  - Entities: rectangles
  - Attributes: ovals
  - Relationships: diamonds
- Identifier
  - special attribute that has a unique value for each entity (underlined)
- Relationships can be
  - one to one
  - one to many
  - many to many

Relational Database design: ER diagram student registration system



Student registration system: tables



Course Number is a foreign key, used to represent the "Belongs" relationship

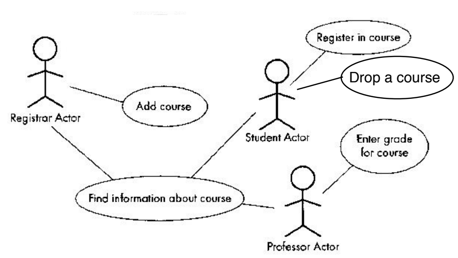
Student ID and CRN are foreign keys, used to represent the many-to-many relationship

3. Object-oriented design

- Object-oriented system is made up of interacting objects
  - Maintain their own local state (private).
  - Provide operations over that state.
- Object-oriented design process involves
  - Designing classes (for objects) and their interactions.
- Documentation is presented in UML diagrams
  - UML = Unified Modeling Language.
  - a graphic design notation (for diagrams/models).

1 Requirements elicitation

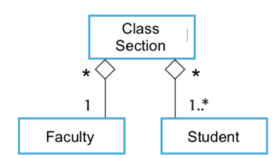
- Client and developers define the purpose of the system:
  - Develop use cases
  - Determine functional and non-functional requirements
- Major activities
  - Identifying actors.
  - Identifying scenarios.
  - Identifying use cases.
  - Refining use cases.



Use case diagrams

2 Object Oriented Analysis

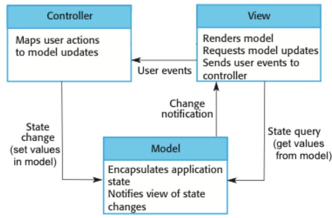
- Developers aim to produce a model of the system
  - Model is a class diagram
  - Describing real world objects (only) (as in the SRS)



- Goal: transform use cases to objects
- Major activities
  - Identifying objects: entities from the real world
    - Look for nouns in use cases
  - Drawing the UML class diagram, with relationships
  - Drawing UML state diagrams as necessary

3 System Design (architecture)

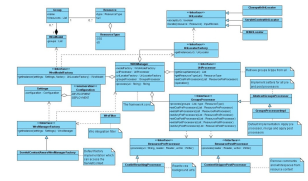
- Developers decompose the system into smaller subsystems (see ch 6).



- Major activities
  - Identify major components of the system and their interactions (including interfaces).
    - Use architectural patterns
  - Identify design goals (non-functional requirements)
  - Refine the subsystem decomposition to address design goals

4 Object Design

- Developers complete the object model by adding implementation classes to the class diagram.



- Major activities
  - Interface specification: define public interface of objects
  - Reuse:
    - frameworks, existing libraries (code)
    - design patterns (like arch. patterns at object level)
  - Restructuring: maintainability, extensibility

5 Implementation

```cpp
#include <string>
#include <iomanip>
#include <sstream>
#include<iostream>
using namespace std;

// models a 12 hour clock
class Time        //new data type
{
private:
    int hour;
    int minute;
    void addHour();

public:
    void setHour(int);
    void setMinute(int);
    int getHour() const;
    int getMinute() const;

    string display() const;
    void addMinute();
};

// class function implementations

void Time::setHour(int hr) {
    hour = hr;            // hour is a member var
}
void Time::setMinute(int min) {
    minute = min;         // minute is a member var
}
int Time::getHour() const {
    return hour;
}
int Time::getMinute() const {
    return minute;
}

void Time::addHour() {  // a private member func
    if (hour == 12)
        hour = 1;
    else
        hour++;
```

- Developers translate the class diagram into source code.
- Goal: map object model to code.

- Major activities
- Map classes in model to classes in source language
- Map associations in model to collections in source language
  - OO languages don't have "associations"
  - tricky: maintaining bidirectional associations
- Refactoring

III. Design characteristics and metrics

- Legacy Characteristics of Design Attributes
  - Programming and programming modules were considered the most important artifacts.
  - Metrics and characteristics focused on the code (and very detailed design, if any).
- More Current Good Design Attributes
  - Design diagrams/models are considered the important design artifacts now.
  - Simplicity is the main design goal now (simplify a complex system into smaller pieces, etc.)
  - How do we measure simplicity?

Halstead Complexity Metric

- Analyze source code to determine:
  - n1 = number of distinct operators
  - n2 = number of distinct operands
  - N1 = total number of operators (counting duplicates)
  - N2 = total number of operands (counting duplicates)
- From these numbers, we calculate
  - Program vocabulary: n = n1+n2
  - Program length: N = N1+N2

Halstead Complexity Metric, cont.

