SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

UNIT II

METHODOLOGY AND UML

- **Overview of methodologies:**
  - –OMT
  - –Booch methodology
  - –Jacobson methodology
  - –Unified Approach
- **UML:**
  - –Static and Dynamic Modelling
  - –UML diagrams.

- **Overview of methodologies:**
  - –OMT
  - –Booch methodology
  - –Jacobson methodology
  - –Unified Approach
- **UML:**
  - –Static and Dynamic Modelling
  - –UML diagrams.

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

What is Object Oriented methodologies?

- Is a set of models , methods and rules for developing systems
- Is important for analysis and design as like object oriented programming concepts

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

Models:-

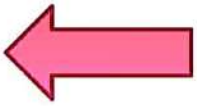
- ✓description about the current and proposed system
- ✓Used in all phase of software life cycle
- ✓Easy to understand

Methods:-

Different methods developed in 1980's – 1990's are listed below

Year	Created / Developed by	Description
1986	Booch	OO design concept BOOCH METHOD
1987	Sally & Steve	Recursive design concept
1989	Beck & Cunningham	CRC Card (Class – Responsibility – Collaboration card)
1990	Brock Weiner	Responsibility driven design
1991	Rumbaugh	OMT Object Modeling Technique
	Pela	Load light weight and prototype oriented approach
1994	Jacabson	Use case and OOSE(Object Oriented Software Engineering)

All methodologies are same and related to one another but with slight difference

- **Overview of methodologies:**
  - –OMT 
  - –Booch methodology
  - –Jacobson methodology
  - –Unified Approach
- **UML:**
  - –Static and Dynamic Modelling
  - –UML diagrams.

Rumbaugh – OMT

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Describes a method for analysis, design and implementation of a system using an OO techniques
- It is the fast approach for identifying the different object in a system
- details such as class, attributes, methods, inheritance and association can be expressed easily
- It consists of four phases – performed iteratively

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

1. Analysis
2. System design
3. Object design
4. Implementation

1. Analysis:- Results are object, dynamic and functional models
2. System design:-Results are a structure of basic architecture of the system along with the high-level strategy decision
3. Object design:- produce design document, consisting of detailed objects static, dynamic and functional models
4. Implementation:-produce reusable, extensible and robust code

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- OMT separates modelling into three different parts

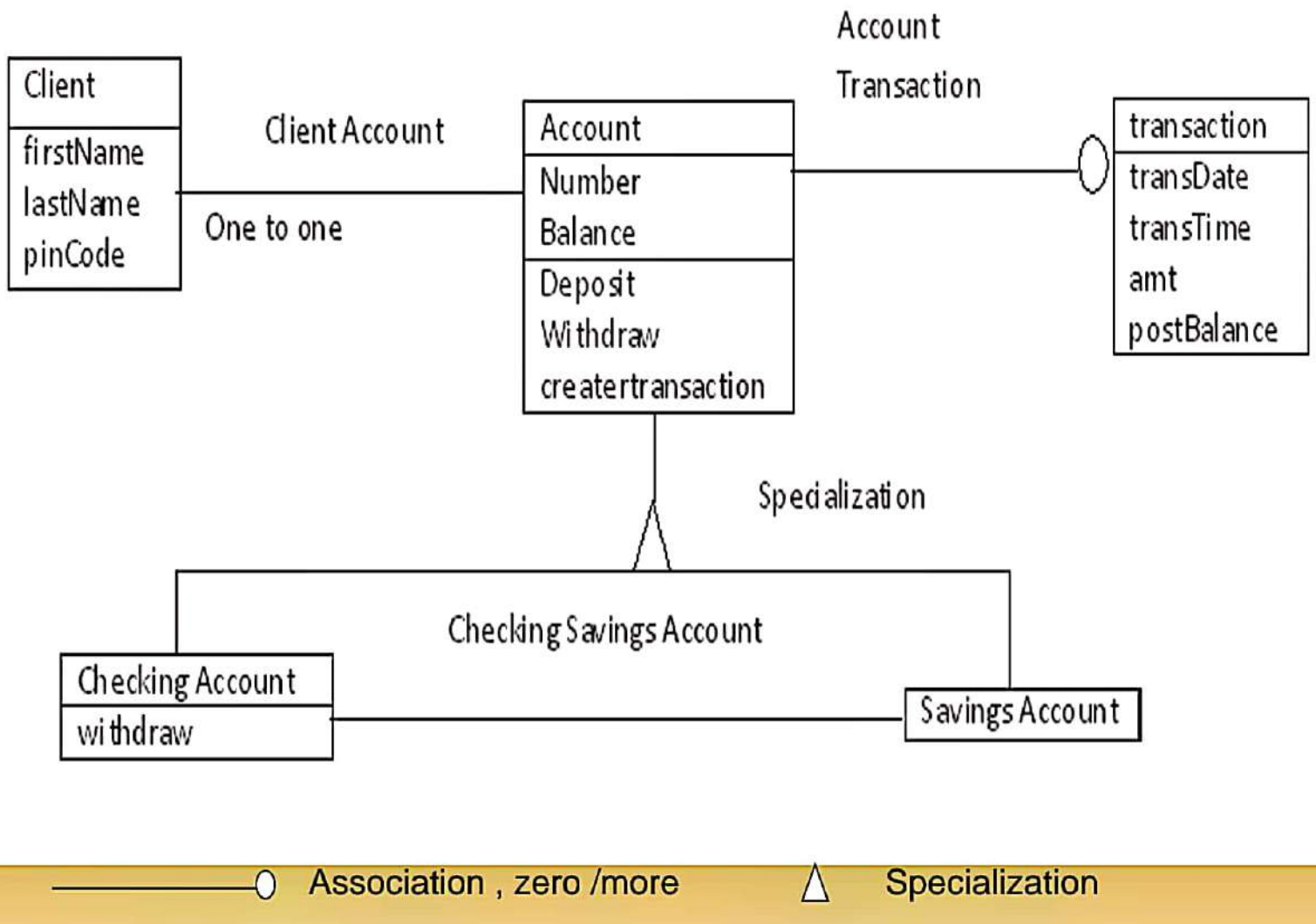
1. Object model
2. Dynamic model
3. Functional model

1. Object Model:- presented by object model and data dictionary, eg., different classes and its relations
2. Dynamic model:- presented by state diagram and event flow diagram eg., different events and dynamic objects
3. Functional model:- presented by dataflow diagram and constraints eg., different process descriptions and consumer-producer relationships

1. Object Model:-

- Describes the structure of object in a system, their identity, relationships to other objects, attributes and operations
- It is represented graphically by object diagram
- Object diagram contains classes connected by association lines
- Each represent a set of individual object
- Following diagram is an example for object diagram/ model

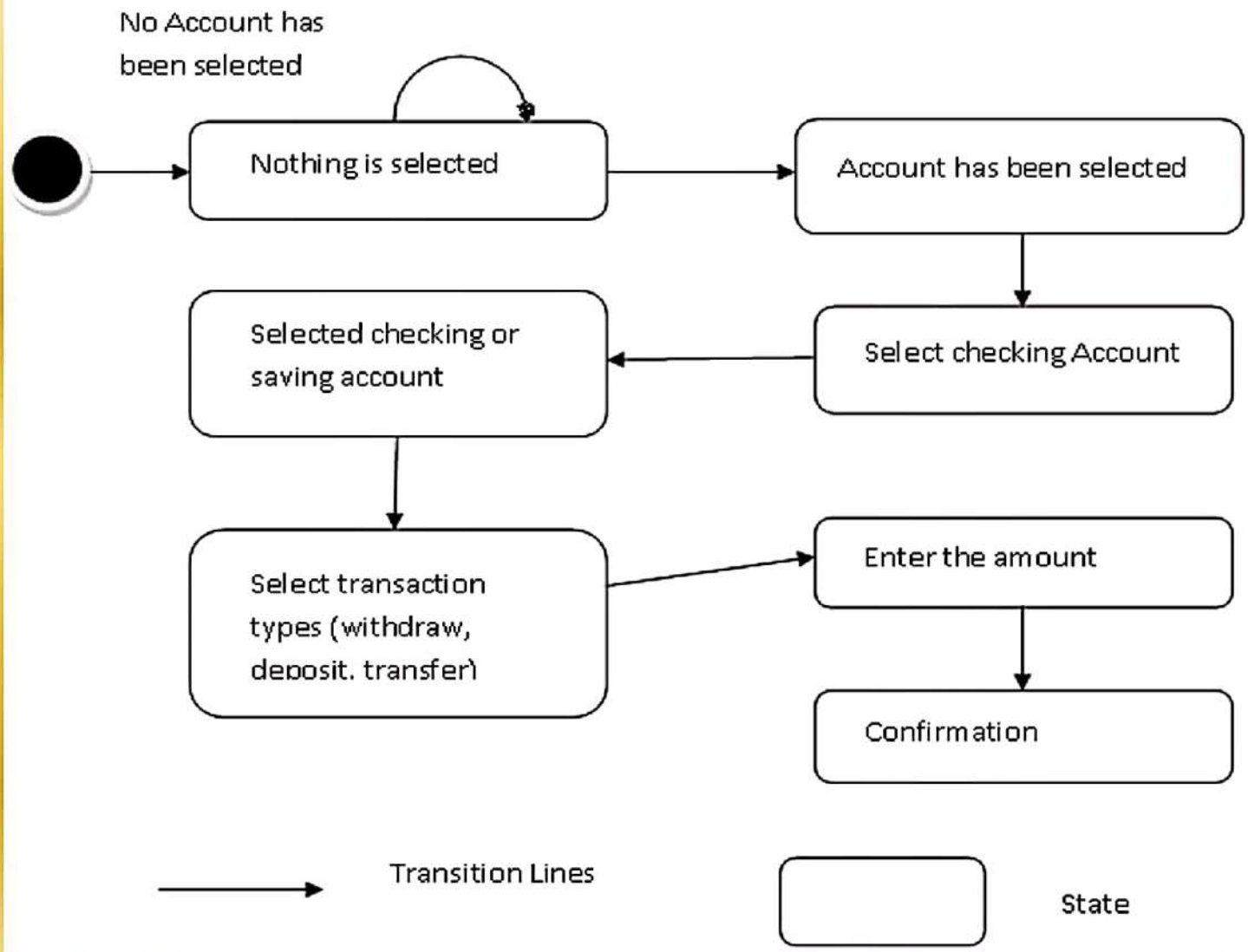
Establish relation among classes and set of links



Association , zero /more Specialization

2. Dynamic Model:-

- State the different states, transitions, action and event flow
- To represent these state transition diagram is used which is a collection of different states and events
- Each state receive one or more events, transitioned to next state
- Next state depends on the current state as well as the event.
- Following diagram is an example for state diagram/ dynamic model



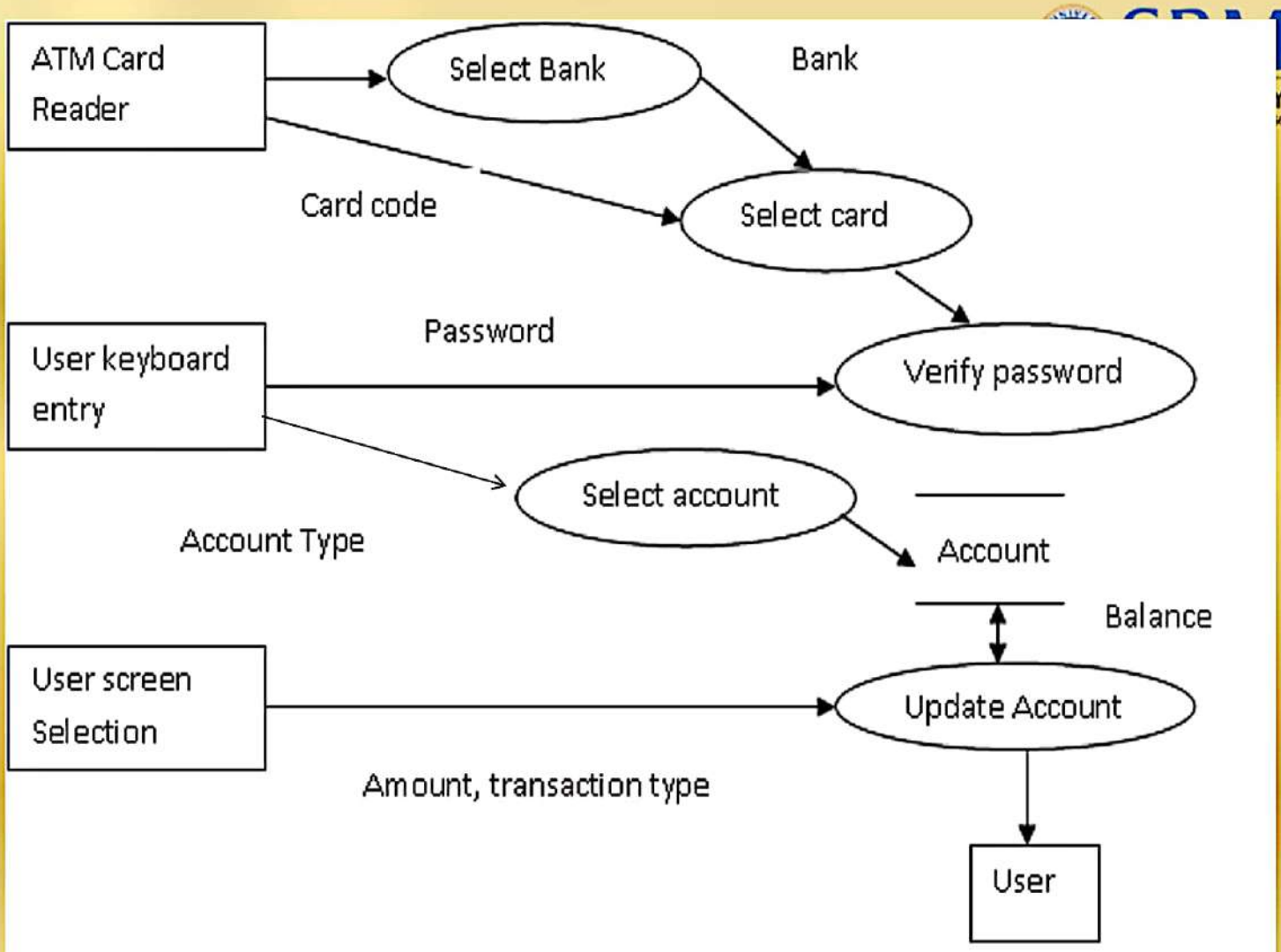
SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

3.Functional Model

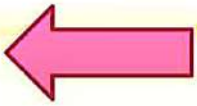
- Explain the different data flow between a process in a system by means of DFD (Data Flow Diagram)
- Give a simple description about process without focusing on the detail of system
- DFD mainly consists of four primary parts

Symbols	Name	Description/ functions
	Process	Any function being performed
	Data flow	Direction of data element movement
	Data store	Location where data are stored
	External entity	Is a source / destination of a data element

- Following diagram is an example for data flow diagram/ functional model



SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- **Overview of methodologies:**
  - –OMT
  - –Booch methodology 
  - –Jacobson methodology
  - –Unified Approach
- **UML:**
  - –Static and Dynamic Modelling
  - –UML diagrams.

The BOOCH Methodology:-

- It is a widely used OO method that helps to design the system using object paradigm
- Covers analysis and design phase of an OO System
- Lots of symbols and diagrams are in BOOCH methodology
- But designing only less symbols and diagram are used
- It consists of six diagram for designing

1. Class diagram
2. Object diagram
3. State Transition diagram
4. Module diagram
5. Process diagram
6. Interaction diagram

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- BOOCH Methodology prescribes 2 different process
  1. Macro Development Process
  2. Micro Development Process

1. Macro Development Process

- Serve as a controlling framework for the microprocessor and can take weeks or even months
- Primary concern is technical management of the system
- Consists of 5 different steps
  1. Conceptualization
  2. Analysis and development of the model
  3. Design or create the system architecture
  4. Evolution or implementation
  5. Maintenance

1. Conceptualization:-

core requirement of the system, prove the concept

2. Analysis and development of the model:-

use class diagram to describe the role and responsibility an object

Class diagram decide what classes exists and how ar

Object diagram decide about then different object collaboration

Module diagram where all classes and object to be declared

Process diagram determines which process to allocate

diagram, module diagram and process diagram

3. Evolution or implementation:-

refine the system by many iteration and produce a stream of software implementation

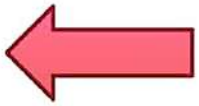
4. Maintenance:-

add new requirement and remove bugs

2. Micro Development Process

- Each macro development process have its own micro development process
- It is the description of day to day description
- It consists of following steps
  1. Identify classes and objects
  2. Identify classes and objects semantics
  3. Identify classes and objects relationships
  4. Identify classes and objects interface and implementation

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- **Overview of methodologies:**
  - –OMT
  - –Booch methodology
  - –Jacobson methodology 
  - –Unified Approach
- **UML:**
  - –Static and Dynamic Modelling
  - –UML diagrams.

Jacobson methodology

- It covers the entire life cycle and stress traceability between the different phases, both forward and backward, eg., OOSE, OOBE, Objectory(object factory for s/w development)
- Enables reuse of analysis and design, reduce development time than reuse of code
- At the heart of their methodologies is the usecase concept, which evolved objectory.

1.Use cases

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- are scenarios for understanding system requirements
- is an interaction between users & the system
- capture the goal of the user & the responsibility of the system to its users.
- in the requirements analysis, the use cases are described as one of the following,
  - ✓Nonformal text with no clear flow of events
  - ✓ Text, easy to read but with a clear flow of event to follow
  - ✓ Formal style using pseudo code.

- Use case description must contain,
  - ✓ How & when the use case begins and ends
  - ✓ Interaction between the use-case & its actors, including when the interaction occurs & what is exchanged
  - ✓ How & when the use-case will need data stored in the system or will store data in the system
  - ✓ Exceptions to the flow of events
  - ✓ How & when the concepts of problem domain are handled
- Every single use-case should one main flow of events.
- Use-case would be viewed as concrete or abstract.(is not complete and has no actors that initiate it but is used by another user case

2. Object Oriented Software Engineering: objectory (OOSE)

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Also called objectory is a method of OO development with the specific aim to fit the development of large, real-time systems.
- Development process, called use-case driver development, stresses that use-cases are involved in several phases of the development, including analysis, design, and validation & testing.
- OOSE- is a disciplinary process for the industrialized development of s/w, based on a use-case driven design.
- Jacobson objectory has been developed & applied to numerous application areas & embedded in the CASE tools system.

