

MODULE 4: SOFTWARE ENGINEERING AND DATABASE SYSTEMS

PART I: SOFTWARE ENGINEERING (Chapter 7.1-7.5)

7.1 THE SOFTWARE ENGINEERING DISCIPLINE

Definition: Software engineering is the branch of computer science that seeks principles to guide the development of large, complex software systems.

Key Challenge: Problems in large systems are NOT merely enlarged versions of problems in small programs. They involve:

- Multiple personnel over extended time periods
- Changing requirements during development
- Personnel changes mid-project
- Project management and resource estimation

Differences from Traditional Engineering:

Traditional Engineering	Software Engineering
Can use prefabricated off-the-shelf components	Historically built from scratch; limited reusable components
Established metrics for measuring properties (e.g., mean time between failures)	Lack of quantitative metrics for complexity, quality
Founded on established science (physics)	Still searching for theoretical foundations
Wear and tear is measurable	Software doesn't wear out

Why Software Engineering is Challenging:

1. **Lack of Reusable Components:** Previously designed components were domain-specific; complex systems historically built from scratch
2. **Lack of Metrics:** Cannot easily measure complexity, quality, or project cost in advance
 - Software doesn't wear out → mean time between failures not applicable
 - No standard unit of "complexity"
3. **Software Failures Have Severe Consequences:**
 - Rising moon interpreted as nuclear attack
 - \$5 million loss by Bank of New York in one day

- Loss of space probes
- Radiation overdoses causing death and paralysis
- Large-scale telephone disruptions

Two Levels of Research:

Practitioners

Develop techniques for immediate application

Produce methodologies (some become obsolete)

Focus on practical solutions

Theoreticians

Search for underlying principles and theories

Progress is slow but foundational

Focus on theoretical understanding

Computer-Aided Software Engineering (CASE):

- Application of computer technology to software development process
- **CASE Tools:** Computerized systems that assist software development

Types of CASE Tools:

Tool Type

Purpose

Project Planning Systems

Cost estimation, project scheduling, personnel allocation

Project Management Systems

Monitoring development progress

Documentation Tools

Writing and organizing documentation

Prototyping and Simulation Systems

Developing prototypes

Interface Design Systems

GUI development

Programming Systems

Writing and debugging programs

Integrated Development
Environments (IDEs)

Combine editors, compilers, debugging tools into single package

Smartphone Development Example:

IDEs for smartphones provide:

- Programming tools for writing and debugging
- Simulators with graphical displays
- Ability to preview software performance on virtual phones

Professional Organizations:

- **ISO** (International Organization for Standardization)
- **ACM** (Association for Computing Machinery)
- **IEEE** (Institute of Electrical and Electronics Engineers)

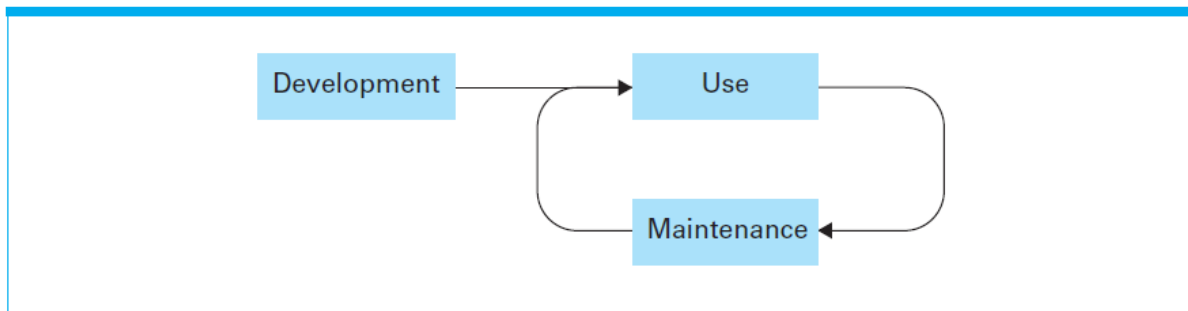
Their Roles:

- Adopting codes of professional conduct and ethics
- Establishing standards for measuring software quality
- Providing guidelines for organizational improvement

7.2 THE SOFTWARE LIFE CYCLE

Definition: The most fundamental concept in software engineering. Represents that once software is developed, it enters a cycle of being used and maintained.

Figure 7.1 The software life cycle



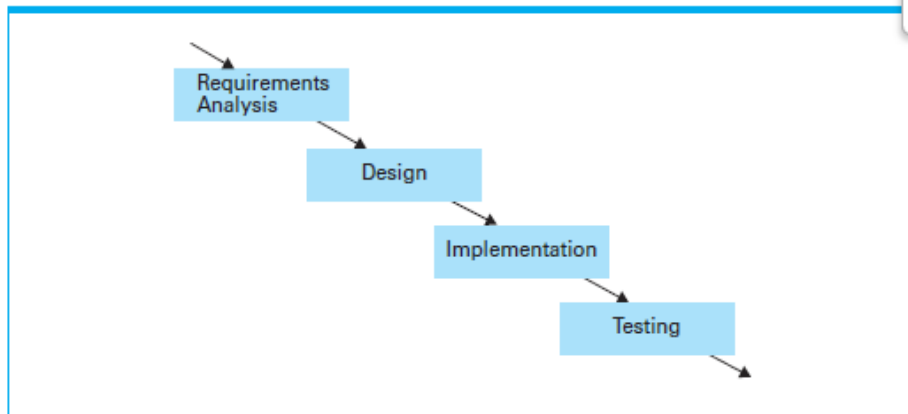
- Software maintenance = correcting errors, updating for changing requirements
- Often requires understanding someone else's code
- Many systems discarded because modification is harder than rebuilding

The 80/20 Rule: A little effort during development pays enormous dividends during maintenance

Traditional Development Phase

Four Major Steps:

Figure 7.2 The traditional development phase of the software life cycle



1. REQUIREMENTS ANALYSIS

Goal: Specify what services the system will provide, identify constraints, define outside world interaction

Stakeholders: Future users, legal/financial interests, etc.