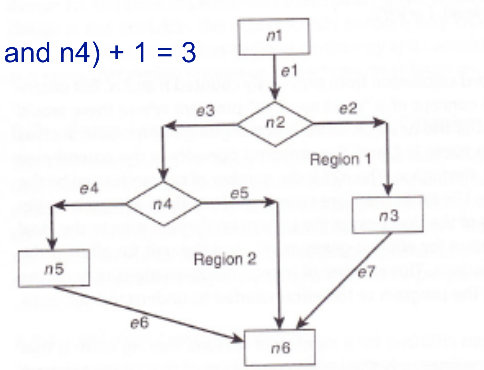
- Three more measurements
  - Volume: V = N * (Log2 n)
  - Difficulty: D = n1/2 * N2/n2
    The difficulty to write or understand the program
  - Effort: E = D * V
    A measure of actual coding time.
- Criticisms:
  - These metrics really measure only the lexical complexity of the source program and not the structure or the logic.
  - Therefore not useful for analyzing design characteristics.

McCabe’s Cyclomatic Complexity

- Basic idea: program quality is directly related to the complexity of the control flow (branching)
- Computed from a control flow diagram
  - Cyclomatic complexity = E - N + 2p
  - E = number of edges of the graph
  - N = number of nodes of the graph
  - p = number of connected components (usually 1)
- Alternate computations:
  - number of binary decision + 1
  - number of closed regions +1

McCabe’s Cyclomatic Complexity example

- Using the different computations:
  - 7 edges - 6 nodes + 2*1 = 3
  - 2 regions + 1 = 3
  - 2 binary decisions (n2 and n4) + 1 = 3



McCabe's Cyclomatic Complexity

- What does the number mean?
- It's the maximum number of linearly independent paths through the flow diagram
  - used to determine the number of test cases needed to cover each path through the system
- The higher the number, the more risk exists (and more testing is needed)
  - 1-10 is considered low risk
  - greater than 50 is considered high risk

Good Design attributes

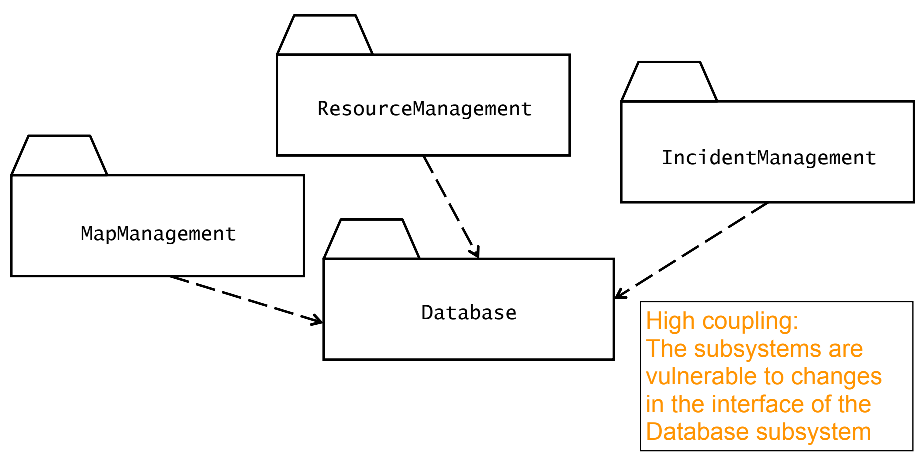
- Main goal: Simplicity
  - Easy to understand
  - Easy to change
  - Easy to reuse
  - Easy to test
  - Easy to code
- How do we measure simplicity of a design?
  - Coupling (goal: loose coupling)
  - Cohesion (goal: strong cohesion)

Coupling

- **Coupling** is an attribute that specifies the number of dependencies <u>between</u> two software units.
  - It measures the dependencies between two subsystems.
- If two subsystems are <u>loosely</u> coupled, they are relatively independent
  - Modifications to one of the subsystems will have little impact on the other.
- If two subsystems are <u>strongly</u> coupled, modifications to one subsystem is likely to have impact on the other.
- **Goal**: subsystems should be as loosely coupled as is reasonable.
  - In chapter 6 we called this "Independence"