- Objectory is built around several different models.
  1. Use case model: defines outside (actor) & inside (use case) of the systems behaviour.
  2. Domain object model: object of real world are mapped into the domain obj model.
  3. Analysis object model: it presents how the source code should be carried out & written.
  4. Implementation model: rep implementation of the system.
  5. Test model: constitutes the test plans, specification & reports.
- Maintenance of each model is specified in its associated process.

3.Object oriented business engg. (OOBE):

- OOBE is obj modelling at the enterprise level.
  - analysis phase
    - Object Model
    - Requirement
    - Analysis
  - design & implementation phase
    - DBMS
    - Distribution of Process
  - testing phase
    - Unit Testing
    - Integration testing
    - System Testing

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

Patterns & Frameworks

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

Patterns

Patterns:

- Each pattern describes a problem which occurs over & over again in real time, & then describes the care of the solution of that problem.
- Patterns are used to develop reusable OO s/w.
- Patterns make easier to cense successful designs & architecture.
- Patterns are invented from the proven successful solution to recurring phenomena.

Ex. In real time student, house address , mail-in etc. Are unique and identified by assigning object identifier. This recurring phenomenon is recognized as object identifier pattern(OID).

- Patterns identify the key aspects of a common structure & make useful for creating reusable OO s/w.
- Good patterns should have the features as,

  ✓It solves a problem.
  ✓It is a proven concept with a perfect track record.
  ✓It describes a relationship of modules, system structure & mechanism in depth.
  ✓It must have the quality of aesthetic & utility.

- Patterns are generative & non-generative.

1.Generative patterns:

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Are abstract representation of system architecture and dynamic.
- Show the characteristic of a good system as well as teach has to build them.

2. Non-generative patterns:

- Static & passive.
- They just describe the recurring phenomena without telling how to reproduce them.

Pattern template:

SRM UNIVERSITY

❖Every pattern must be expressed "in the form of a rule[template] which establishes a relationship between a context, a system of forces which arises in that context & a configuration, which allows these forces to resolve themselves in that context.

❖Partition should contain essential components.

❖Following essential components should be clearly recognize on reading a pattern:

a) Name
b) Problem
c) Context
d)Forces
e)Solution
f) examples
g) Resulting context
h) Rationale
i) Related patterns
j) Known users

➢Good patterns must begin normally with an abstract, which provides a short summary of general outlook.

➢Anti patterns

✓Pattern represents a "best practice", wherever an anti pattern represents "worst practice" or "lesson learners"

✓Anti patterns come in two varieties:

→Those describing a bad solution to a problem that resulted is a bad situation

→Those describing how to get out of a bad situation & how to proceed from these to a good solution.

➢Capturing patterns:

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Process of working for patterns to document is called pattern mining(reverse architecture)
- It is important to remember that a solution in which no forces are present is not a pattern.
- Guidelines:
  - ❖Focus on practicability
  - ❖Aggressive disregard of originality
  - ❖Nonanonymous review
  - ❖Writers workshops instead of presentation
  - ❖Careful editing

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

Frameworks

Frame works:-

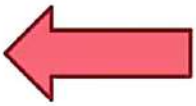
- It is a set of classes providing a general solution that can be refined to provide an application or system.
- They are reusable partial application that can be specialized further to produce custom application.
- They emphasize extensibility & reuse.
- They dictate the architecture of application.
- Are a way of delivering application development patterns to support best practice shaving during application development- not just within one company, but across many companies

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Framework is a way of presenting a generic solution to a problem that can be applied to all levels in a development.
- OO s/w framework is a set of cooperative classes, reuse the class.
- Framework & patterns are related to one another, but they have different meaning.
- Framework is executable software.
- Design patterns- exp knowledge & experience about s/w.

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Some difference
  - ✓Design patterns are more abstract than frameworks.
  - ✓Design patterns are smaller architectural elements than framework.
  - ✓Design patterns are less specialized than frameworks.

- **Overview of methodologies:**
  - –OMT
  - –Booch methodology
  - –Jacobson methodology
  - –Unified Approach 
- **UML:**
  - –Static and Dynamic Modelling
  - –UML diagrams.

Unified approach:-

- A methodology for s/w development, based on methodologies by Booch, Ram , Jae & others that tries to combine the best practices, processes & guidelines along with the object management group's UML for a better understanding of OO concept & s/w system development.
- Core concept is based on jacobson's use case model.
- UA to s/w development revolves around the following processes & concepts

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

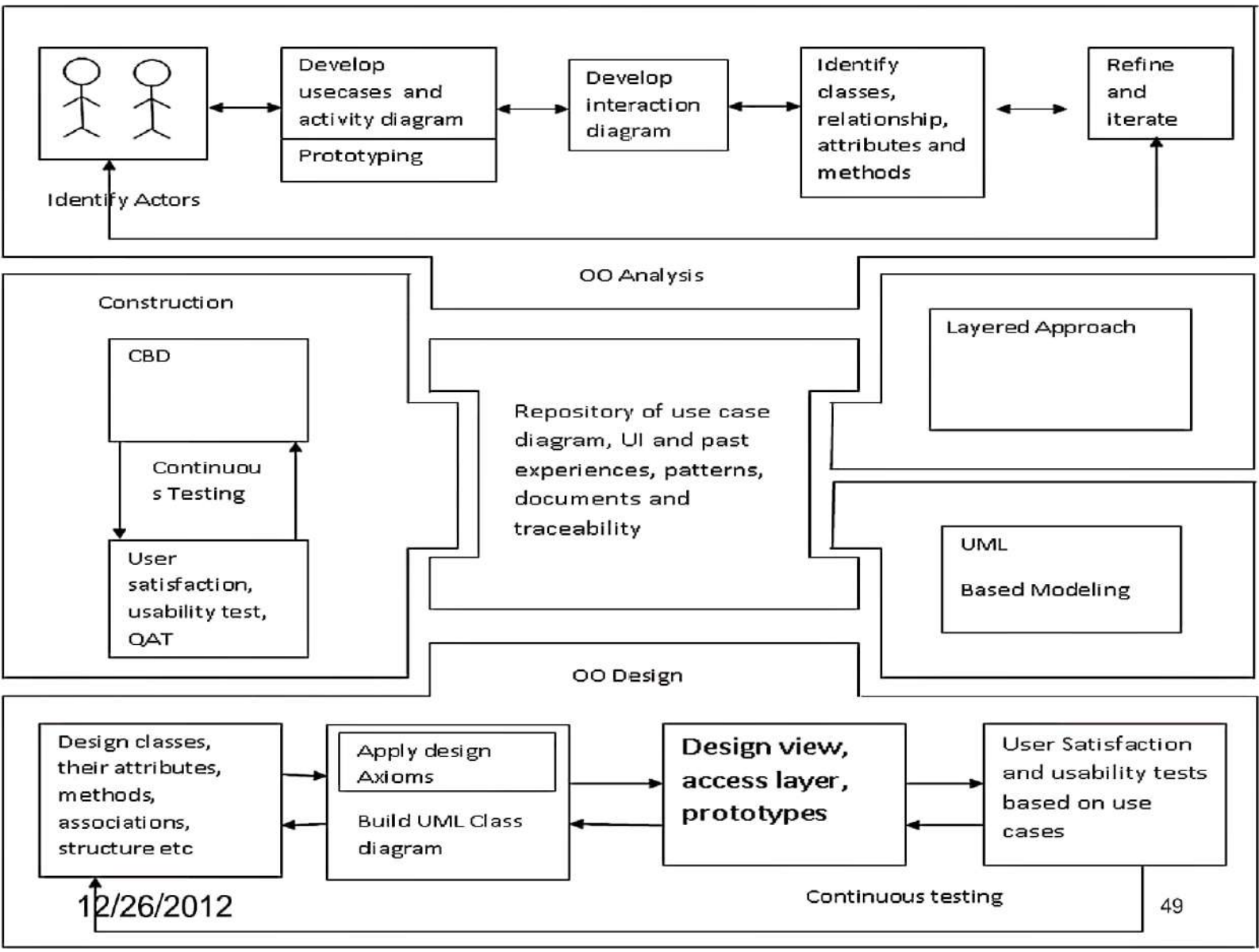
- The processes are:

✓Use-case driven development
✓Object oriented analysis
✓Object oriented design
✓Instrumental development & prototyping
✓Continuous testing

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Methods & technology employed include

✓UML used for modelling
✓Layered approach
✓Repository for OO system develop patterns & frameworks
✓CBD



SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

1. OO analysis:

- ✓Identify the actors
- ✓Develop a simple business process model using UML activity diagram
- ✓Develop the use case
- ✓Identify classes

2.OO design:

- ✓Designing classes, attributes, method, association, structure, protocols
- ✓Design access layer
- ✓Design & proto type user interface
- ✓User satisfaction & test based on use cases
- ✓Iterate & define the design

3.Iterative development & continuous testing:

- ✓Iterative develop till the system satisfaction
- ✓Continuous testing in each phase

4.Modelling based on UML

- ✓Used for modelling
- ✓Std rotation for OO modelling system
- ✓UA uses UML to describe & model the analysis
- ✓Design phases of system development

5.UA proposed repository:

- ✓Repository -storage of code, pattern, document etc.

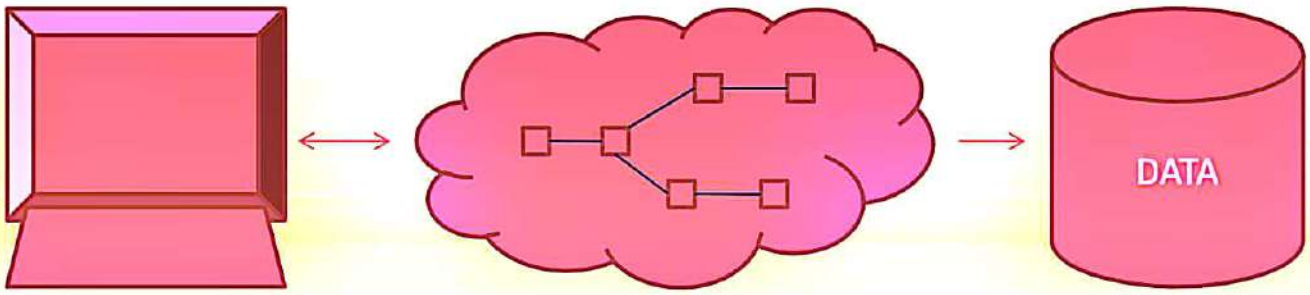
6.Layered approach to s/w development:

→Most system developed with today's case tools or client server application development environment tend to lean toward what is known as two- layered architecture interface & data

Work Station

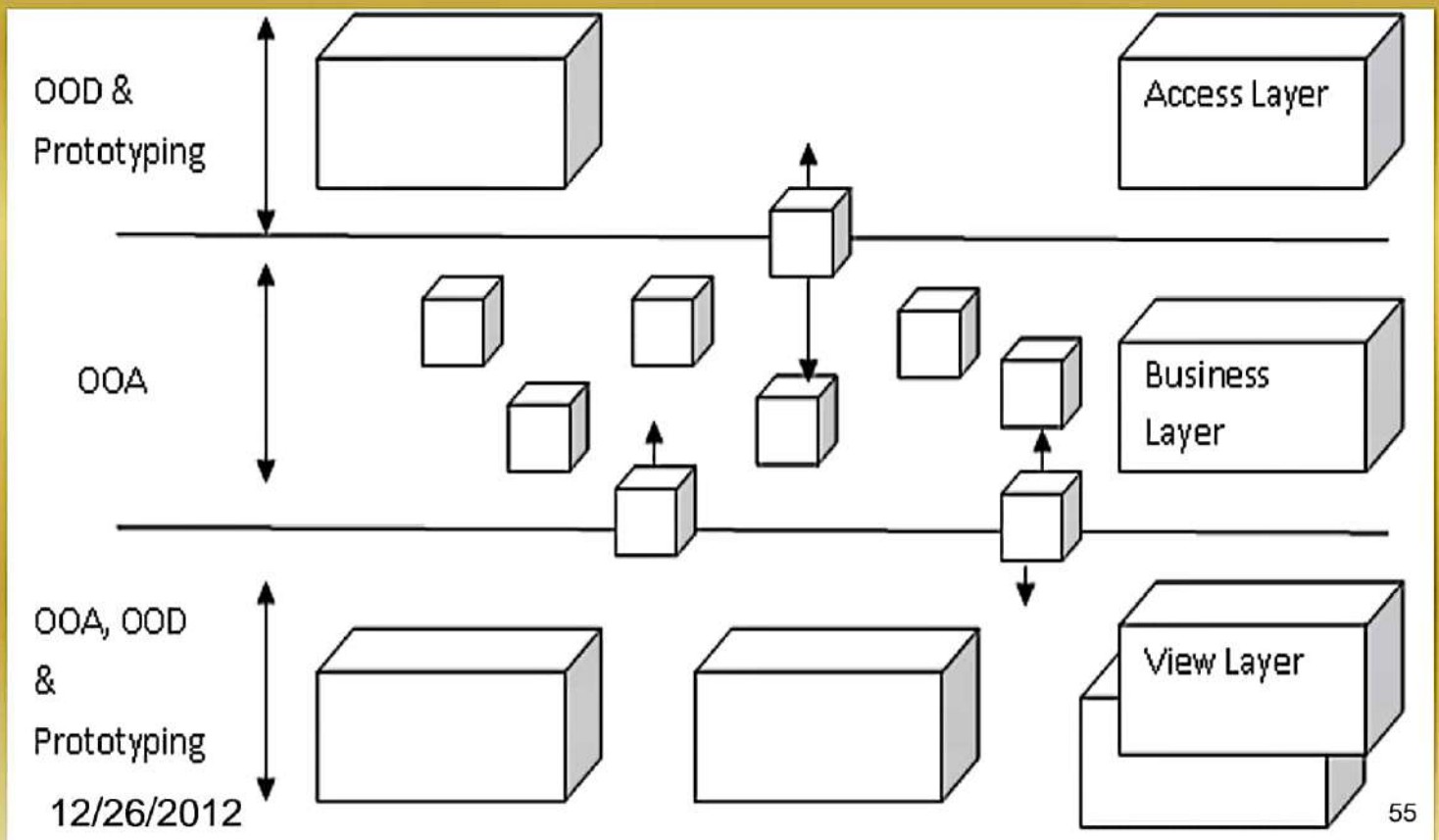
➢A better approach to system architecture is one that isolates the function of the interface from then fun of the business.

➢Using three layered approach, we are able to create object that represent tangible element of your business, yet are completely independent of how they are rep to the user.



Work Station

➢The three-layered approach consists a view or user interface layer, business layer & an access layer.



A. Business layer:

- It contains all the objects that represent the business(both data & behaviour)
- Responsibilities of business layer – to model the objects of the business & how they interact to accomplish the business process.
- When crating the business layer, it is important to keep in mind a couple things.
- These object should not be responsible for
  - ✓Displaying details
  - ✓Data access details

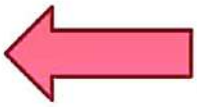
B. User interface (view) layer:

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- This layer consists of objects with which the user interface as well as the objects needed to manage or control the interface.
- Also called view layer.
- This layer typically responsible for two major aspects.
  - ✓Responding to user interaction.
  - ✓Displaying business objects.
- UI layer object are identified during the OO design phase.

C. Access layer:

- It contain object that know how to communicate with the place where the data actually resides, whether it be a relational database, mainframe, internet or file.
- Regardless of where the data actually reside, the access layer has two major responsibilities.
  - ✓Translate request
  - ✓Translate results
- Access object are identified during OOD.

- **Overview of methodologies:**
  - –OMT
  - –Booch methodology
  - –Jacobson methodology
  - –Unified Approach
- **UML:** 
  - –Static and Dynamic Modelling
  - –UML diagrams.

Introduction:

Model:-

→Is an abstract representation of a system, constructed to understand the system (Is any process or structure) prior to building /modifying it

→Modelling enables us to cope with the complexity of a system.

→Most modelling technique used for analysis and design involve graphic languages (set of **symbols**(are used according to certain rules of methodology for communicating the complex relationships of information))

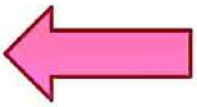
→Main goal of CASE tool is using these graphical language along with association and methodologies

→Modelling is done in most of the phases of software life cycle such as analysis , design, and implementation

→Example for different models

1. Use-Case Model
2. Domain – Object Model
3. Analysis Object Model
4. Implementation Model
5. Test Model

→Models are represented – static and dynamic situations

- **Overview of methodologies:**
  - –OMT
  - –Booch methodology
  - –Jacobson methodology
  - –Unified Approach
- **UML:**
  - –Static and Dynamic Modelling 
  - –UML diagrams.

Static Model:-

SRM UNIVERSITY
(under section 3 of UGC Act 1956)

→It can be viewed as a snapshot of a system's parameters at rest or at a specific point in time

→Are needed to represent the structural or static aspect of a system

→Static models assume stability and an absence of change in data overtime

→eg., Class diagram : a customer can have more than one account

Dynamic Model:-

→Contrast to a static model, can be viewed as a collection of procedures or behaviour that, taken together, reflects the behaviour of a system over time

→Dynamic relationships show how the business object interact to perform tasks

→System can be described by first developing its static model, which is the structure of its objects and their relationships to each other frozen in time, a baseline

→Useful during design and implementation phases of the system development

→Eg: UML Interaction Diagram and Activity Diagram, Order interact with to determine product availability

Why Modelling?