Two Scenarios:

COTS Software (Commercial Off-the-Shelf)

Custom Development

Mass market product

Client hires developer

Sold in retail stores/downloaded via Internet

Feasibility study by client

Market study by developer

Requirements analysis with client

User is imprecisely defined

User is specific entity

Process:

1. Compile and analyze user needs
2. Negotiate trade-offs between wants, needs, costs, feasibility
3. Develop **Software Requirements Specification (SRS)**

Significance of SRS:

- Written agreement between all parties
- Guides software development
- Resolves disputes
- IEEE and DoD have adopted standards for SRS composition

Major Challenge: Changing requirements

- Unlike buildings (can't change foundation mid-construction)
- Companies expand scope mid-project
- Technology advances during development
- **Solution:** Frequent communication with stakeholders

2. DESIGN

Goal: Create a plan for constructing the proposed system

Simple View:

- Requirements analysis = WHAT the system will do
- Design = HOW the system will do it

Reality: There's HOW in requirements analysis and WHAT in design

Output: Detailed description of software system's structure that can be converted into programs

Analogy to Architecture:

- Building design = blueprints at various levels of detail
- Software design = diagramming and modeling methodologies

Challenge: Software design methodologies are not as stable as architectural ones

- Dynamic field with constant research
- Shifting terrain of approaches

3. IMPLEMENTATION

Goal: Actual writing of programs, creation of data files, development of databases

Roles:

System Analyst

Programmer

Involved with entire development process

Primarily involved with implementation step

Emphasis on requirements analysis and design

Writes programs that implement the design

Broader scope

Narrower scope

Reality: Terminology is blurred

- Many "software analysts" are essentially programmers
 - Many "senior programmers" are actually analysts
 - Steps in development are often intermingled
-

4. TESTING

Traditional View: Debugging programs + confirming final product matches requirements specification

Modern View: Much broader

- Every intermediate step must be "tested" for accuracy
- Testing is one segment in overall **quality assurance**
- Quality assurance permeates entire software life cycle

Current Thinking: Testing should be incorporated into other steps:

- "Requirements analysis and confirmation"
- "Design and validation"
- "Implementation and testing"

Reality: Even with modern QA techniques, large software systems still contain errors after significant testing

- Some errors go undetected for life of system
- Others cause major malfunctions
- Goal of software engineering = eliminate these errors

7.3 SOFTWARE ENGINEERING METHODOLOGIES

Waterfall Model (Traditional)

Characteristics:

- Strictly sequential: Requirements → Design → Implementation → Testing
- Complete requirements specification before design begins
- Complete design before implementation begins
- Flow is in ONE direction only

Problem: Contradicts creative problem-solving process (trial-and-error)

Incremental Model

Characteristics:

- System constructed in increments
- First increment = simplified version with limited functionality
- Tested and evaluated by user
- Features added incrementally until system complete

Example - Hospital Patient Records System:

1. First increment: View patient records from small sample
2. Next increment: Add and update records
3. Continue until system complete

Iterative Model

Characteristics:

- Refining each version (not just extending)
- Similar to incremental, sometimes equated, but distinct
- Incremental model involves iterative process
- Iterative model may incrementally add features

Rational Unified Process (RUP):

- Created by Rational Software Corporation (now IBM)
- Software development paradigm redefining development phase steps
- Provides guidelines for performing those steps

- Guidelines marketed with CASE tools
- **Unified Process:** Nonproprietary version

Prototyping

Evolutionary Prototyping:

- Incomplete versions (prototypes) evolve into complete, final system
- Used in incremental model

Throwaway Prototyping:

- Prototypes discarded in favor of fresh implementation
- **Rapid Prototyping:** Simple example quickly constructed early in development
 - May consist of only screen images
 - Shows system interaction and capabilities
 - NOT working product
 - Tool for clarifying communication
 - Advantages during requirements analysis and sales presentations

Open-Source Development

Characteristics:

- Less formal incarnation of incremental/iterative ideas
- Used by computer enthusiasts/hobbyists
- Prominent example: Linux operating system (Linus Torvalds)

Process:

1. Single author writes initial version (for personal need)
2. Posts source code and documentation on Internet
3. Others download and use without charge
4. Users modify/enhance software, correct errors
5. Changes reported to original author
6. Author incorporates changes into posted version
7. Extended version available for further modifications

Result: Software can evolve through several extensions in a single week

Agile Methods

Characteristics:

- Most pronounced shift from waterfall model
- Early and quick implementation on incremental basis
- Responsiveness to changing requirements
- Reduced emphasis on rigorous requirements analysis and design

Extreme Programming (XP):

- Team of less than a dozen individuals
- Communal workspace
- Freely share ideas and assist each other
- Daily cycles of informal requirements analysis, designing, implementing, testing
- New expanded versions appear regularly
- Stakeholders evaluate and point toward further increments

Key Contrast:

Waterfall Model

Agile Methods (XP)

Rigid, sequential

Flexible, iterative

Managers/programmers in individual offices

Team in communal workspace

Well-defined portions of development task

Daily cycles of all activities

Complete specifications before proceeding

Responsive to changing requirements

Conclusion: Research in the field is ongoing; progress is being made, but much work remains

7.4 MODULARITY

Definition: Division of software into manageable units called **modules**, each dealing with only part of software's overall responsibility

Purpose: Without modularity, understanding large software systems would be nearly impossible

Forms of Modules:

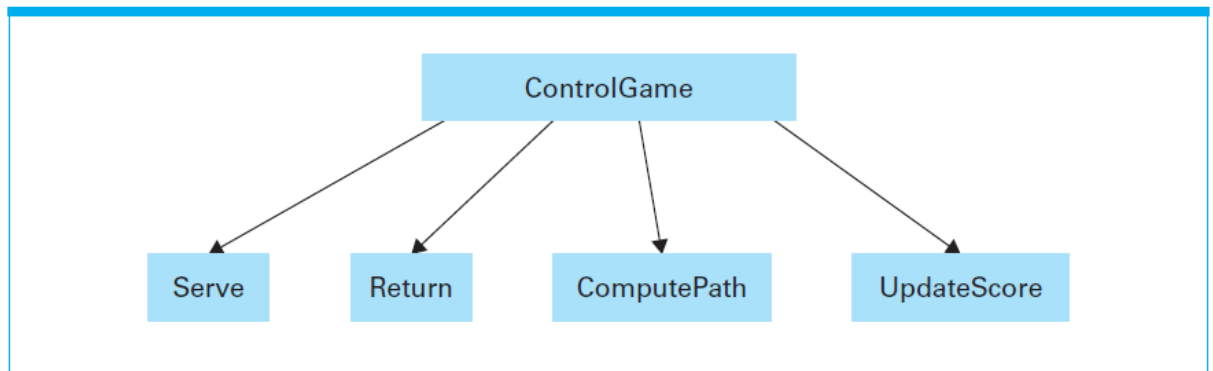
- Imperative paradigm: **Functions**
- Object-oriented paradigm: **Objects**