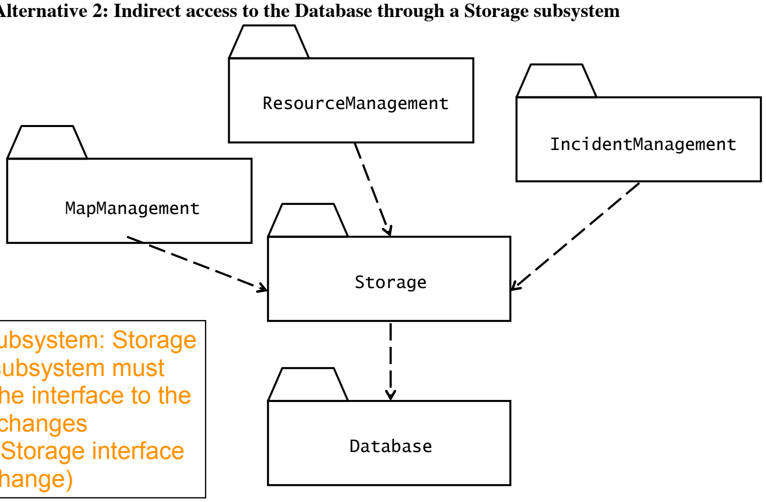
Example: reducing the coupling of subsystems

Alternative 1: Direct access to the Database subsystem



Example: reducing the coupling of subsystems

Alternative 2: Indirect access to the Database through a Storage subsystem

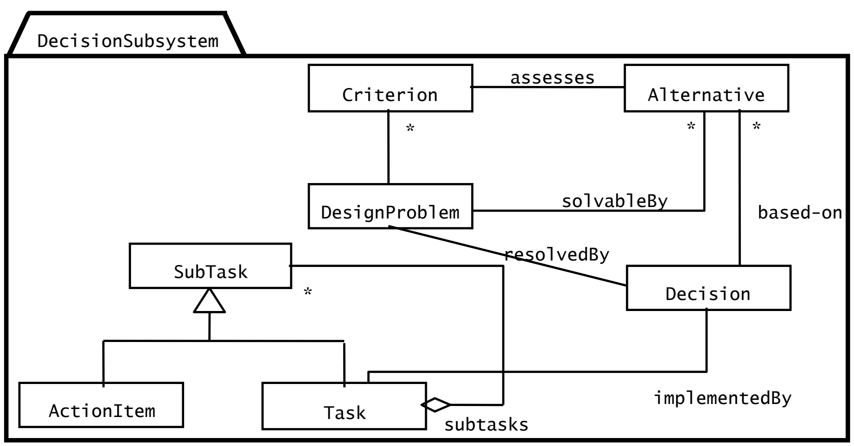


Added a subsystem: Storage
Only one subsystem must change if the interface to the Database changes
(Assumes Storage interface does not change)

Cohesion

- **Cohesion** is the number of dependencies <u>within</u> a subsystem.
  - It measures the dependencies among classes within a subsystem.
- If a subsystem contains many objects that are related to each other and perform similar tasks, its cohesion is high.
- If a subsystem contains a number of unrelated objects, its cohesion is low.
- **Goal**: decompose system so that it leads to subsystems with high cohesion.
  - These subsystems are more likely to be reusable

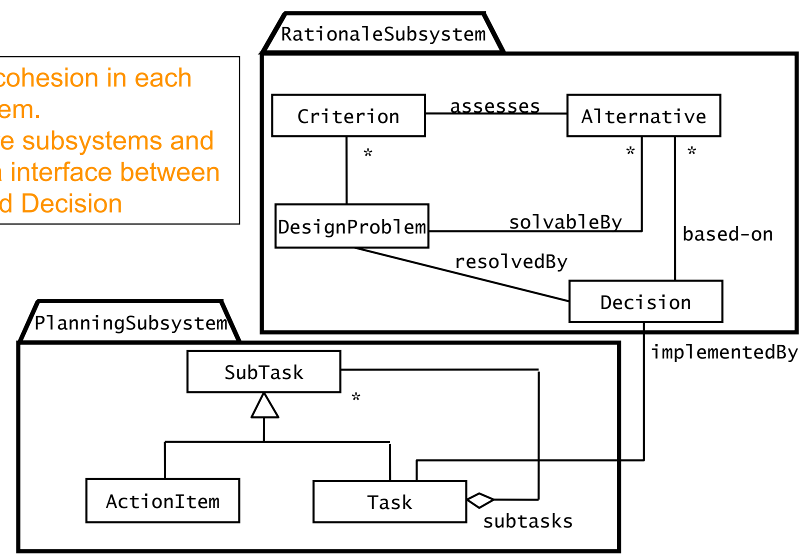
Example: Decision tracking system



Low Cohesion:
Criterion, Alternative, and DesignProblem have No relationships with SubTask, ActionItem, and Task

Alternative decomposition: Decision tracking system

Higher cohesion in each subsystem.
But more subsystems and an extra interface between Task and Decision



Law of Demeter

- Good guideline for object-oriented design
- An object should send messages to only the following
  - the object itself
  - the objects attributes (instance variables)
  - the parameters of member functions of the object
  - Any object created by this object
  - Any object returned from a call to one of this objects member function
  - Any object in any collection that is in one of the preceding categories.
- “Only talk to your immediate neighbors”
  “Don’t talk to strangers”