→Building a model for software system is like to have a blueprint for a building large construction

→Modelling language must include

1. Model elements – fundamental modelling concepts and semantics
2. Notations – visual rendering of modelling elements
3. Guidelines – expression of usage within the trade

→Use of model, a problem provides us several benefits relating to

- ✓Clarity
- ✓Familiarity
- ✓Maintenance
- ✓Simplification

→Advantages

1. Designed to use complex ideas
2. Reduce complexity
3. Enhance learning and training
4. Cost is less compared to real system experience
5. Manipulation is easier

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

Summarize

1. It is rarely correct in the first try
2. Seeks advice and criticism of others
3. Avoid excess modelling revisions

Introduction to Unified Modelling Language:-

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

→UML is a language for

- ✓Specifying
- ✓Constructing
- ✓Visualizing
- ✓Documenting the s/w sym and its concepts

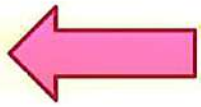
→Is a graphical language with a set of rules and semantics in the form of OCL(object Constraints Language)

→Is not intended to be a visual programming language, but have a high mapping to OOP Language

→Primary goals in the design of the UML were as follows

1. Provides users a ready- to- use, expressive visual modelling language so they can develop and exchange meaningful models
2. Provide extensibility and specialization mechanisms to extend the core concept
3. Be independent of particular programming language and development processes
4. Provide the formal basis for understanding the modelling language
5. Encourage the growth of the OO tools market
6. Support higher-level development concepts
7. Integrated best practices and methodologies

- **Overview of methodologies:**
  - –OMT
  - –Booch methodology
  - –Jacobson methodology
  - –Unified Approach
- **UML:**
  - –Static and Dynamic Modelling
  - –UML diagrams.



SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

What is the UML?

- Unified Modeling Language
- It is a modeling language, not a process
- Rumbaugh joined Booch at Rational in 1994; in 1995, Rational added Jacobsen to their team. In 1996, work on the UML was begun.
- In January of 1997, Rational released UML 1.0 to the OMG as their proposal for a methods standard.

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

UML Diagrams to be Covered

1. Class Diagrams (Static)
2. Use Case Diagrams
3. Behavior Diagram (Dynamic)
   I. Interaction Diagram
      1. Collaboration Diagrams
      2. Sequence Diagrams
   II. State chart Diagram
   III. Activity Diagram
4. Implementation Diagram
   I. Component Diagrams
   II. Deployment Diagrams

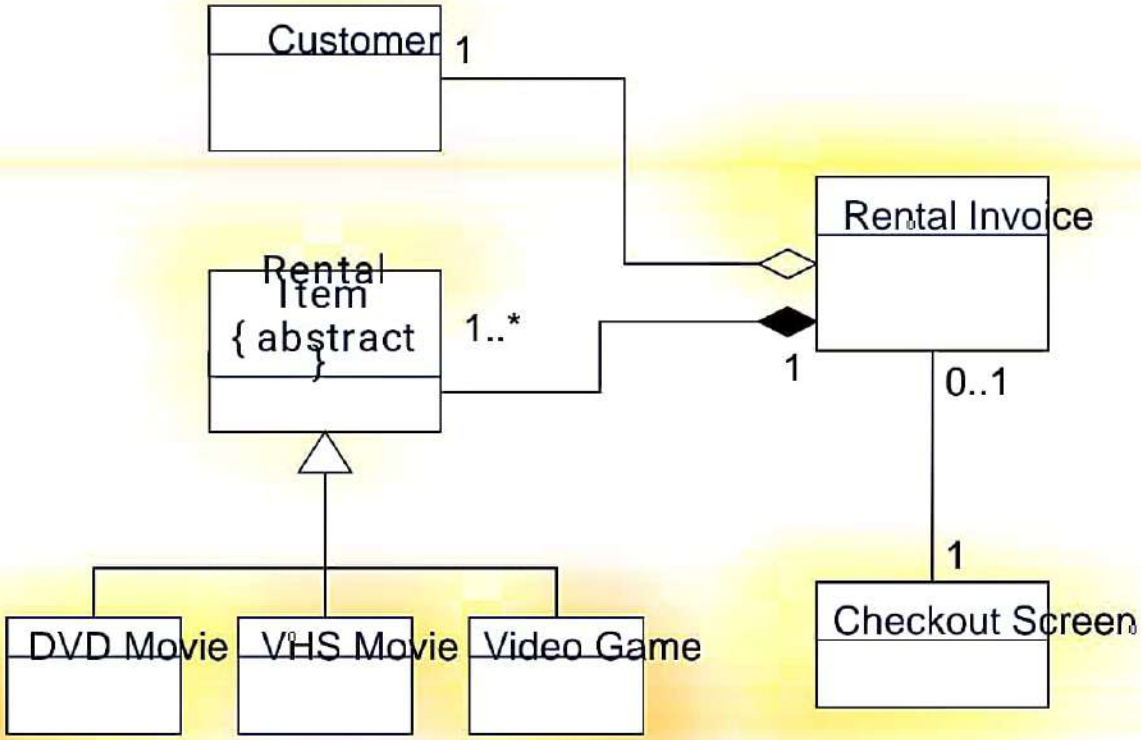
SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

Class Diagrams

- Are the most fundamental UML Diagram.
- Describe the classes in the system, and the static relationships between classes.
- Class diagrams are used during Analysis, Design and Development.

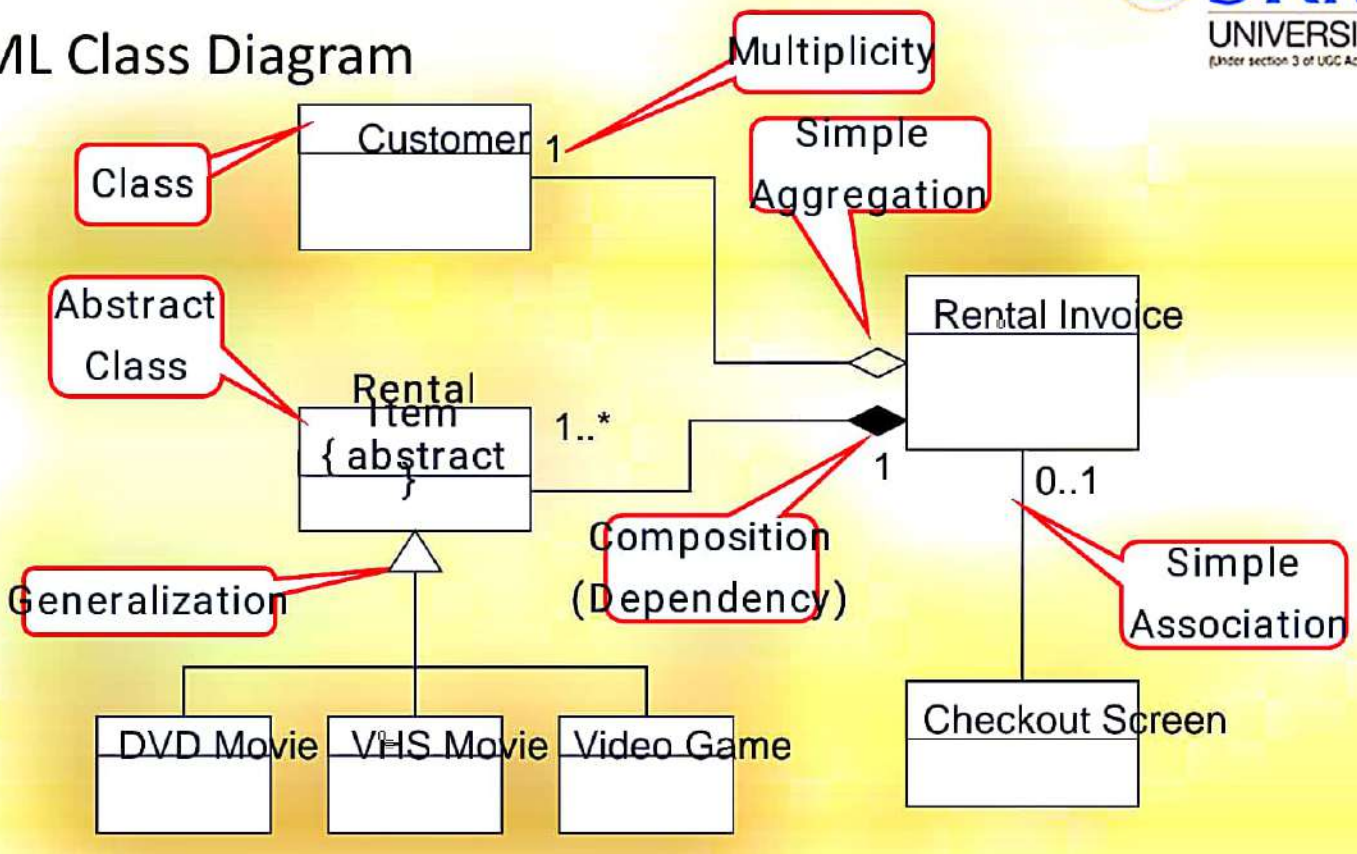
SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

UML Class Diagram



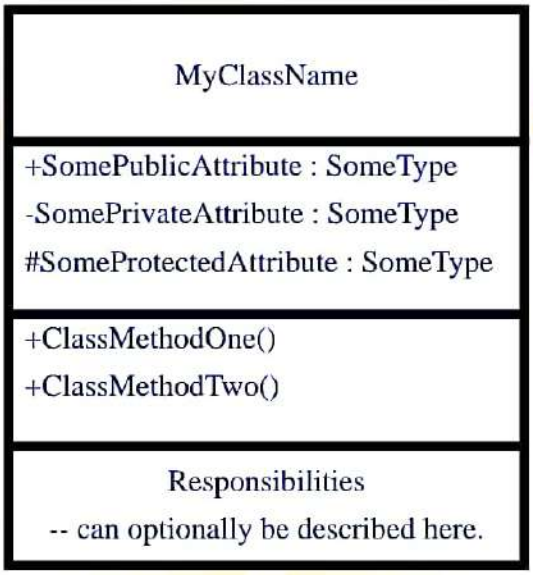
SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

UML Class Diagram



Parts of a Class

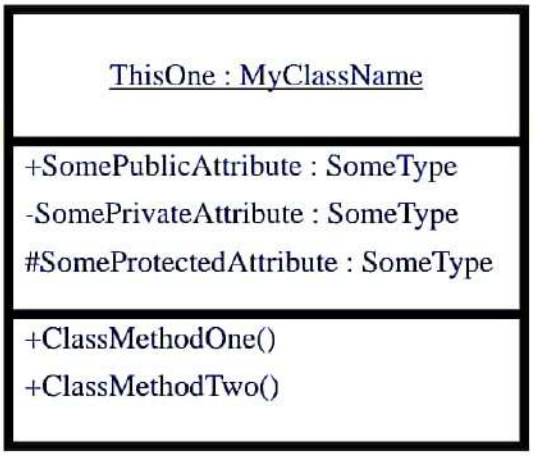
- Classes can have four parts
  - Name
  - Attributes
  - Operations
  - Responsibilities
- Classes can show visibility and types.
- All parts but the Name are optional.



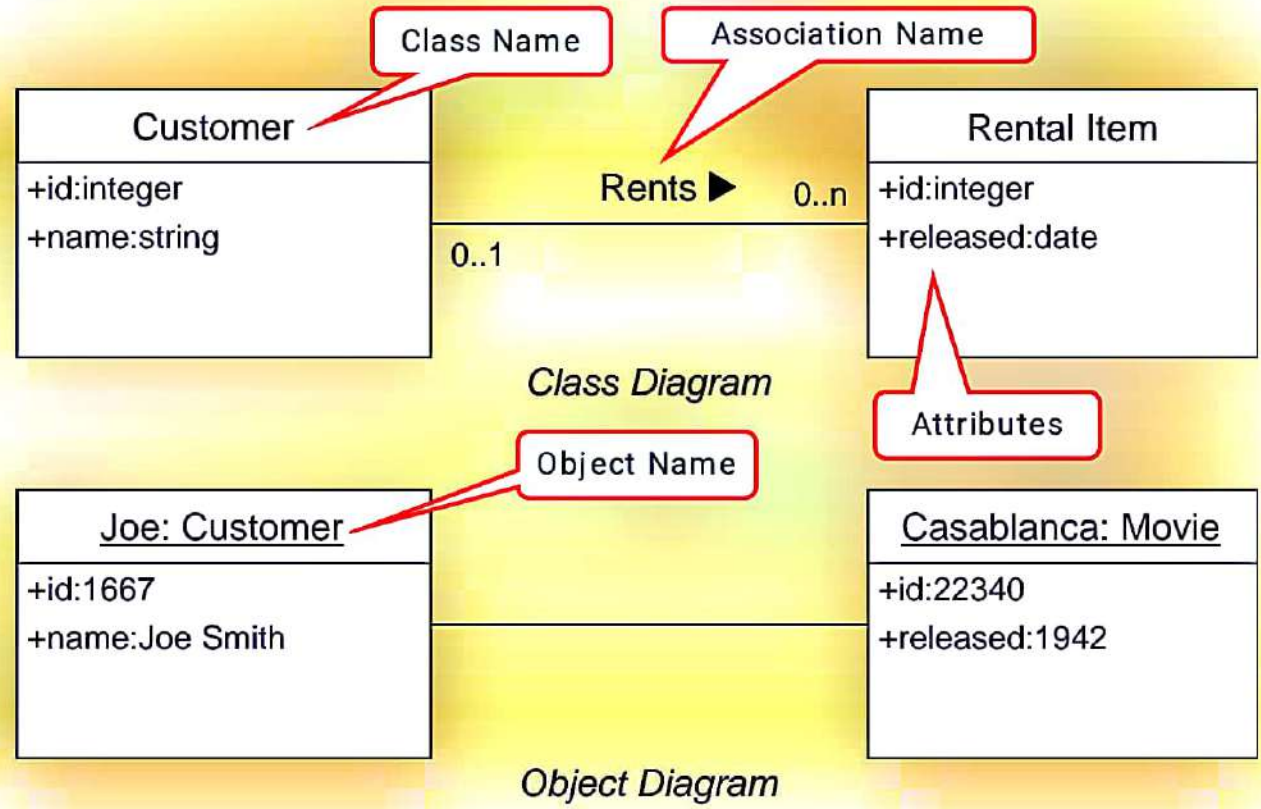
SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

Object Diagrams

- An Object is an instance of a class.
- Object names are underlined.
- Object diagrams are similar to class diagrams. Many of the same notations are used.
- Object diagrams capture instances of classes, and allow the dynamic relationships to be shown.

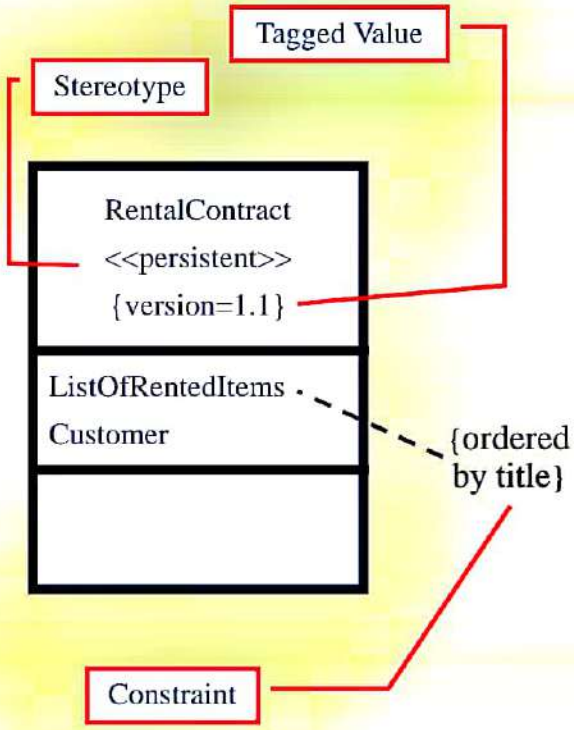


Class and Object Diagrams



Stereotypes, Tagged Values and Constraints

- Stereotypes, Tagged Values and Constraints extend the UML
  - Stereotypes are shown using <<>>
  - Tagged Values and Constraints are shown using { }

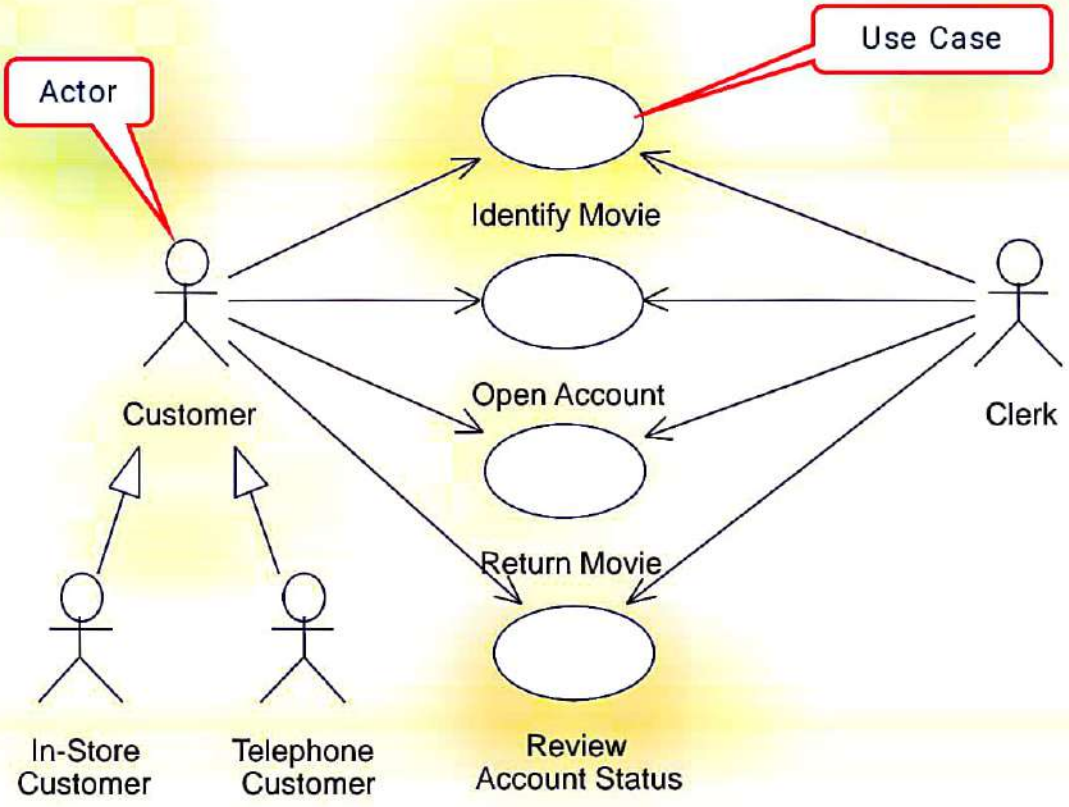


Use Cases

- Describe interactions between users and computer systems (both called actors) .
- Capture user-visible functions.
- Achieve discrete measurable goals.
- May be
  - small ("Make selected text bold")
  - large ("Generate a table of contents")
- Are typically used during Analysis and Design.