Modular Implementation - Design Approaches

Imperative Paradigm Design:

- Consider actions that must take place
- Identify functions needed
- Represent dependencies between functions
- **Structure Chart:** Rectangles represent functions, arrows represent function calls

Figure 7.3 A simple structure chart



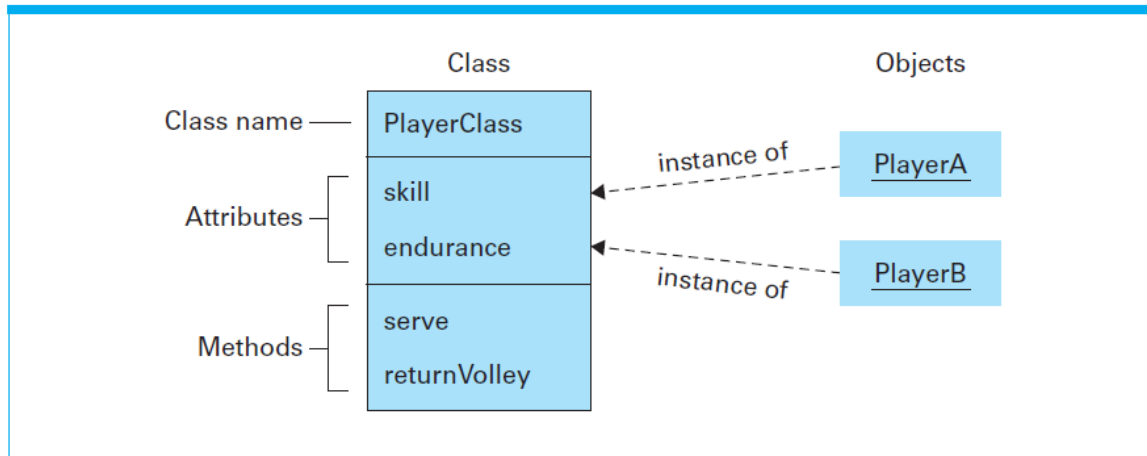
- Chart shows functions and dependencies
- Does NOT show HOW each function performs its task
- Identifies WHAT functions exist and HOW they depend on each other

Object-Oriented Paradigm Design:

- Identify objects in the system
- Understand how they interact
- Objects with same functionality but different characteristics → instances of same class
- **Class:** Template defining methods (functions) and attributes (instance variables)

text

Figure 7.4 The structure of PlayerClass and its instances



Sequence of Events Diagram:

- Objects represented as rectangles
- Horizontal arrows = method calls
- Events chronological from top to bottom

Coupling

Definition: Intermodule coupling = linkage between modules

Goal: Minimize coupling → maximize module independence

Metric: Measure intermodule coupling to estimate software maintenance expense

Types of Coupling:

1. Control Coupling:

- Occurs when one module passes control to another (function call)
- Structure chart arrows represent control coupling
- Sequence diagram arrows represent control coupling between objects

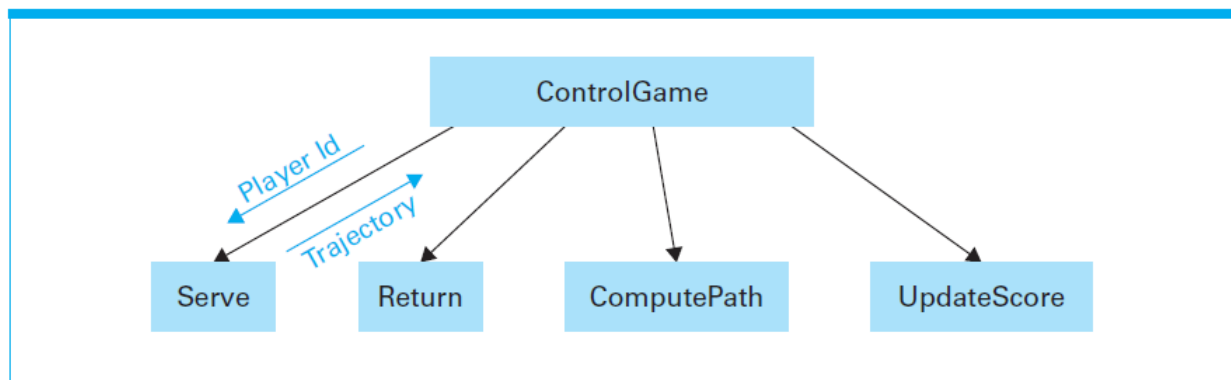
2. Data Coupling:

- Sharing data between modules
- Modifications to one module may affect another
- Modifications to data format affect both modules

Forms of Data Coupling in Functions:

Form	Description	Representation
Explicit	Passing data as parameters	Labeled arrows on structure chart
Implicit	Global data accessible to all modules	No explicit representation

Figure 7.6 A structure chart including data coupling



Global Data Problem:

- Automatically available to all modules
- Person modifying module may not identify all interactions
- Degrades module's usefulness as abstract tool

Object-Oriented Advantage:

- Methods within an object manipulate object's internal data
- No need to pass that information to other objects
- Interobject data coupling is minimized

Cohesion

Definition: Internal binding within a module; degree of relatedness of module's internal parts

Goal: Maximize cohesion

Importance: If changes are needed, variety of activities within module confuses the process