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

Use Case Diagram

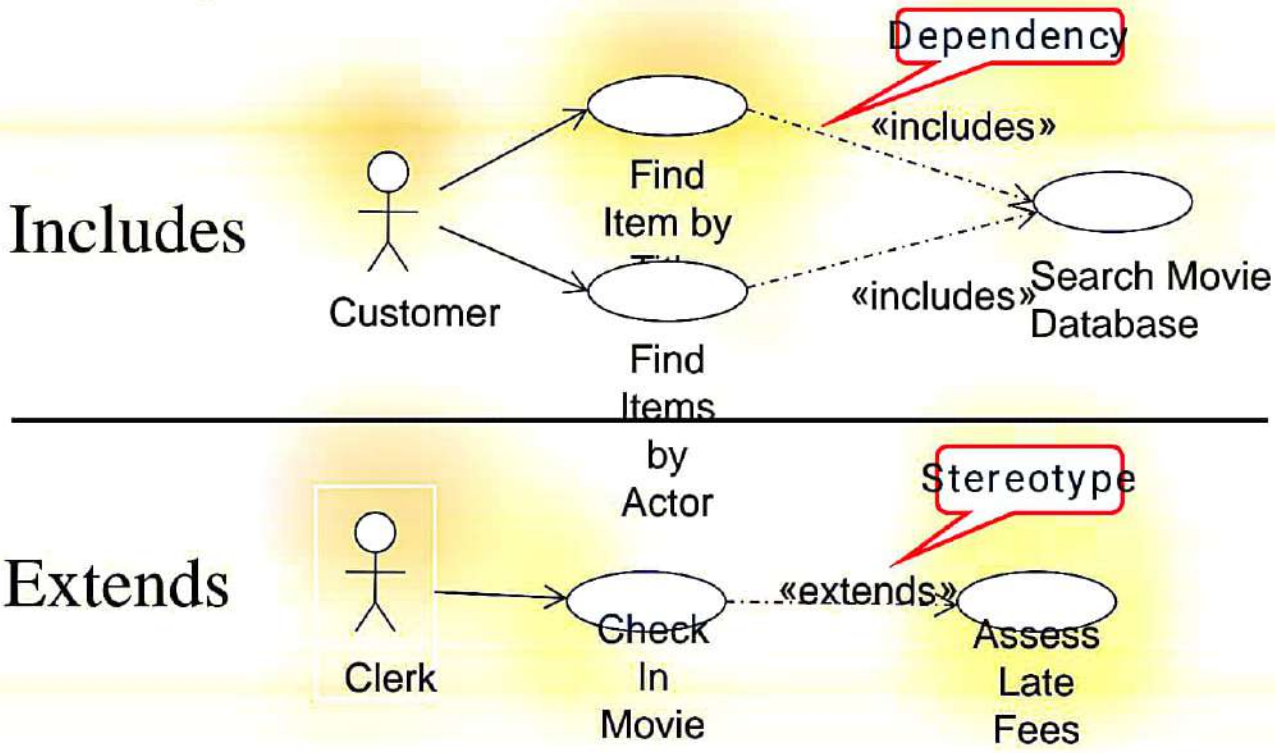


Use Case Report

- The Use Case Report provides documentation for the Use Case.
- A Use Case is not complete without the report.
- The elements of the Use Case Report are shown on the right.

- Brief description
- Precondition
- Flow of events
  - Main flow
  - Subflows
  - Alternate flows
- Postcondition
- Special Requirements
- Enclosures
  - Diagrams
  - Pictures of the UI

Extends and Includes Relationships

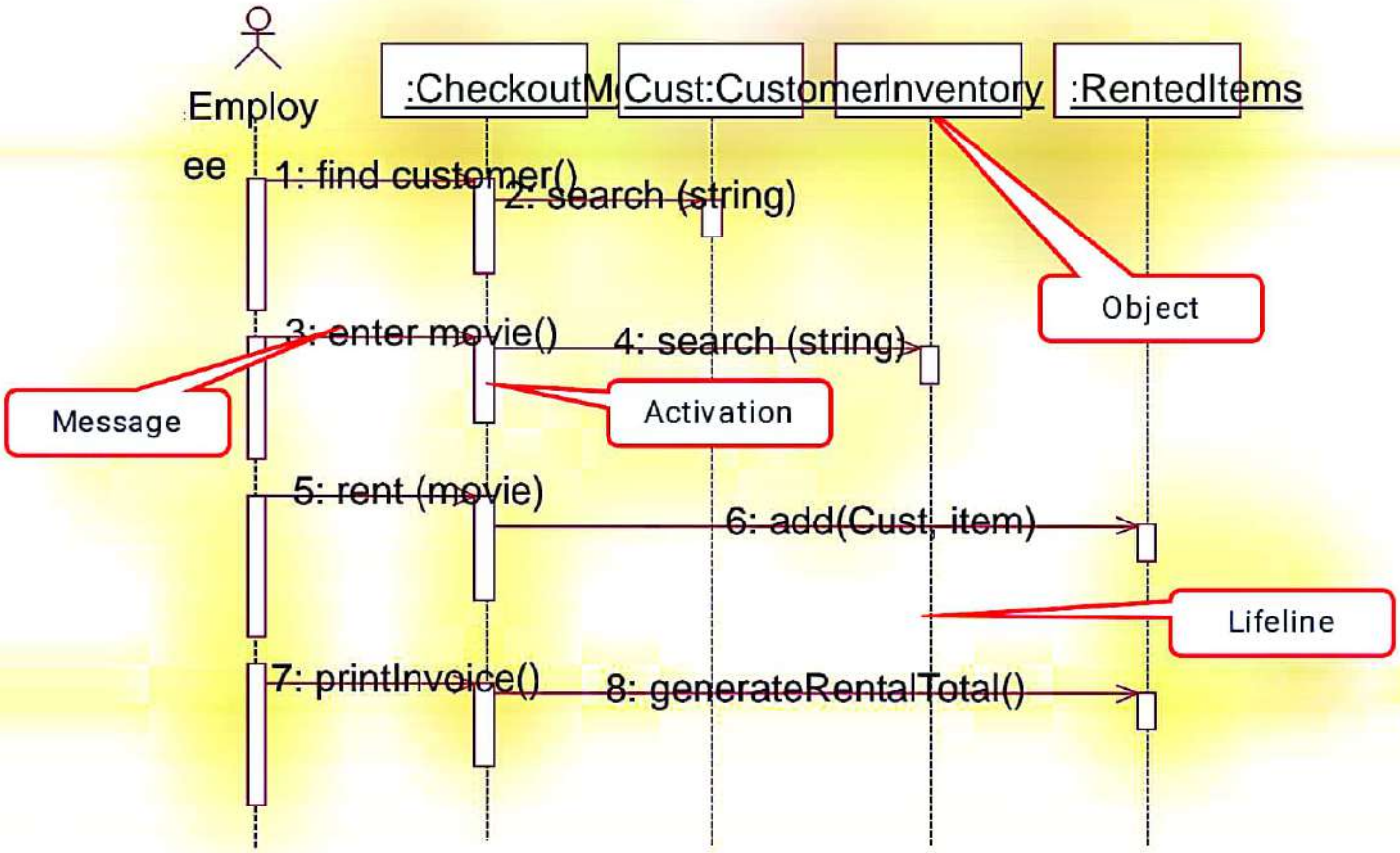


Sequence Diagrams

- Can be "morphed" from Collaboration Diagrams.
- Describe interactions between objects arranged in time sequence
- Focus on objects and classes involved in the scenario and the sequence of messages exchanged
- Associated with use cases
- Used heavily during Analysis phase and are enhanced and refined during Design phase

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

Sequence Diagram - Rent Movie

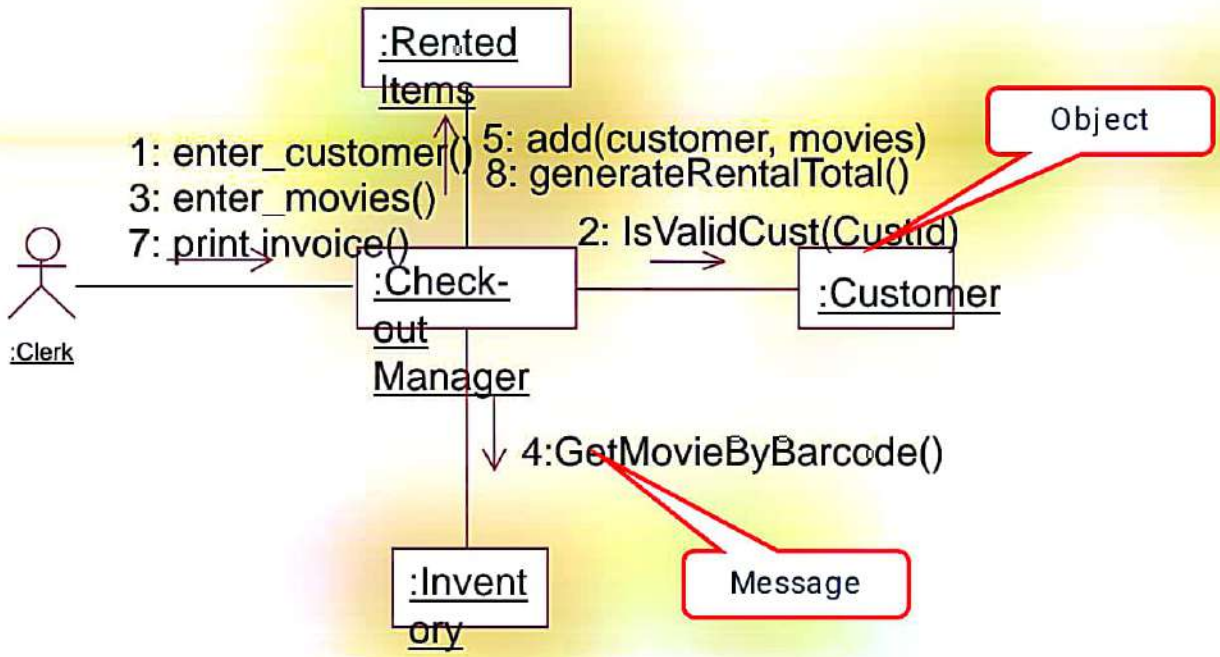


Collaboration Diagrams

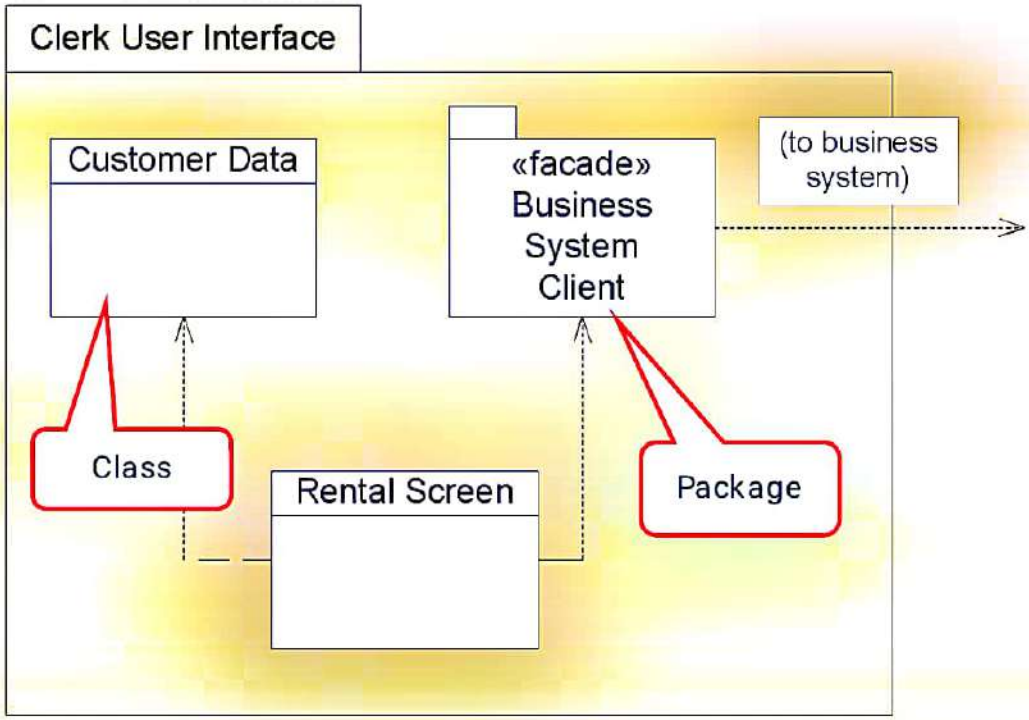
- Collaboration diagrams describe object interactions organized around the objects and their links to each other
- Focus on exchange of messages between objects through their associations.
- Appears during Analysis phase
- Enhanced during Design phase

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

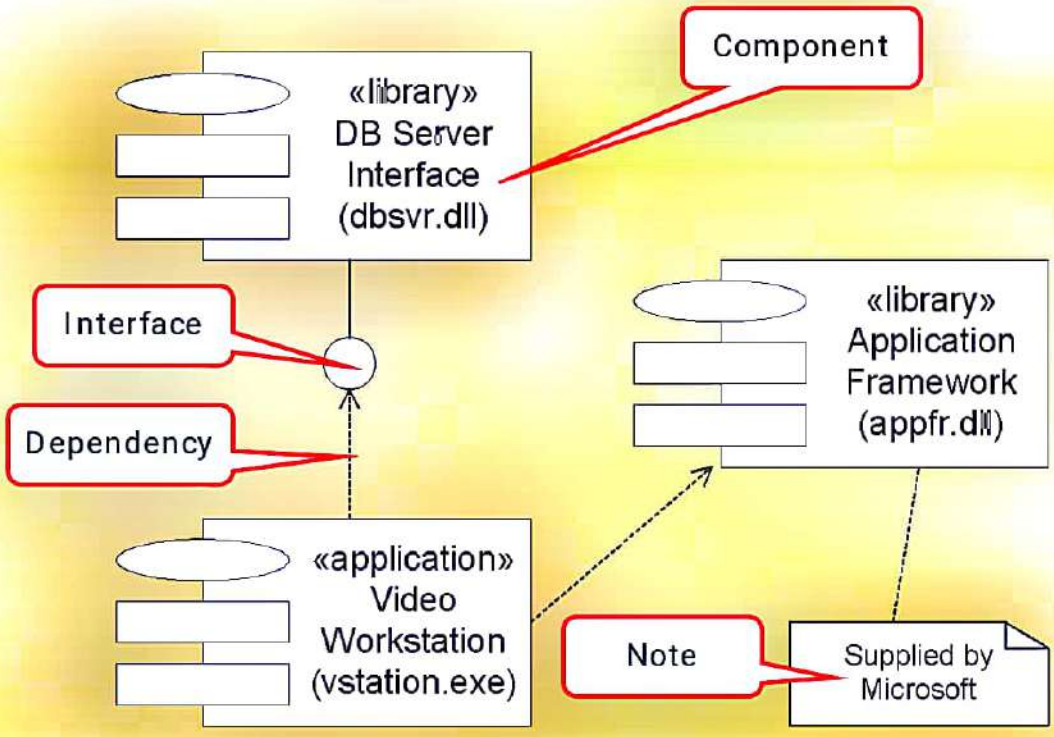
Collaboration Diagram - Rent Movie



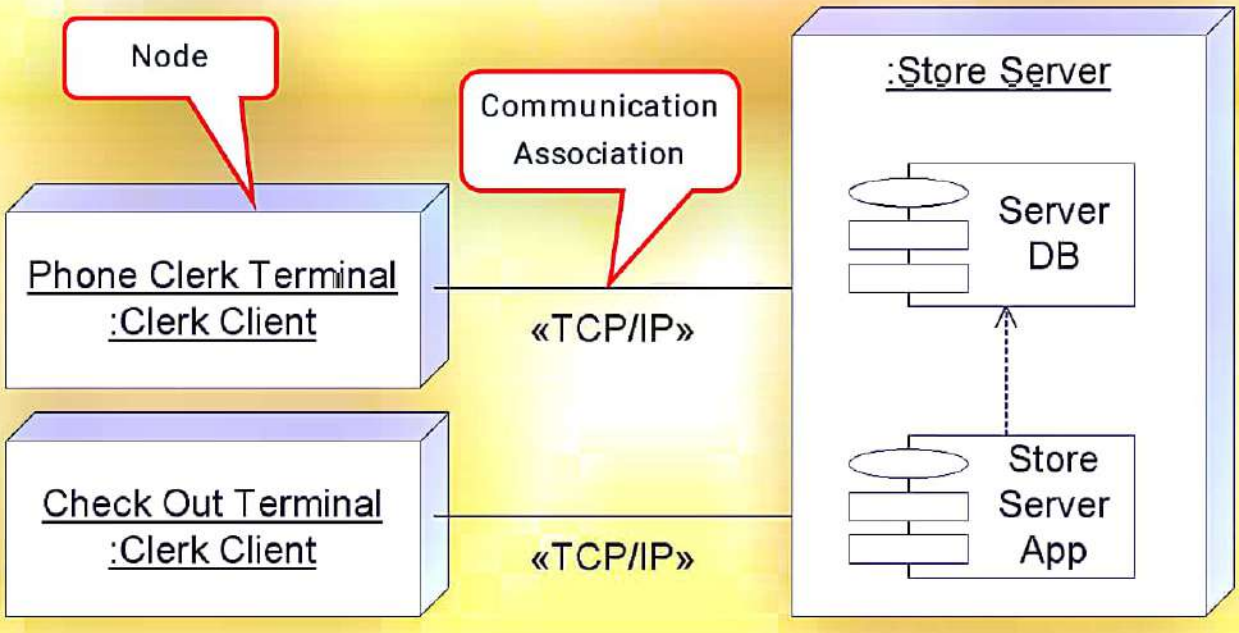
Package Diagram



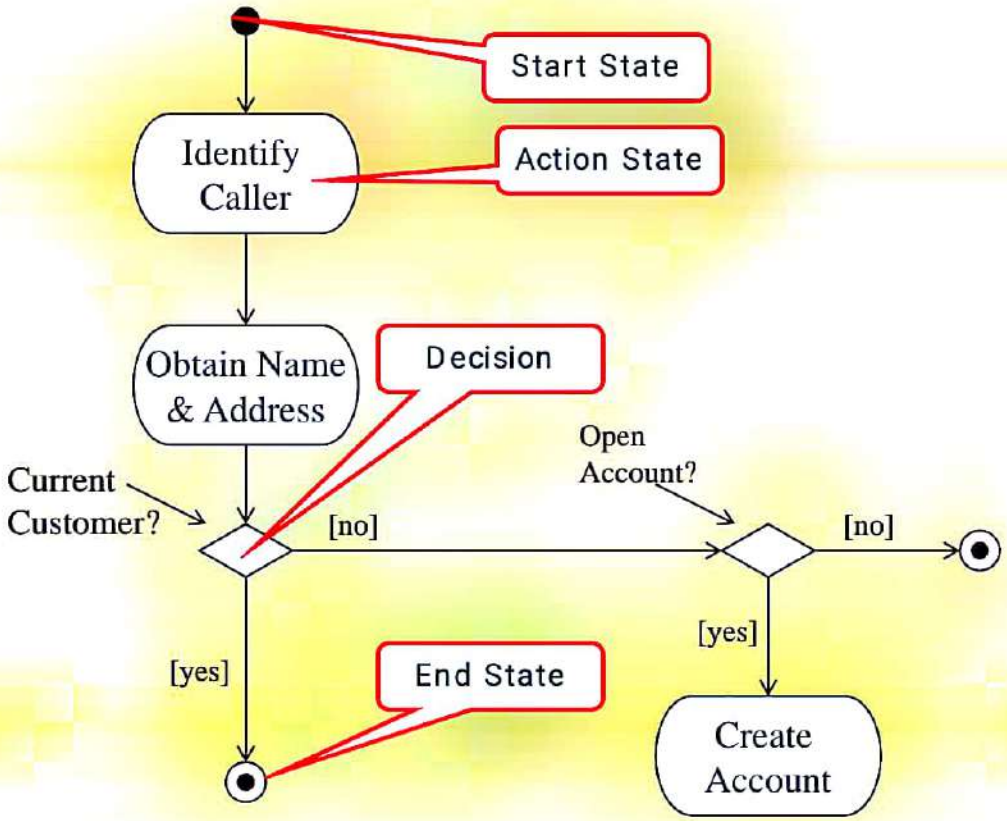
Component Diagram



Deployment Diagram

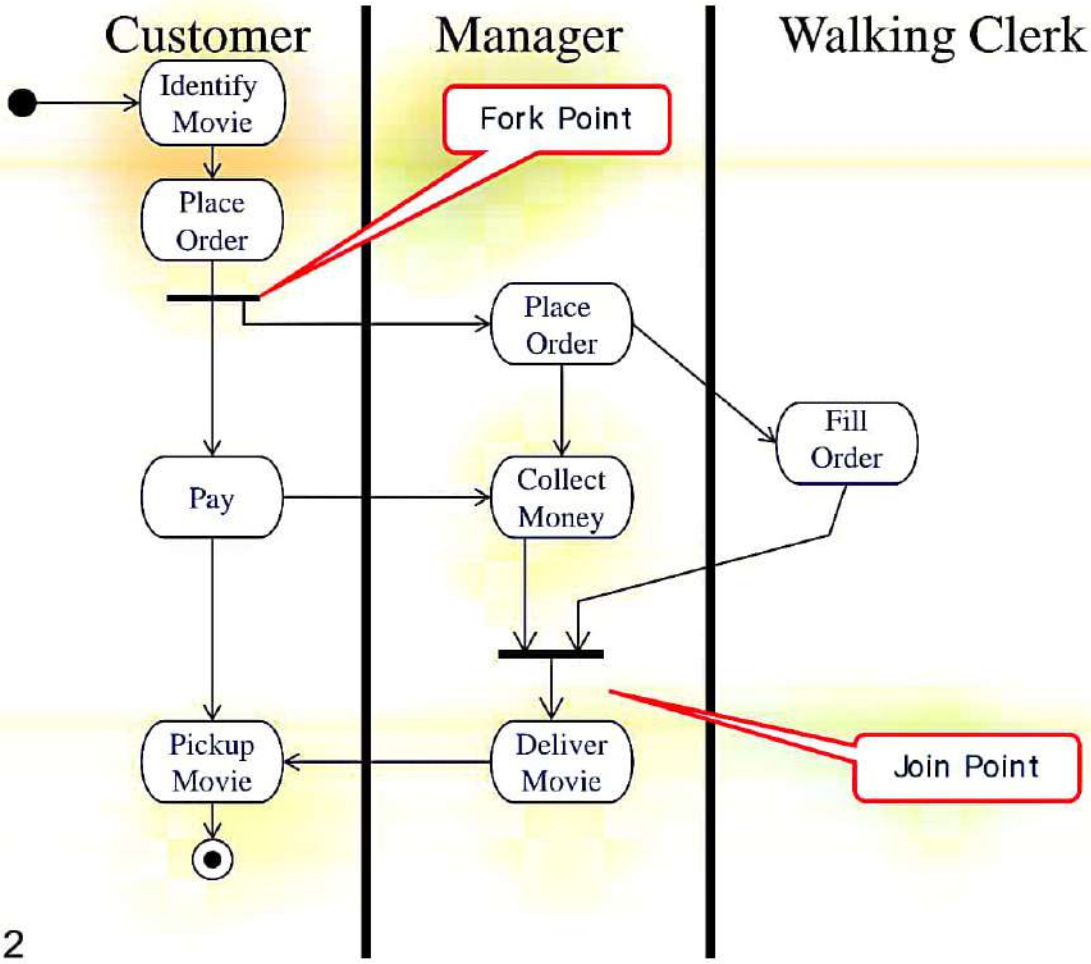


Activity Diagram

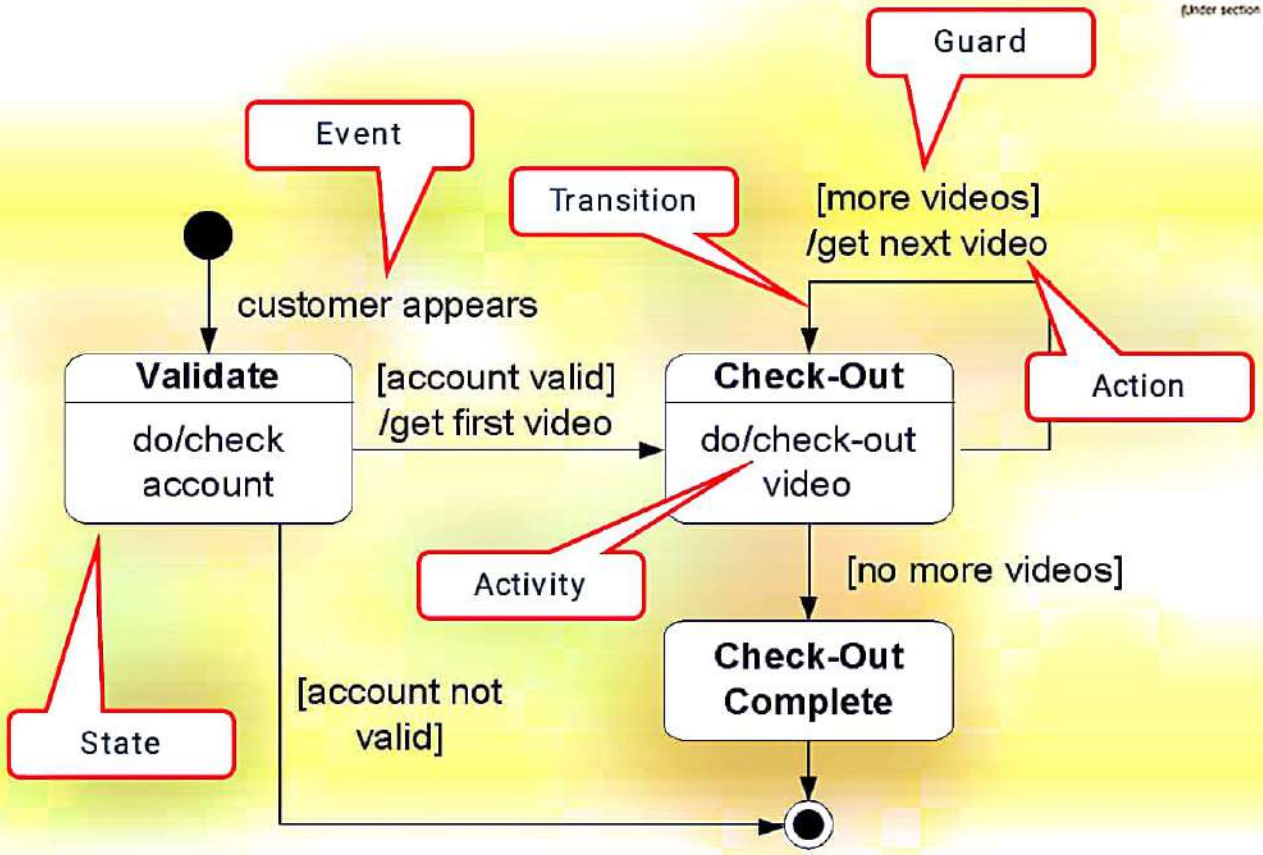


SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

Swimlanes and Fork/Join Points



State Diagram



SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

UML Diagram Usage

Development Phase	UML Diagrams
Analysis	Use Cases, Class Diagrams, Activity Diagrams, Collaboration Diagrams, Sequence Diagrams
Design	Class Diagrams, Collaboration Diagrams, Sequence Diagrams, State Diagrams, Component Diagrams Deployment Diagrams
Development	Collaboration Diagrams, Sequence Diagrams, Class Diagrams, State Diagrams, Component Diagrams, Deployment Diagrams
Implementation	Package Diagrams, Deployment Diagrams

Review Questions

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- What is a use case? Explain with example.
- Describe the difference between patterns and frame works.
- What is UML? What is the importance of UML?
- Why do we need to model a problem?
- What is macro development process?
- Name the four primary symbols of data flow diagrams.
- Explain macro development process.
- Name the pattern templates.
- What are the processes and concepts of unified approach to software development?
- What are the steps involved on OOA of unified approach.
- What is the use of repository in Unified Approach?
- List out the relationships of usecase.
- What is a model?
- What is a qualifier of an association?
- Describe the class diagram.

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

Bibliography

- Object oriented system development by Ali Brahami
- Object oriented methodology by Booch
- A text book on UML by Srimathi

Module 4

MEMORY MANAGEMENT

Topics

Main memory

1. Swapping,
2. Contiguous Memory Allocation,
3. Paging,
4. Structure Of Page Table,
5. Segmentation,
6. Examples.

Virtual memory

1. Demand Paging,
2. Copy On Write,
3. Page Replacement,
4. Allocation of Frames,
5. Thrashing,
6. Memory Mapped Files,
7. Allocating Kernel Memory,
8. Memory Management Utilities

Main Memory - Introduction

- Program must be brought (from disk) into memory and then brought into processor for execution
- Main memory and registers are only storage CPU can access directly
- A typical instruction-execution cycle, for example, first fetches an instruction from memory
- The instruction is then decoded and may cause operands to be fetched from memory
- After the instruction has been executed on the operands, results may be stored back in memory

Figure : A base and a limit register define a logical address space

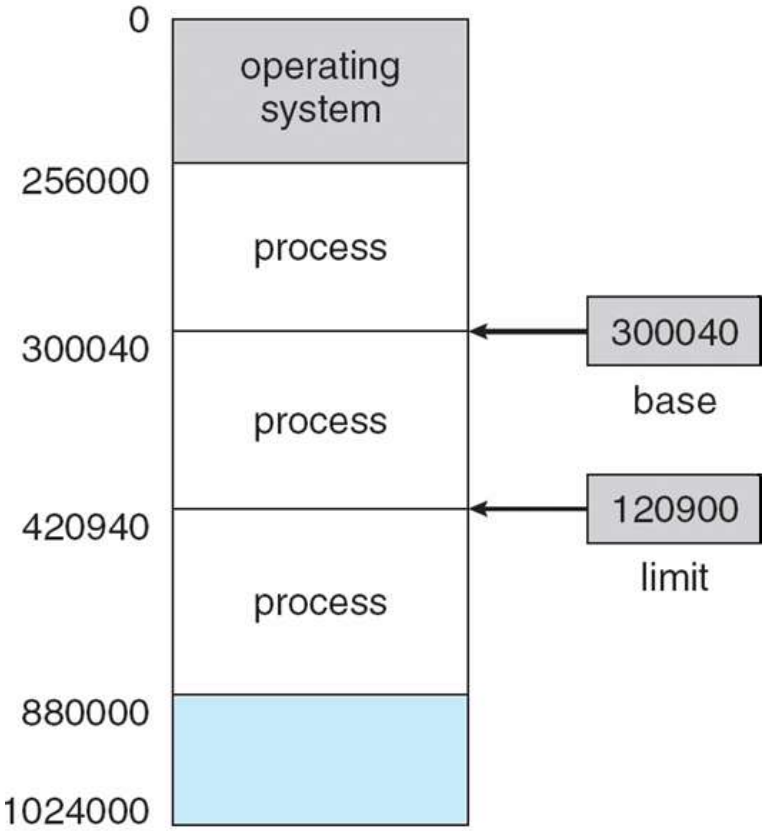
Each process has a separate memory space.

System has the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.

This protection is provided by using two registers, a base and a limit registers

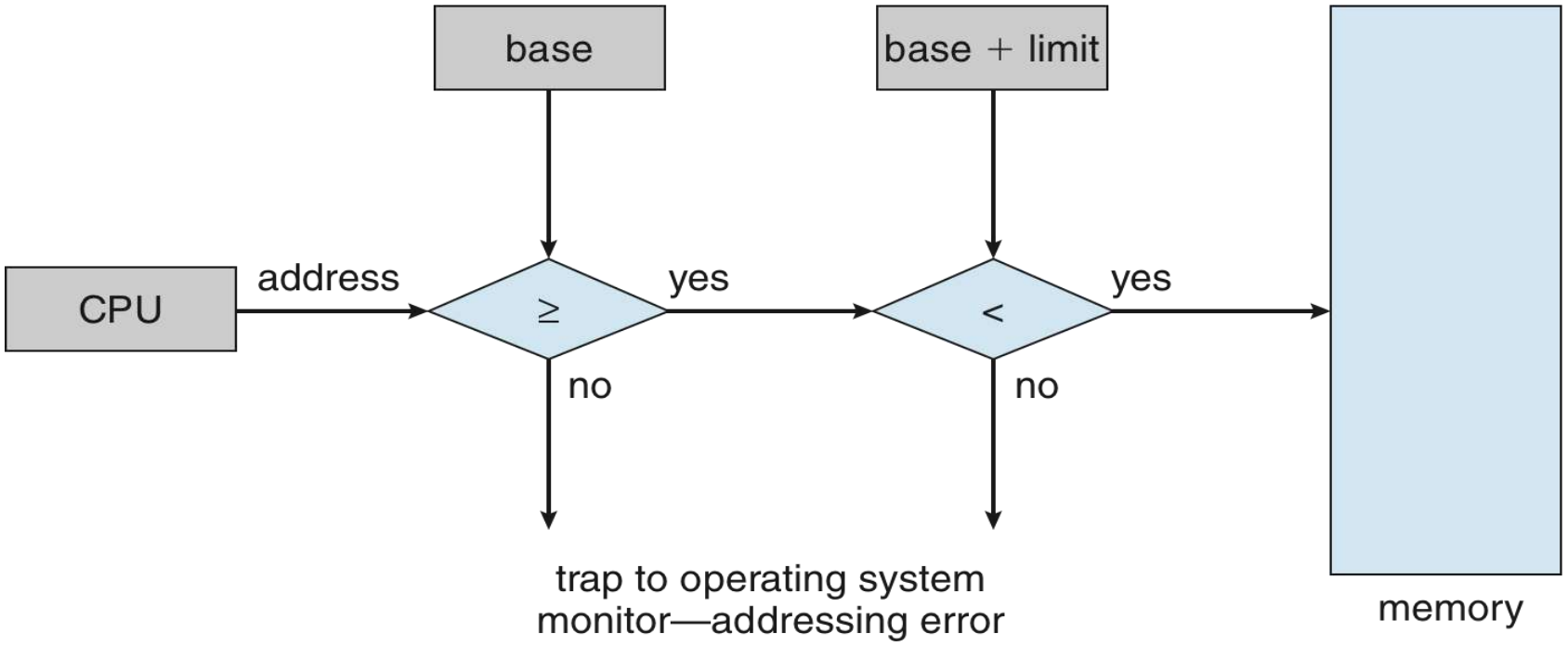
The base holds the smallest legal physical memory address;

Limit register the specifies the size of the range.



For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive of both).

Hardware Address Protection with Base and Limit Registers



Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.

Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error .