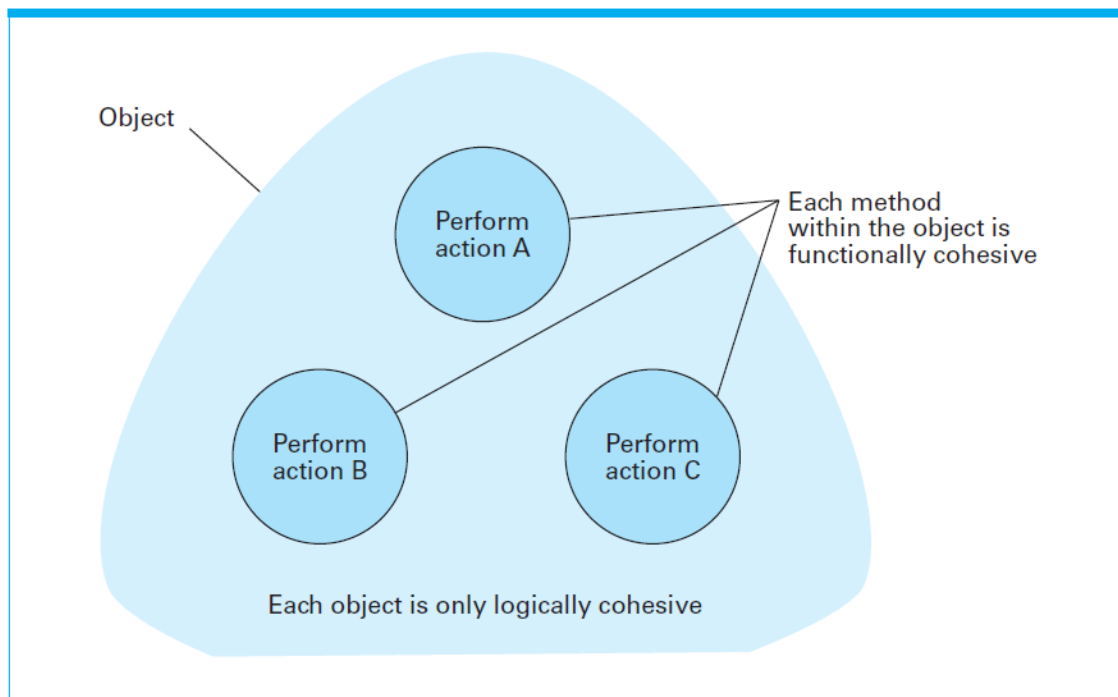
Types of Cohesion:

Type	Description	Example
Logical Cohesion	Weak form; elements perform logically similar activities	Single module handling all system communication (data input AND result reporting)
Functional Cohesion	Strong form; all parts focus on single activity	Module dedicated only to computing trajectory

In Object-Oriented Design:

- Entire objects are usually **logically cohesive** (methods perform loosely related activities)
- Each method within an object should be **functionally cohesive** (performs single task)

Figure 7.7 Logical and functional cohesion within an object



Information Hiding

Definition: Restriction of information to specific portion of software system

- **Information** = broad sense: data, data structures, encoding systems, module composition, procedural logic

Purpose:

- Keep module actions from having unnecessary dependencies on other modules
- Prevent errors in other modules from compromising module validity
- Prevent misguided maintenance efforts from causing problems

Two Incarnations:

Design Goal

Modules designed so others don't need internal access

Achieved through maximizing cohesion, minimizing coupling

Implementation Goal

Modules implemented to reinforce boundaries

Achieved through local variables, encapsulation, well-defined control structures

Relation to Abstraction:

- Information hiding = sealing the abstract tool
- Protects users from dangers inside
- Protects interior from intrusion from users

Components

Problem: Lack of prefabricated "off-the-shelf" building blocks for software construction

Object-Oriented Solution:

- Objects form complete, self-contained units
- Clearly defined interfaces with environments
- Once designed, can be reused in any program requiring that service
- **Inheritance:** Refines prefabricated object definitions for customization

Predefined Template Collections:

Language

Collection

C++

C++ Standard Template Library

Language	Collection
Java	Java Application Programmer Interface (API)
C#	.NET Framework Class Library

Component Architecture (Component-Based Software Engineering):

- **Component:** Reusable unit of software (collection of one or more objects)
- **Component Assembler:** Replaces traditional programmer role
- Constructs software systems from prefabricated components
- Components displayed as icons in graphical interface
- Methodology: Select components → connect them → minimal customization

Component Property: Well-designed component can be extended without internal modifications

Smartphone Component Architecture Example:

- Due to resource constraints, applications are sets of collaborating components
- Each display screen = separate component
- Service components: store/access memory card information, continuous functions (music), Internet access
- Components started/stopped individually as needed
- Application appears as seamless series of displays and actions

Integration Benefits:

- Facebook on smartphone may use contacts application components
- Adds Facebook friends as contacts
- Telephony application accesses contacts components to lookup caller
- Friend's picture displayed on screen during incoming call

7.5 TOOLS OF THE TRADE

Some Old Friends (From Imperative Paradigm)

Dataflow Diagrams:

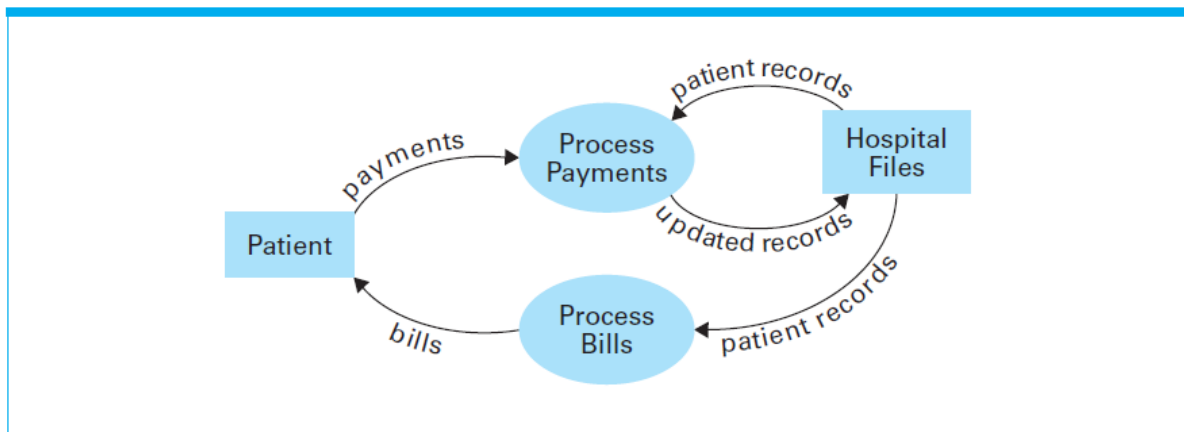
- Represent information gained from dataflow studies
- Study how data moves through system

- Identify points where data formats alter or data paths merge/split
- These locations = processing occurs → identifies functions

Symbols in Dataflow Diagrams:

Symbol	Meaning
Arrows	Data paths
Ovals	Points where data manipulation occurs
Rectangles	Data sources and stores

Figure 7.8 A simple dataflow diagram



Benefits:

- Identify procedures during design stage
- Understand proposed system during analysis stage
- Improve communication between clients and software engineers

Data Dictionary:

- Central repository of information about data items in software system

Information Stored:

Category	Details
Identifier	Name used to reference item

Category	Details
Valid entries	Numeric/ alphabetic? Range of values?
Storage location	Which file/ database?
Reference points	Which modules require this item?