This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

Address Binding

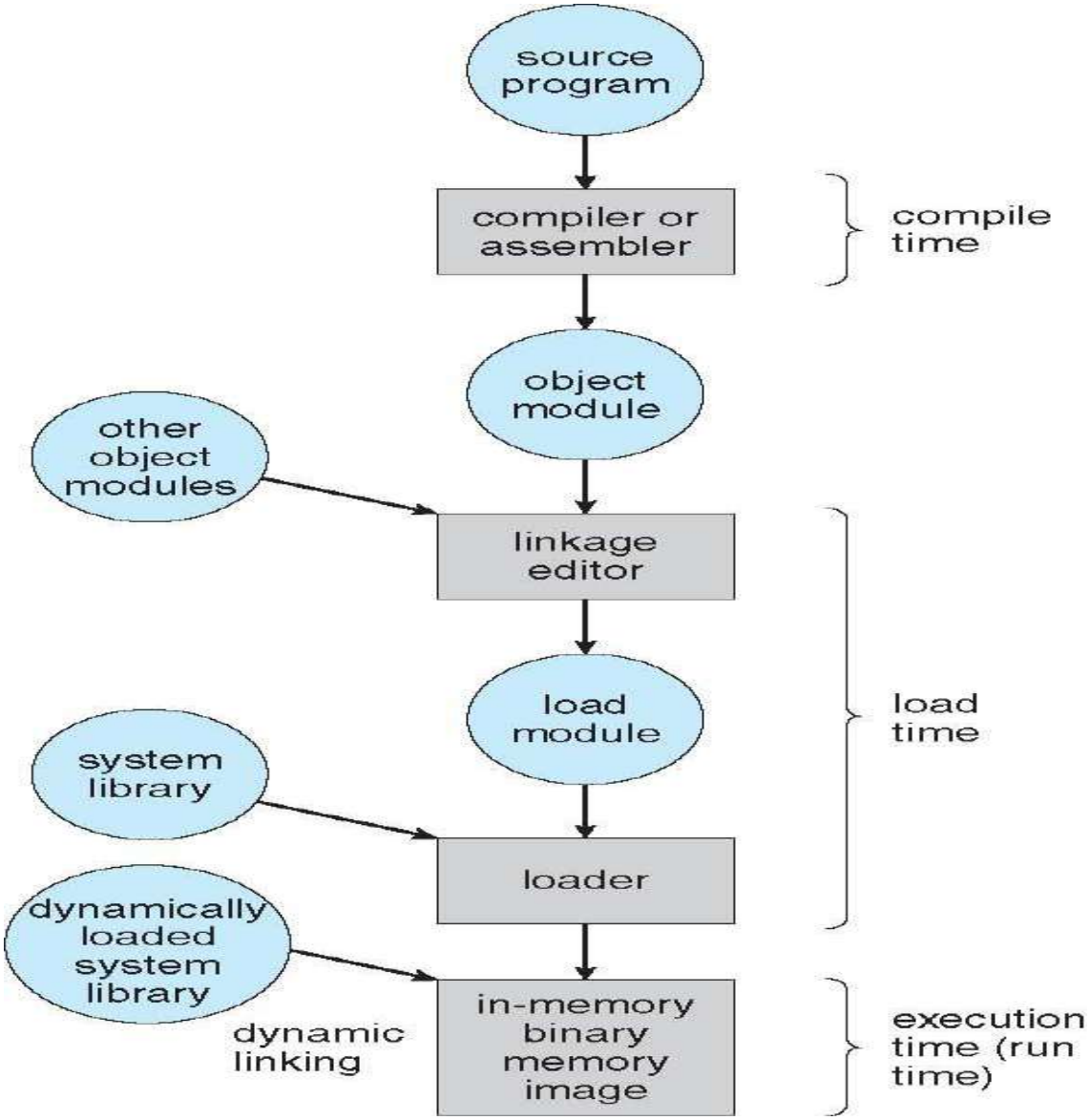
- Most systems allow a user process to reside in any part of the physical memory.

- Thus, although the address space of the computer starts at 00000, the first address of the user process need not be 00000.

- **Address binding of instructions and data to memory addresses** can happen at three different stages

  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit

- **Logical and physical addresses are the same in compile-time and load-time address-binding schemes**

- logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** the set of all physical addresses corresponding to these logical addresses

Memory-Management Unit

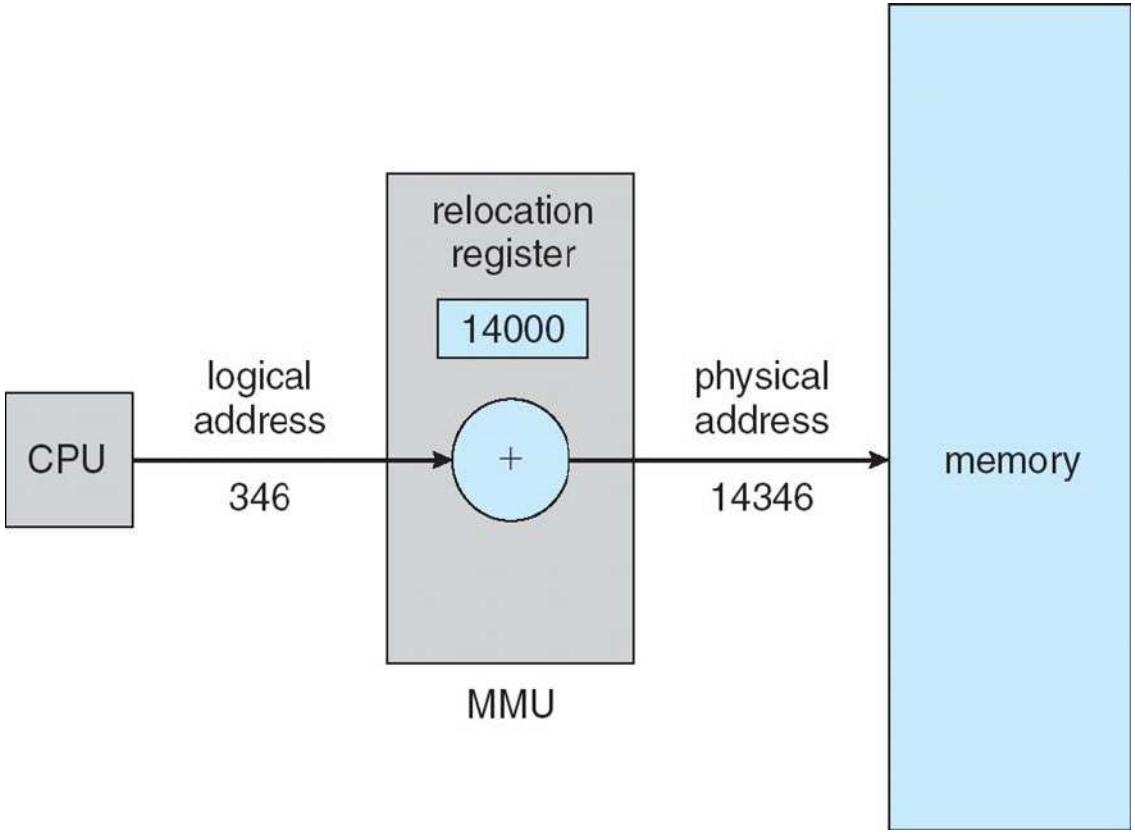
- **Hardware device that at run time maps virtual to physical address**

- Many methods possible

- Consider simple scheme where the **value in the relocation register is added to every address generated by a user process** at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers

- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

Dynamic relocation using a relocation register



- The base register is now called a the value in the relocation register is added to every address generated by a user process at the time the address is sent to memory.
- For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000
- an access to location 346 is mapped to location 14346.

Dynamic Loading

- it has been necessary for the entire program and all data of a process to be in physical memory for the process to execute.
- The size of a process has thus been limited to the size of physical memory.
- **To overcome this in dynamic loading , routine is not loaded until it is called**
- **Better memory-space utilization; unused routine is never loaded**
- **All routines kept on disk in relocatable load format**
- **Useful when large amounts of code are needed to handle infrequently occurring cases**
- **No special support from the operating system is required**
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

Dynamic Linking

- **Static linking – system libraries and program code combined by the loader into the binary program image**
- **Dynamic linking –linking postponed until execution time**
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

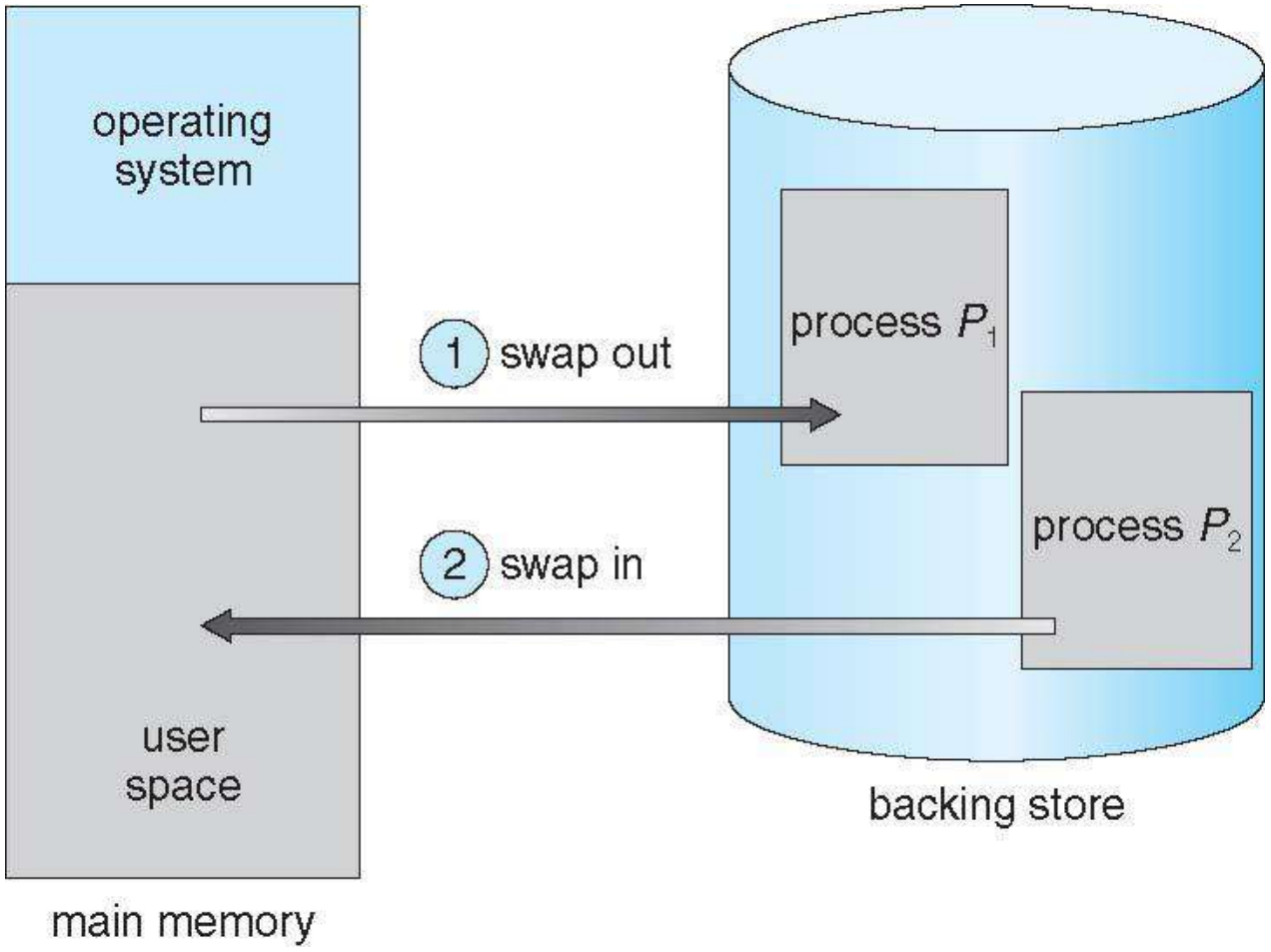
Swapping

- A process must be in memory to be executed.
- **A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.**
- For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm
- When a quantum expires, the memory manager will start to swap out the process that just finished and
- to swap another process into the memory space that has been freed

Swapping

- In the meantime, the CPU scheduler will allocate a time slice to some other process in memory.
- When each process finishes its quantum, it will be swapped with another process.
- A variant of this swapping policy is used for priority-based scheduling algorithms
- If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process
- When the higher-priority process finishes, the lower-priority process can be swapped back in and continued
- **This variant of swapping is sometimes called roll out and roll in**

Schematic View of Swapping



Swapping

- **Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously.**
- This restriction is dictated by the method of address binding.
- **If binding is done at assembly or load time, then the process cannot be easily moved to a different location.**
- **If execution-time binding is being used, however, then a process can be swapped into a different memory space,**
- because the physical addresses are computed during execution time
- Swapping requires a backing store. The backing store is commonly a fast disk.

Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
- The actual transfer of the 100-MB process to or from main memory takes
- 100MB/50MB per second= 2 seconds
  - Plus disk latency of 8 ms
  - Swap out time of 2008 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4016ms (> 4 seconds)

- Can reduce if reduce size of memory swapped – by knowing how much memory really being used-System calls to inform OS of memory use via `request memory` and `release memory`

Contiguous Allocation

- The memory is usually divided into two partitions: **one for the resident operating system and one for the user processes.**

- We can place the operating system in either low memory or high memory.

- The major factor affecting this decision is the location of the interrupt vector.

- Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.

Contiguous Allocation

- **We usually want several user processes to reside in memory at the same time**

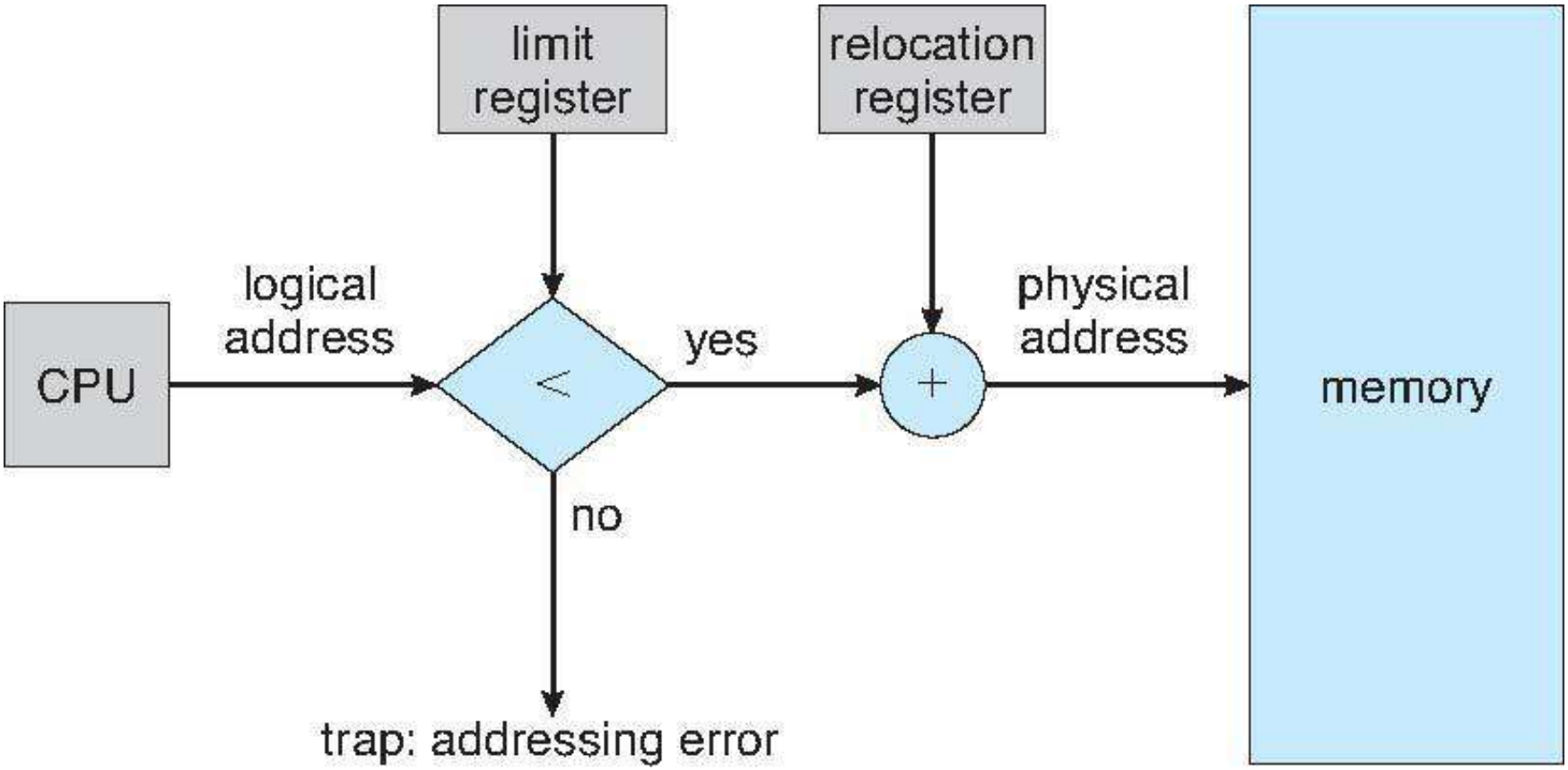
- We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory

- **In contiguous memory allocation, each process is contained in a single contiguous section of memory**

Memory Mapping and Protection

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - **Base register contains value of smallest physical address**
  - **Limit register contains range of logical addresses – each logical address must be less than the limit register**
  - **MMU maps logical address *dynamically***

Hardware Support for Relocation and Limit Registers

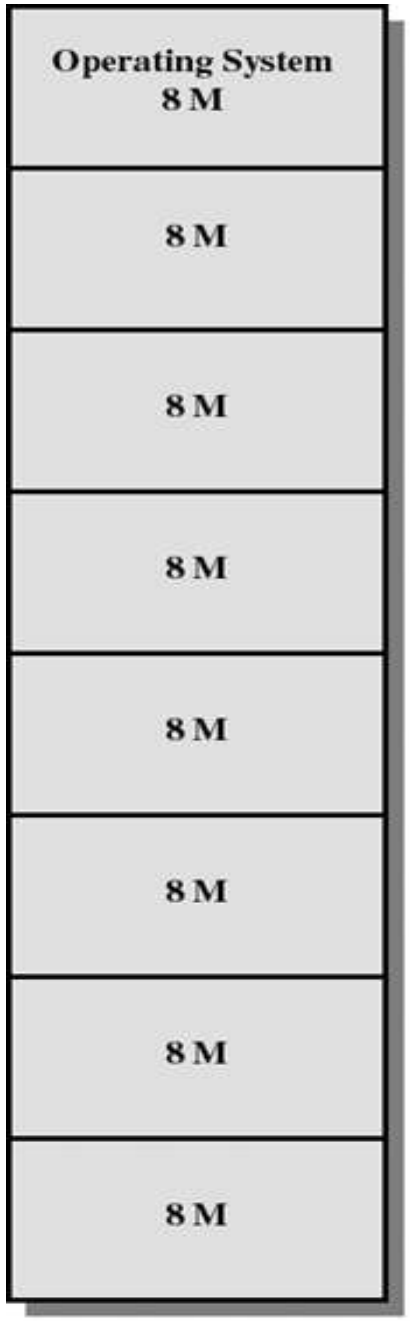


Memory Allocation – Fixed sized partitions

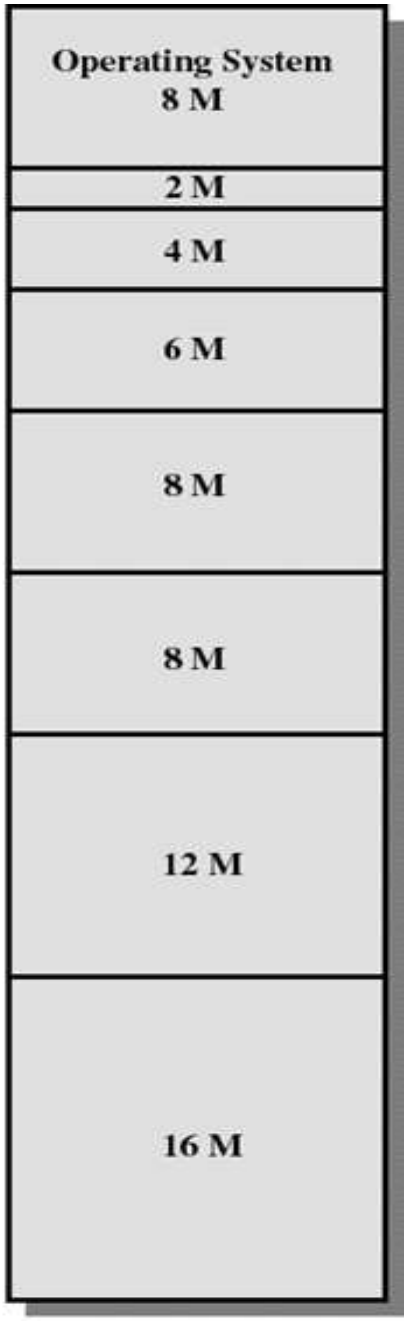
- One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions.
- Each partition may contain exactly one process.
- Thus, the degree of multiprogramming is bound by the number of partitions.
- In this when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.

Fixed partitions can be of equal or unequal sizes

This kind of partitions no longer in use as it suffers from internal fragmentation



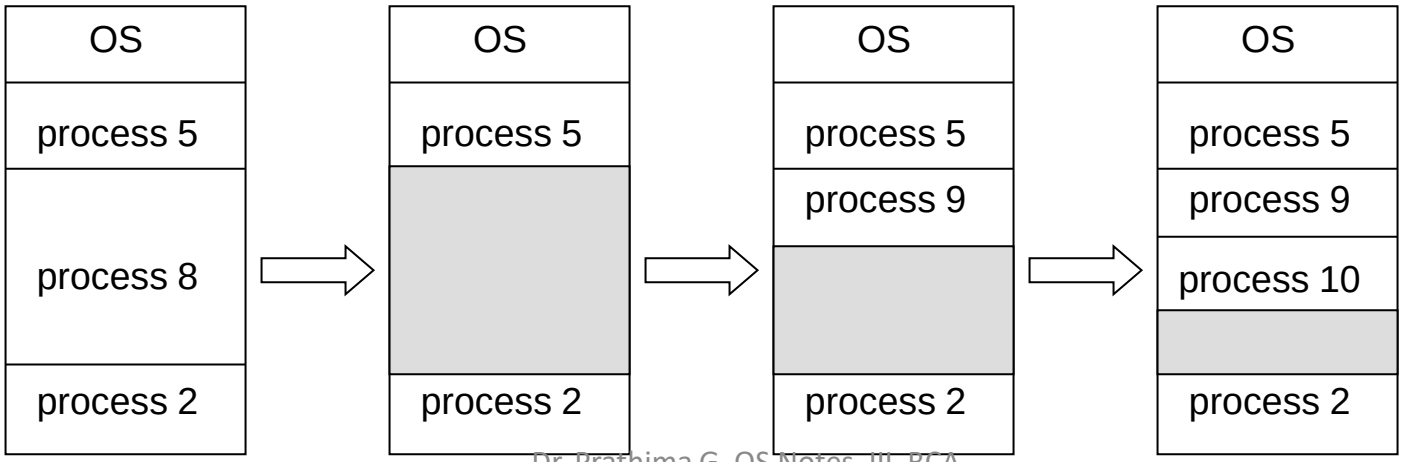
Equal-size partitions



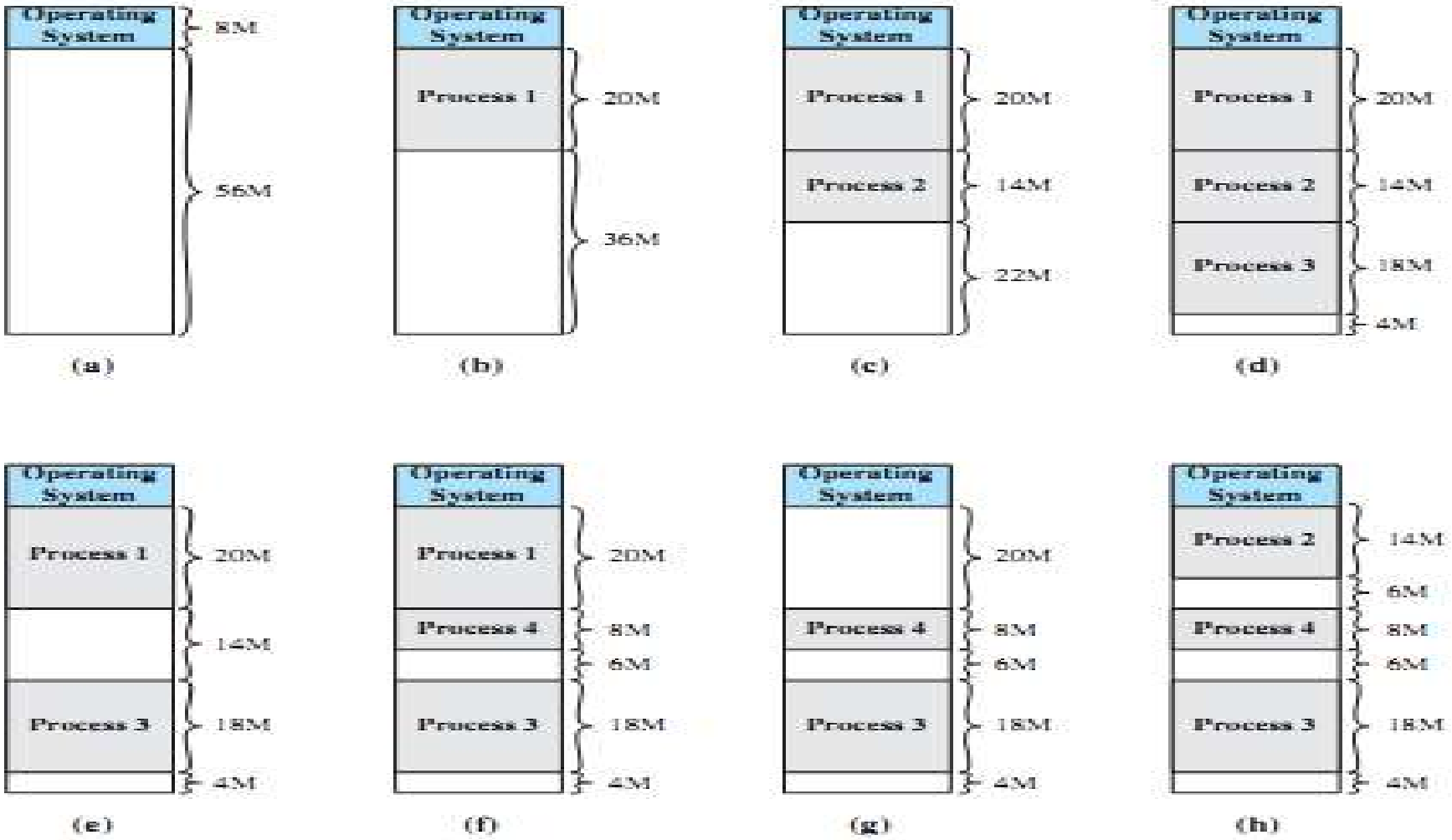
Unequal-size partitions

Memory Allocation – Variable sized partitions

- In the scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory a hole.
- Eventually, memory contains a set of holes of various sizes



Variable Partitioning: example



Variable sized partitions

- The memory blocks available comprise a set of partitions of various sizes scattered throughout memory.

- **When a process arrives and needs memory, the system searches the set for a partition that is large enough for this process.**

- **If the partition is too large, it is split into two parts.**

- **One part is allocated to the arriving process; the other is returned to the set of free partitions.**

Variable sized partitions

- **When a process terminates, it releases its block of memory, which is then placed back in the set of partitions.**

- **If the new partition is adjacent to other partitions, these adjacent partitions are merged to form one larger partition.**

- At this point, the system may need to check whether there are processes waiting for memory and

- whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

Dynamic Storage-Allocation Problem

- This procedure is to select a free partition from the set of available free partitions.

- Three main strategies are

- First Fit
- Best Fit
- Worst Fit

First Fit

- **Allocate the first partition that is big enough.**
- **Searching can start either at the beginning of the set of partitions or**
- **at the location where the previous first-fit search ended.**
- **We can stop searching as soon as we find a free partition that is large enough.**

Best Fit

- **Allocate the smallest partition that is big enough.**
- **Search the entire list, unless the list is ordered by size.**
- **This strategy produces the smallest leftover partition**

Worst Fit

- **Allocate the largest partition.**
- **Again, we must search the entire list, unless it is sorted by size.**
- **This strategy produces the largest leftover partition**

Comparison

- Both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.
- First fit is generally faster.
- Best fit gives better space utilization

Fragmentation

- Wastage of memory space due to fixed sized and variable sized partitions in memory

- Two types

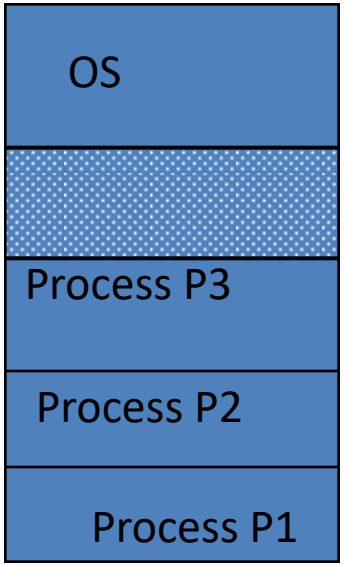
1. Internal Fragmentation

2.External Fragmentation

Internal Fragmentation

A program may not fit in a partition

Any program, no matter how small, occupies an entire partition.

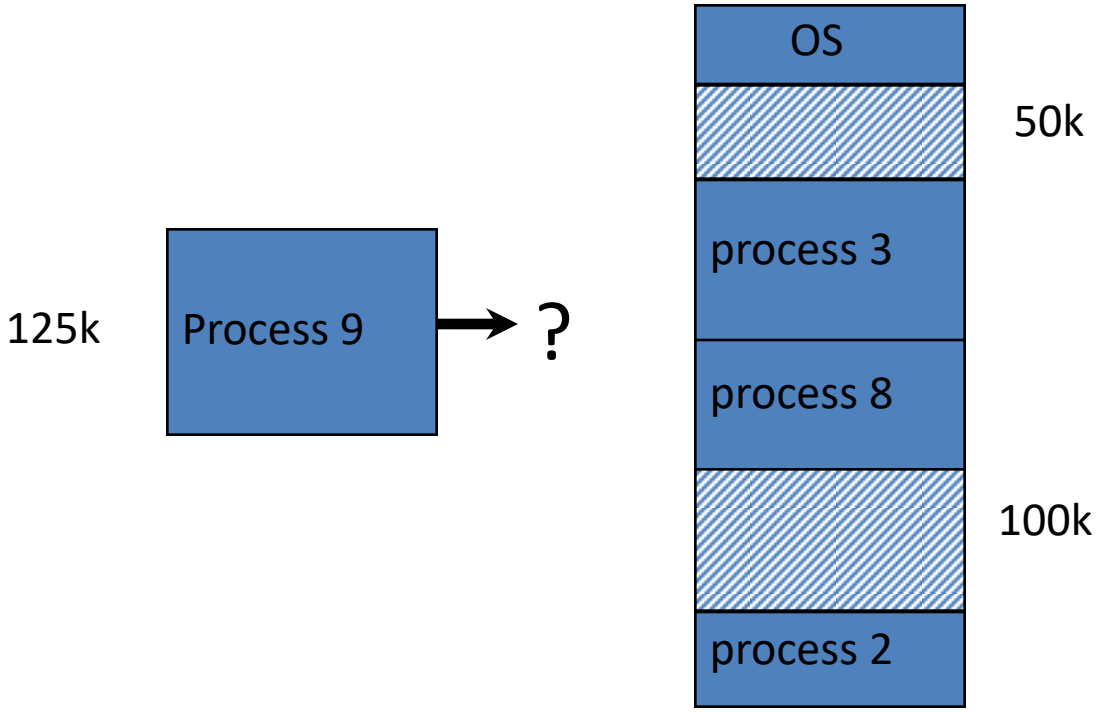


- Have some “empty” space for each processes

- **Internal Fragmentation** - allocated memory may be slightly larger than requested memory ;
- this size difference is memory internal to a partition, but not being used.

External Fragmentation

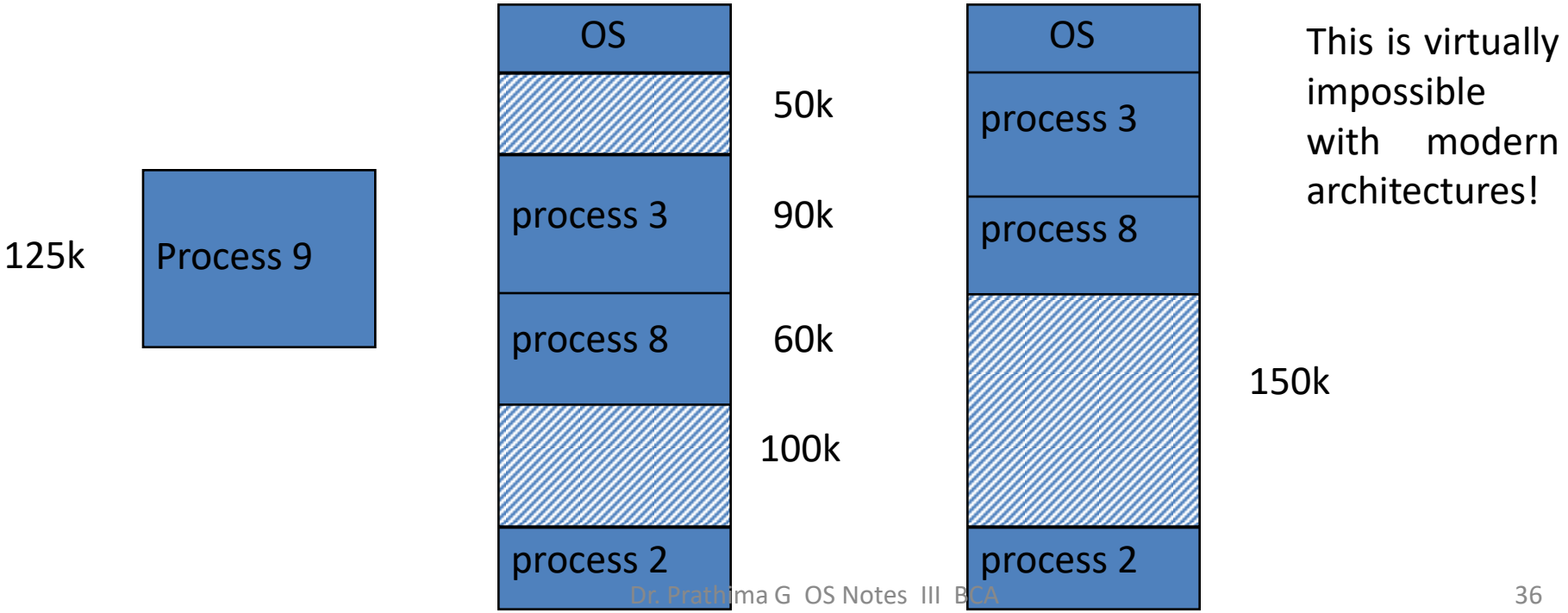
- External Fragmentation - total memory space exists to satisfy request but it is not contiguous



Memory external to all processes is fragmented

Compaction

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

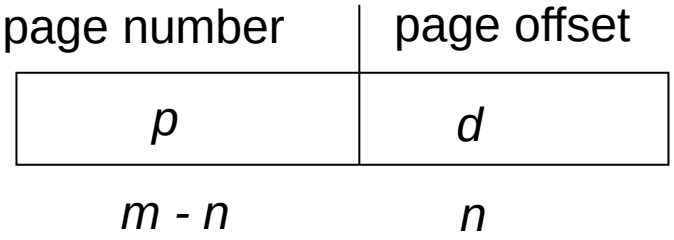


Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- **Divide physical memory into fixed-sized blocks called frames**
  - **Size is power of 2, between 512 bytes and 16 Mbytes**
- **Divide logical memory into blocks of same size called pages**
- **Keep track of all free frames**
- **To run a program of size *N* pages, need to find *N* free frames and load program**
- **Set up a page table to translate logical to physical addresses**
- **Backing store likewise split into pages**
- Still have Internal fragmentation(only in the last page)