Goals:

1. Improve communication between stakeholders and software engineers
2. Establish uniformity throughout system
3. Surface redundancies and contradictions
 - Example: "PartNumber" in inventory = "PartId" in sales
 - Example: "Name" in personnel (employee) vs. "Name" in inventory (part)

Unified Modeling Language (UML)

Purpose: Collection of tools developed with object-oriented paradigm in mind

1. USE CASE DIAGRAM

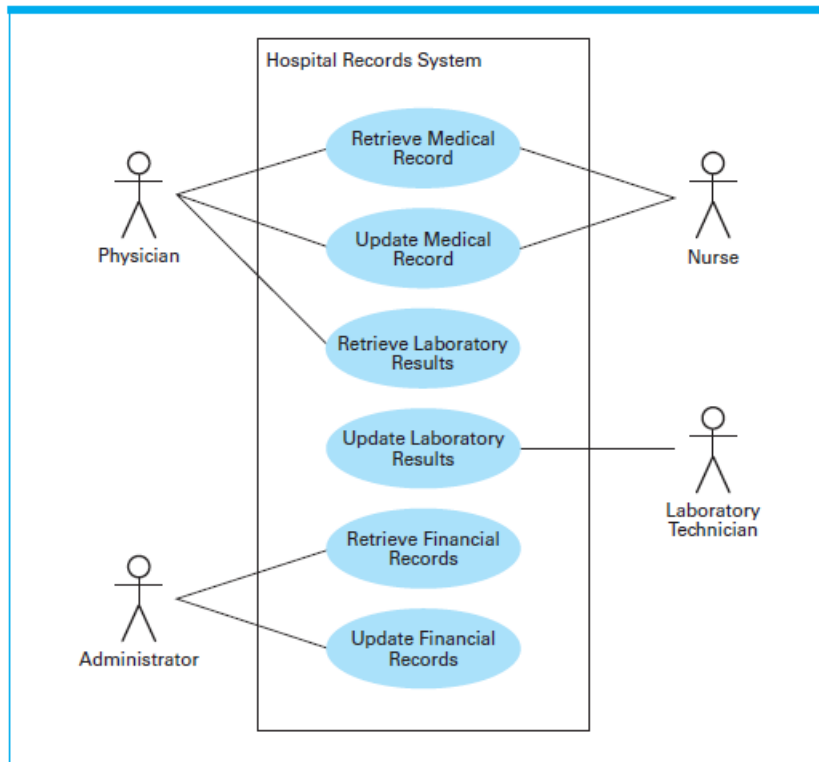
Purpose: Captures image of proposed system from user's point of view

- Useful regardless of underlying paradigm

Components:

Component	Representation	Meaning
System	Large rectangle	Proposed system
Use Cases	Ovals inside rectangle	Interactions between system and users
Actors	Stick figures	Users of system (may not be persons)

Figure 7.9 A simple use case diagram



2. CLASS DIAGRAM

Purpose: Represent structure of classes and relationships between classes (called **associations** in UML)

Notation:

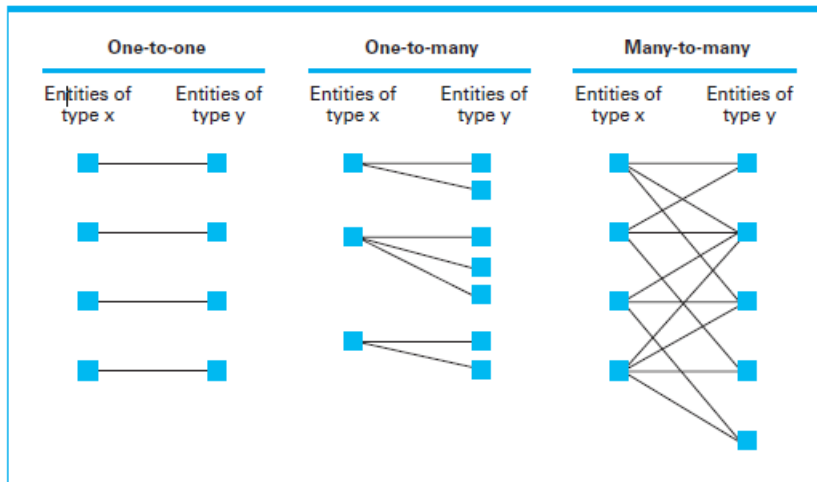
- Classes = rectangles
- Associations = lines
- Labels may have arrowhead indicating direction to read

Multiplicity Indicators:

Notation	Meaning
1	Exactly one
0..1	Zero or one
*	Arbitrary nonnegative number (many)

Types of Relationships:

Figure 7.11 One-to-one, one-to-many, and many-to-many relationships between entities of types X and Y



Examples:

Relationship Type	Example
One-to-one	Patient ↔ Private Room
One-to-many	Physician ↔ Patients (primary care)
Many-to-many	Physician ↔ Patients (including consulting)

3. GENERALIZATION (INHERITANCE)

Notation: Arrows with hollow arrowheads

Caution: Inheritance is NOT appropriate for all cases of generalization

- Introduces strong degree of coupling between classes
- Changes in parent class automatically reflected in child classes
- Minor modifications during maintenance can lead to unforeseen consequences

Example - Recreation Facility:

- Initially: RecreationMember class inherited from Employee class
- Later: Facility opened to dependents and retirees

- Embedded coupling between Employee and RecreationMember must be severed

Rule: Inheritance should be restricted to cases where generalization is IMMUTABLE

4. SEQUENCE DIAGRAM

Purpose: Express dynamic features; depict communication between individuals

Components:

- **Life line:** Rectangle + dashed line extending downward
- **Arrows:** Communication between life lines; labeled with action requested
- Read from top to bottom (chronological)
- **Unlabeled arrow:** Return of control

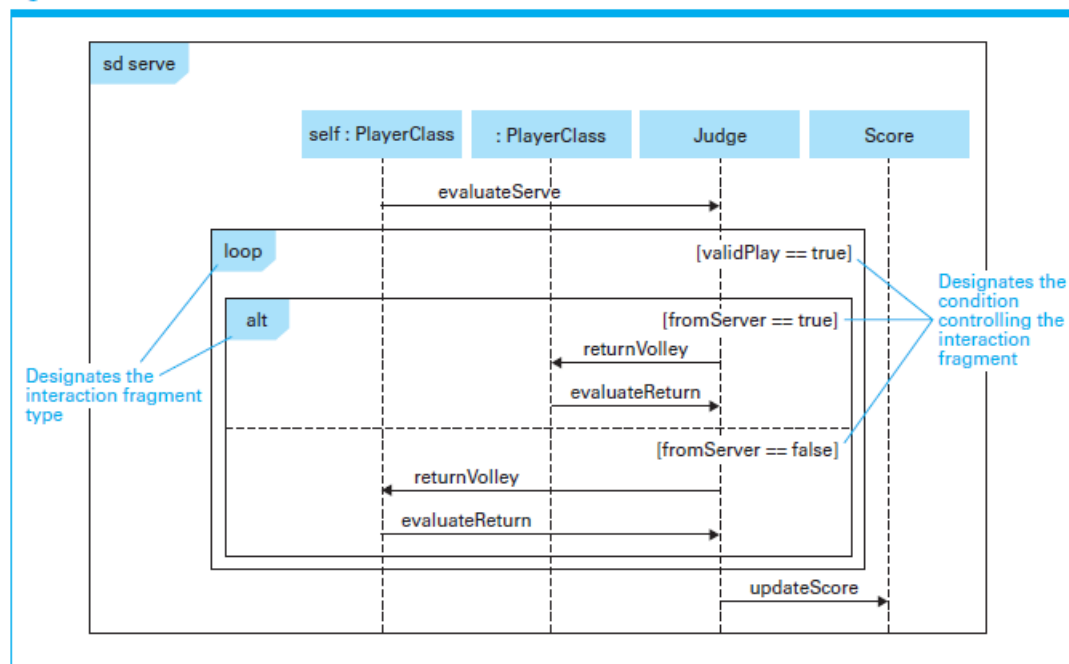
Frame: Entire sequence diagram enclosed in rectangle

- Upper left corner: pentagon containing "sd" + identifier

Interaction Fragments:

- Represent alternative sequences within one diagram
- **loop:** While structure
- **alt:** If-else structure

Figure 7.13 A sequence diagram depicting a generic volley



5. CRC CARDS (Class-Responsibility-Collaboration)

Purpose: Validate object-oriented designs

Methodology:

- Software designer produces card for each object in proposed system
- Cards used to represent objects in simulation
- "Theatrical" experiment: each team member holds card, plays role of object

Design Patterns

Definition: Predeveloped model for solving recurring problem in software design

Origin: Christopher Alexander's research in traditional architecture

Goal: NOT merely to find solutions, but HIGH-QUALITY solutions that provide flexibility later in software life cycle

Examples:

Pattern	Problem Solved	Solution
Adapter Pattern	Prefabricated module has needed functionality but incompatible interface	"Wrap" module inside another that translates between original interface and outside world
Decorator Pattern	System performs different combinations of same activities depending on situation	Standardized way of implementing such systems leading to manageable solution

Impact:

- Reflected in library of tools in today's software development packages
- Java programming environments
- .NET Framework
- Many templates are essentially design pattern skeletons

PART II: DATABASE SYSTEMS (Chapter 9.1-9.2)

9.1 DATABASE FUNDAMENTALS

Definition: A database is a collection of data that is multidimensional in the sense that internal links between its entries make information accessible from a variety of perspectives.

Contrast with Traditional File System (Flat File):

- Flat file = one-dimensional storage system
- Presents information from single point of view
- Example: List of compositions arranged by composer only

Database Example:

- All works by a single composer
- All composers who wrote a particular type of music
- Composers who wrote variations of another composer's work

The Significance of Database Systems

Historical Problem - Separate Systems:

- Payroll → payroll file
- Personnel → employee records
- Inventory → inventory file
- Result: Information duplicated throughout company
- Related items stored in separate systems

Solution - Integrated Database:

- Same sales data can produce:
 - Restocking orders
 - Market trend reports
 - Targeted advertisements
 - Bonus checks for sales force

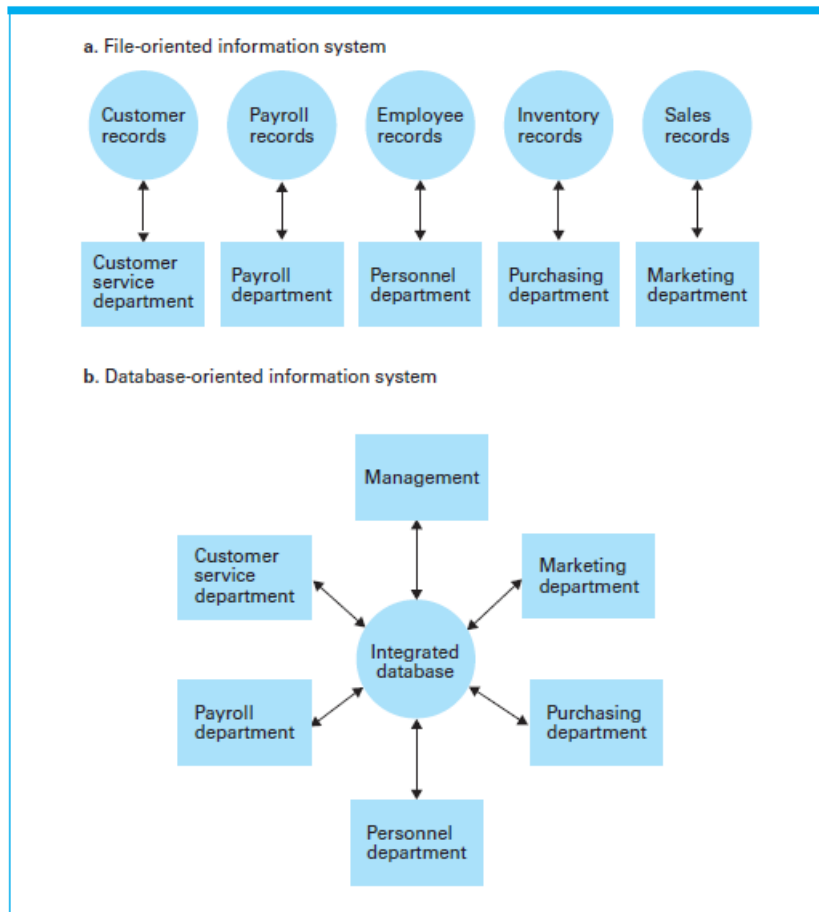
Database as Management Tool:

- Integrated information pools = valuable resource for decision-making
- Combined with data mining = important management tool
- Extract pertinent information from enormous data amounts

World Wide Web Connection:

- Database technology supports popular websites (Google, eBay, Amazon)
- Server interrogates database, organizes results as Web page
- Database becomes company's PRODUCT, not just record storage
- Internet + Database technology = major worldwide information source

Figure 9.1 A file versus a database organization



The Role of Schemas

Problem: Sensitive data accessed by unauthorized personnel

- Customer shouldn't access company's financial data
- Benefits employee shouldn't access inventory/sales records

Solution - Schemas and Subschemas:

Term	Definition
------	------------

Schema	Description of entire database structure used by DBMS to maintain the database
---------------	--

Term	Definition
Subschema	Description of only the portion of database pertinent to particular user's needs

University Database Example:

Schema contains:

- Student records: address, phone number, academic record
- Link between student and faculty adviser
- Faculty records: address, employment history
- Linkage system connects student info to faculty employment history

Registrar's Subschema:

- Faculty records do NOT include employment history
- Can find which faculty member advises which student
- CANNOT access additional faculty information

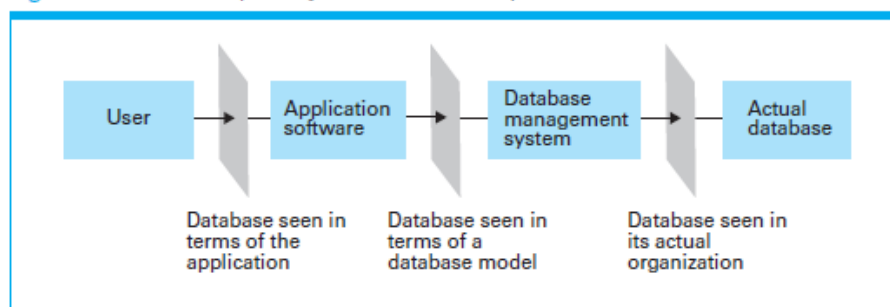
Payroll Department's Subschema:

- Provides employment history of each faculty member
- Does NOT include linkage between students and advisers
- Can modify salary but cannot obtain names of students advised

Database Management Systems (DBMS)

Two Major Software Layers:

Figure 9.2 The conceptual layers of a database implementation



Roles:

Layer	Responsibility
Application Software	Handles communication with user; determines requested action; uses DBMS as abstract tool
DBMS	Performs actual database manipulation; executes searches; adds/deletes data; enforces subschema restrictions

Benefits of Separation:

1. Abstract Tool Construction:

- Details of database storage isolated within DBMS
- Application software design simplified
- Example: DBMS handles distributed database concerns automatically

2. Access Control:

- DBMS performs all database access
- Can enforce subschema restrictions
- Application software remains within user's authorized bounds

3. Data Independence:

- Ability to change database organization without changing application software
- Example: Adding new field to employee records (health insurance choice)
- Without DBMS: Changes required to ALL application programs (payroll, mailing labels, etc.)
- With DBMS: Change only overall schema and relevant subschemas; other subschemas unchanged → no reprogramming

Database Models

Definition: Conceptual view of the database that hides internal complexities

Purpose: Allow user to imagine information arranged in more useful format

DBMS Function:

- Contains routines that translate commands stated in conceptual view
- Converts to actions required by actual data storage system

Two Models Covered:

Model	Conceptual View	Access Method
Relational Database Model	Collection of tables (rows and columns)	SELECT, PROJECT, JOIN operations
Object-Oriented Database Model	Collection of objects with links	Methods, messages, inheritance

How DBMS Supports Application Software:

- Application software written in general-purpose programming languages
- These languages lack instructions for database manipulation
- DBMS provides routines as prewritten subroutines
- Effectively extends language capabilities to support conceptual image

Ongoing Research Goal:

- Find models that allow easy conceptualization of complex data systems
- Lead to concise ways of expressing information requests
- Produce efficient database management systems

9.2 THE RELATIONAL MODEL

Definition: Portrays data as being stored in rectangular tables called **relations**

Terminology:

Term	Meaning	Analogy
Relation	Rectangular table	Spreadsheet
Tuple	Row in a relation	Information about one employee
Attribute	Column in a relation	Characteristic of entity (name, address, etc.)

Issues of Relational Design

Problem Example - Adding Job History:

Initial Approach: Extend EMPLOYEE relation to include job attributest

Problems with This Design:

1. Redundancy (Inefficiency):

- Relation has one tuple per job assignment, NOT per employee
- Employee who held multiple jobs → personal information repeated
- Job description repeated for each employee who held that position

2. Deletion Anomaly:

- Joe Baker only employee holding job D7
- If Joe leaves company and is deleted → lose information about job D7
- The only tuple containing "job D7 requires skill level K2" is Joe's record

3. Partial Tuple Temptation:

- Might try to erase only portion of tuple
- Leads to other complications
- Strong indication that database design can be improved

Root Cause: Combined more than one concept in a single relation

Three Distinct Concepts Mixed:

1. Information about employees (name, ID, address, SSN)
2. Information about jobs (job ID, title, department, skill code)
3. Information about relationship between employees and jobs (start date, termination date)

Solution - Three Relations

Figure 9.5 An employee database consisting of three relations

EMPLOYEE relation			
Empl Id	Name	Address	SSN
25X15	Joe E. Baker	33 Nowhere St.	111223333
34Y70	Cheryl H. Clark	563 Downtown Ave.	999009999
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555

JOB relation			
Job Id	Job Title	Skill Code	Dept
S25X	Secretary	T5	Personnel
S26Z	Secretary	T6	Accounting
F5	Floor manager	FM3	Sales
•	•	•	•
•	•	•	•
•	•	•	•

ASSIGNMENT relation			
Empl Id	Job Id	Start Date	Term Date
23Y34	S25X	3-1-1999	4-30-2010
34Y70	F5	10-1-2009	*
23Y34	S26Z	5-1-2010	*
•	•	•	•
•	•	•	•
•	•	•	•

Benefits:

- No redundancy (employee info stored once, job info stored once)
- No deletion anomaly (job info remains even if no current employee holds it)
- Information implicitly available by combining relations

Finding information across relations:

- Employee ID → find jobs held via ASSIGNMENT → find departments via JOB

Lossless vs. Lossy Decomposition

Not all decomposition is good:

Lossless Decomposition (Nonloss Decomposition):

- Dividing relation into smaller relations WITHOUT losing information
- Requires careful design consideration

Relational Operations

1. SELECT Operation:

- Extracts **rows** (tuples) from a relation

- Creates new relation containing selected tuples

Syntax:

$NEW \leftarrow SELECT \text{ from } EMPLOYEE \text{ where } EmpId = '34Y70'$

Result: New relation containing only the tuple(s) matching the condition

2. PROJECT Operation:

- Extracts **columns** (attributes) from a relation
- Creates new relation containing only specified columns

Syntax:

$MAIL \leftarrow PROJECT \text{ Name, Address from } EMPLOYEE$

Result: New relation with only the projected columns

3. JOIN Operation:

- Combines different relations into one relation
- New relation's attributes = attributes from original relations (prefixed by relation name)

Syntax:

$C \leftarrow JOIN \text{ A and B where } A.W = B.X$

Process:

- Concatenate tuples from two original relations
- Which tuples are joined? Those satisfying the condition

JOIN with other conditions:

- Can use $<$, $>$, $\leq$, $\geq$, etc.
- Example: $JOIN \text{ A and B where } A.W < B.X$

SQL (Structured Query Language)

Definition: Language that forms backbone of most relational database query systems

Example: MySQL used by many database servers on the Internet

Reasons for Popularity:

1. Standardized by ANSI
2. Originally developed and marketed by IBM
3. High level of exposure

Key Insight: SQL statement is essentially **declarative**

- Read as description of information desired
- NOT a sequence of activities to be performed
- Relieves programmers from developing algorithms for manipulating relations

SQL Query Structure:

SELECT attributes

FROM relations

WHERE condition;

Rough Translation:

1. Form JOIN of all relations listed in FROM clause
2. SELECT tuples satisfying WHERE condition
3. PROJECT attributes listed in SELECT clause

Note: Terminology is reversed! SELECT clause identifies attributes used in PROJECT operation

SQL Examples:

Operation	SQL Statement
-----------	---------------

PROJECT only	SELECT Name, Address FROM Employee;
---------------------	-------------------------------------

SELECT only	SELECT * FROM Employee WHERE Name = 'Cheryl H. Clark';
--------------------	--

Operation	SQL Statement
-----------	---------------

SELECT + PROJECT	SELECT Name, Address FROM Employee WHERE Name = 'Cheryl H. Clark';
-------------------------	--

JOIN + SELECT + PROJECT	SELECT Employee.Name, Assignment.StartDate FROM Employee, Assignment WHERE Employee.EmplId = Assignment.EmplId;
--------------------------------	---

SQL Data Modification Statements:

Operation	SQL Statement
-----------	---------------

INSERT	INSERT INTO Employee VALUES ('42Z12', 'Sue A. Burt', '33 Fair St.', '444661111');
---------------	---

DELETE	DELETE FROM Employee WHERE Name = 'G. Jerry Smith';
---------------	---

UPDATE	UPDATE Employee SET Address = '1812 Napoleon Ave.' WHERE Name = 'Joe E. Baker';
---------------	---

SUMMARY - KEY TERMS TO REMEMBER

Software Engineering

Term	Definition
Software Engineering	Branch of CS seeking principles for developing large, complex software systems
CASE Tools	Computerized systems that assist software development
Software Life Cycle	Development → Use → Maintenance cycle
Requirements Analysis	Specifying WHAT the system will do
Design	Creating plan for HOW the system will do it

Term	Definition
Implementation	Writing programs, creating data files, databases
Testing	Verifying accuracy of all development steps
Software Requirements Specification	Written agreement between all parties
Waterfall Model	Strictly sequential development process
Incremental Model	System built in increments with increasing functionality
Iterative Model	Refining each version of the system
Prototyping	Building incomplete versions for evaluation
Evolutionary Prototyping	Prototypes evolve into final system
Throwaway Prototyping	Prototypes discarded after learning
Rapid Prototyping	Quick construction of simple example
Open-Source Development	Publicly developed software with source code available
Agile Methods	Flexible, responsive, iterative development
Extreme Programming (XP)	Daily cycles of informal development by small team
Modularity	Dividing software into manageable units
Module	Manageable unit dealing with part of software's responsibility
Structure Chart	Diagram showing functions and their dependencies

Term	Definition
Coupling	Linkage between modules (minimize)
Control Coupling	Passing control from one module to another
Data Coupling	Sharing data between modules
Cohesion	Internal binding within module (maximize)
Logical Cohesion	Weak cohesion; elements perform logically similar activities
Functional Cohesion	Strong cohesion; all parts focus on single activity
Information Hiding	Restricting information to specific portion of system
Component	Reusable unit of software
Component Assembler	Constructs systems from prefabricated components
Dataflow Diagram	Represents data movement through system
Data Dictionary	Central repository of data item information
UML	Unified Modeling Language - tools for object-oriented design
Use Case Diagram	Shows system interactions from user's perspective
Class Diagram	Shows classes and their relationships
Association	Relationship between classes
Multiplicity	How many instances of one class relate to another
Generalization	Inheritance relationship (hollow arrowhead)

Term	Definition
Sequence Diagram	Shows communication between objects over time
Interaction Fragment	Alternative sequences (loop, alt)
CRC Card	Class-Responsibility-Collaboration card for design validation
Design Pattern	Predeveloped model for recurring design problem
Adapter Pattern	Wraps module with incompatible interface
Decorator Pattern	Standardized way to handle combinations of activities

Database Systems

Term	Definition
Database	Multidimensional data collection with internal links
Flat File	One-dimensional storage (single perspective)
Schema	Description of entire database structure
Subschema	Description of portion pertinent to specific user
DBMS	Database Management System - performs actual database manipulation
Data Independence	Ability to change database organization without changing application software
Database Model	Conceptual view hiding internal complexities
Relational Model	Data viewed as collection of tables
Relation	Rectangular table

Term	Definition
Tuple	Row in a relation
Attribute	Column in a relation
Lossless Decomposition	Dividing relation without losing information
SELECT	Operation extracting rows
PROJECT	Operation extracting columns
JOIN	Operation combining relations
SQL	Structured Query Language
SELECT clause	Identifies attributes for PROJECT (in SQL)
FROM clause	Identifies relations for JOIN
WHERE clause	Identifies condition for SELECT
INSERT	SQL statement to add tuples
DELETE	SQL statement to remove tuples
UPDATE	SQL statement to modify tuples