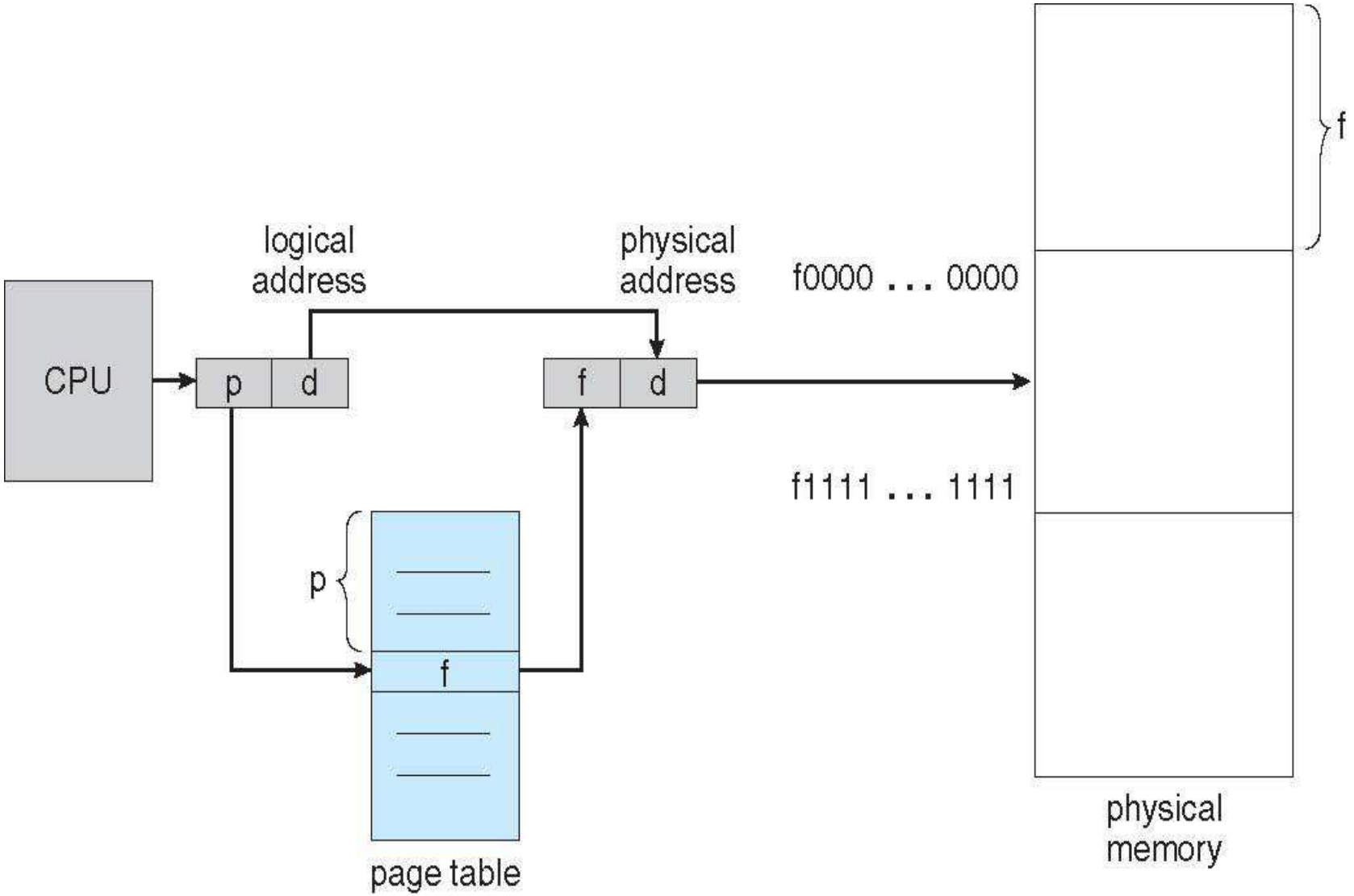
Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset (*d*)** – combined with base address to define the physical memory address that is sent to the memory unit

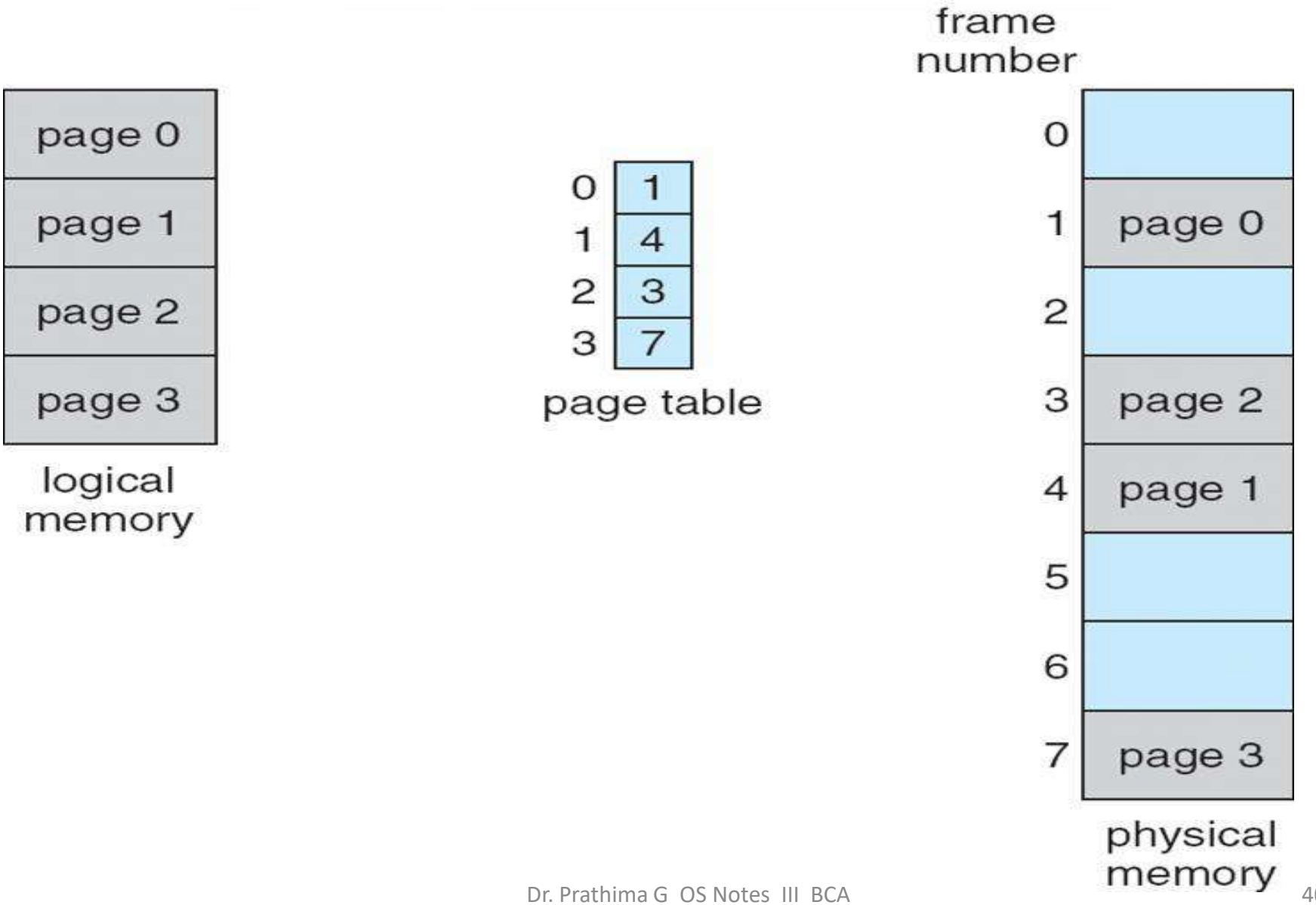


  - If the size of the logical address space is 2^m, and a page size is 2^n addressing units (bytes or words) then
  - the high-order m- n bits of a logical address designate the page number, and
  - the n low-order bits designate the page offset.

Paging Hardware



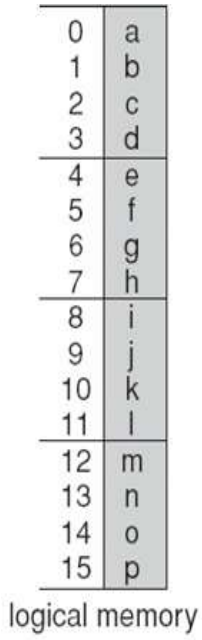
Paging Model of Logical and Physical Memory

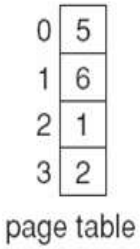


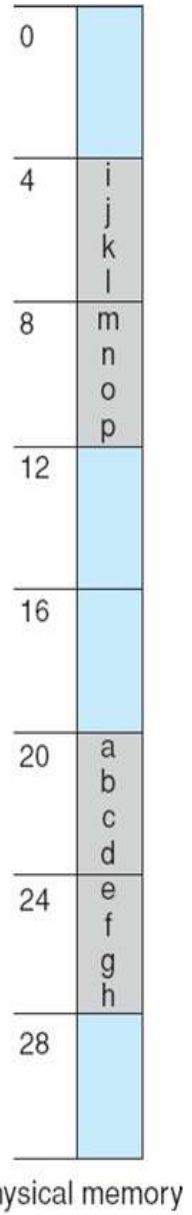
Paging Example – mapping logical address to Physical address

- Logical address, n= 2 and m = 4.
- Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages),
- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5.
- Thus, logical address 0 maps to physical address 20 [= (5 x 4) + 0].
- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 x 4) + 3].

- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6.
- Thus, logical address 4 maps to physical address 24 [= (6 x 4) + O].
- Logical address 13 maps to physical address 9.







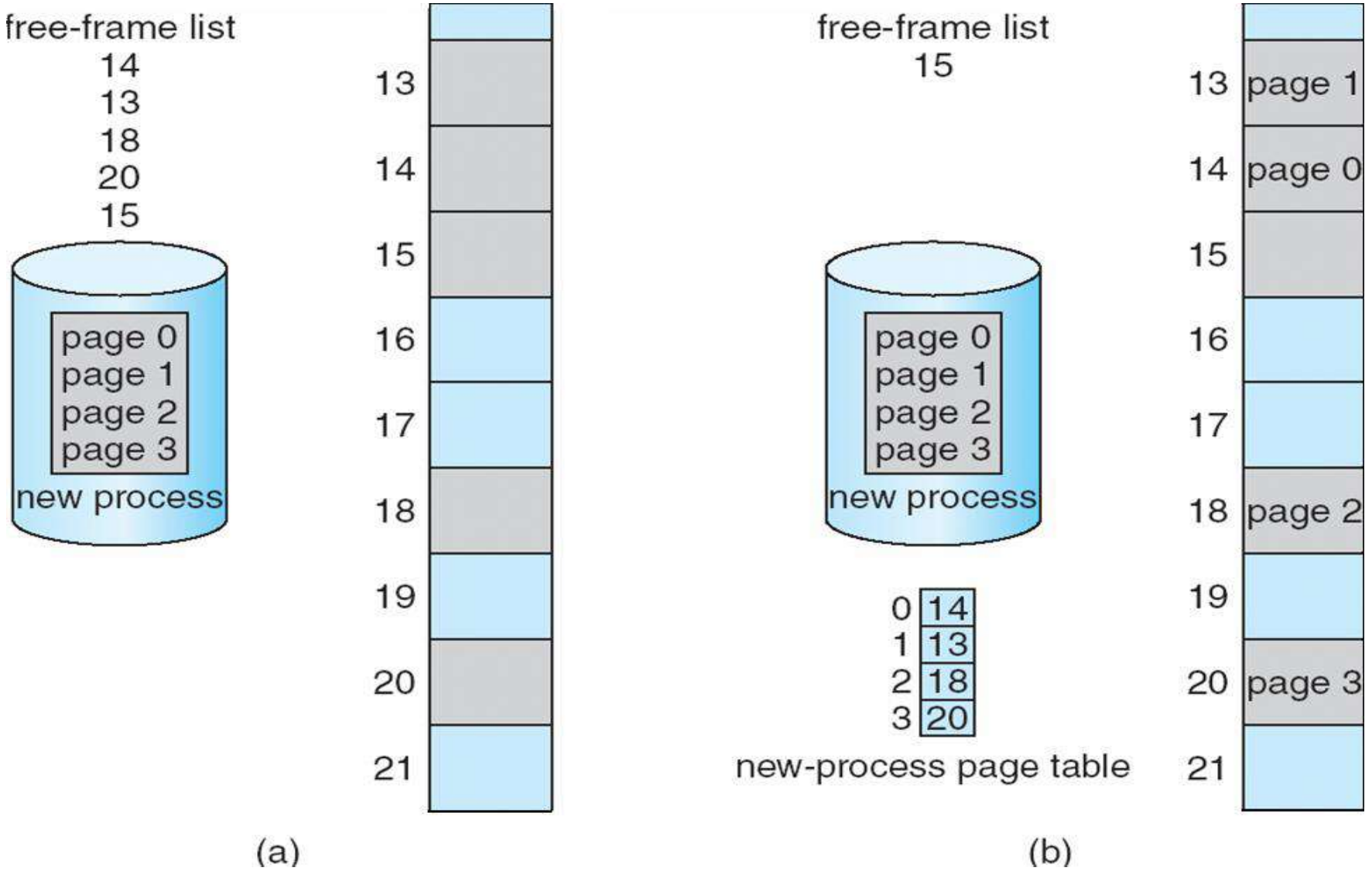
Paging Concept

- Paging is a form of dynamic relocation
- **No external fragmentation: any free frame can be allocated to a process that needs it.**
- **However, we may have some internal fragmentation in the last frame**
- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size

Allocating Free Frames

- **When a process arrives in the system to be executed, its size, expressed in pages, is examined.**
- **Each page of the process needs one frame.**
- **Thus, if the process requires n pages, at least n frames must be available in memory.**
- **If n frames are available, they are allocated to this arriving process.**
- **The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process.**
- **The next page is loaded into another frame, its frame number is put into the page table, and so on**

Free Frames



Before allocation

After allocation

Implementation of Page Table

- Page table is kept in main memory

- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table

- In this scheme every data/instruction access requires two memory accesses
  - **One for the page table and one for the data / instruction**

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

Implementation of Page Table

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry

  – uniquely identifies each process to provide address-space protection for that process

- TLBs typically small (64 to 1,024 entries)

- On a TLB miss, value is loaded into the TLB for faster access next time

  – Replacement policies must be considered
  – Some entries can be **wired down** for permanent fast access

Translation look-aside buffers (TLBs)

- The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is found, its frame number is immediately available and is used to access memory.
- The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used
- If the page number is not in the TLB (known as a a memory reference to the page table must be made.

Translation look-aside buffers (TLBs)

- When the frame number is obtained, we can use it to access memory
- In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.
- If the TLB is already full of entries, the operating system must select one for replacement.
- Replacement policies range from least recently used (LRU) to random.
- Furthermore, some TLBs allow certain entries to be meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are wired down

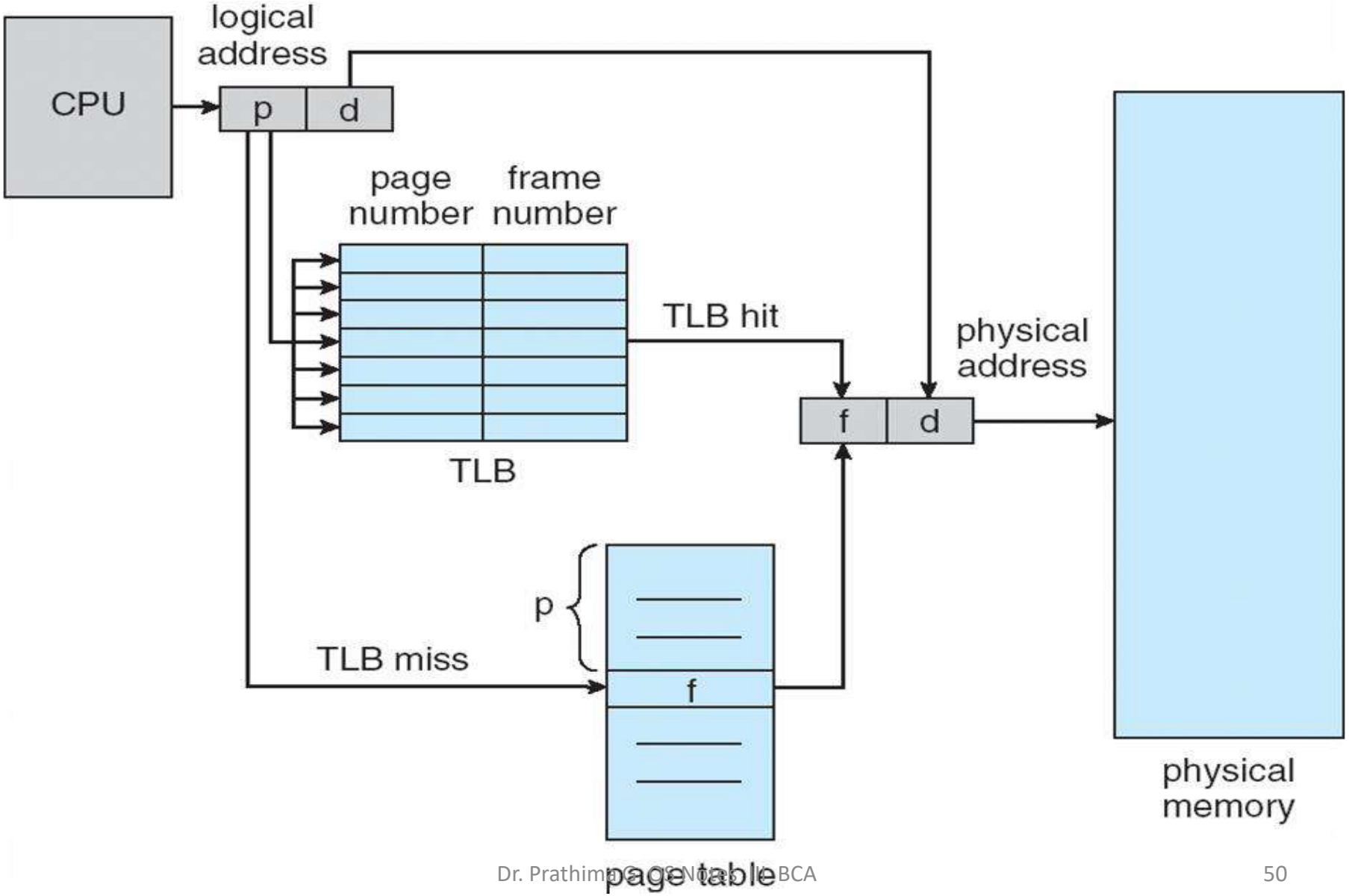
Associative Memory

- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

- The percentage of times that a particular page number is found in the TLB is called the An 80-percent hit ratio,
- for example, means that we find the desired page number in the TLB 80 percent of the time.
- If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory,
- then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB.
- If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and
- then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds.

Effective Access Time

- To find the effective we weight the case by its probability:
- effective access time = 0.80 x 120 + 0.20 x 220 = 140 nanoseconds.
- In this example, we suffer a 40-percent slowdown in memory-access time (from 100 to 140 nanoseconds).
- For a 98-percent hit ratio, we have
- effective access time = 0.98 x 120 + 0.02 x 220 = 122 nanoseconds.
- This increased hit rate produces only a 22 percent slowdown in access time.

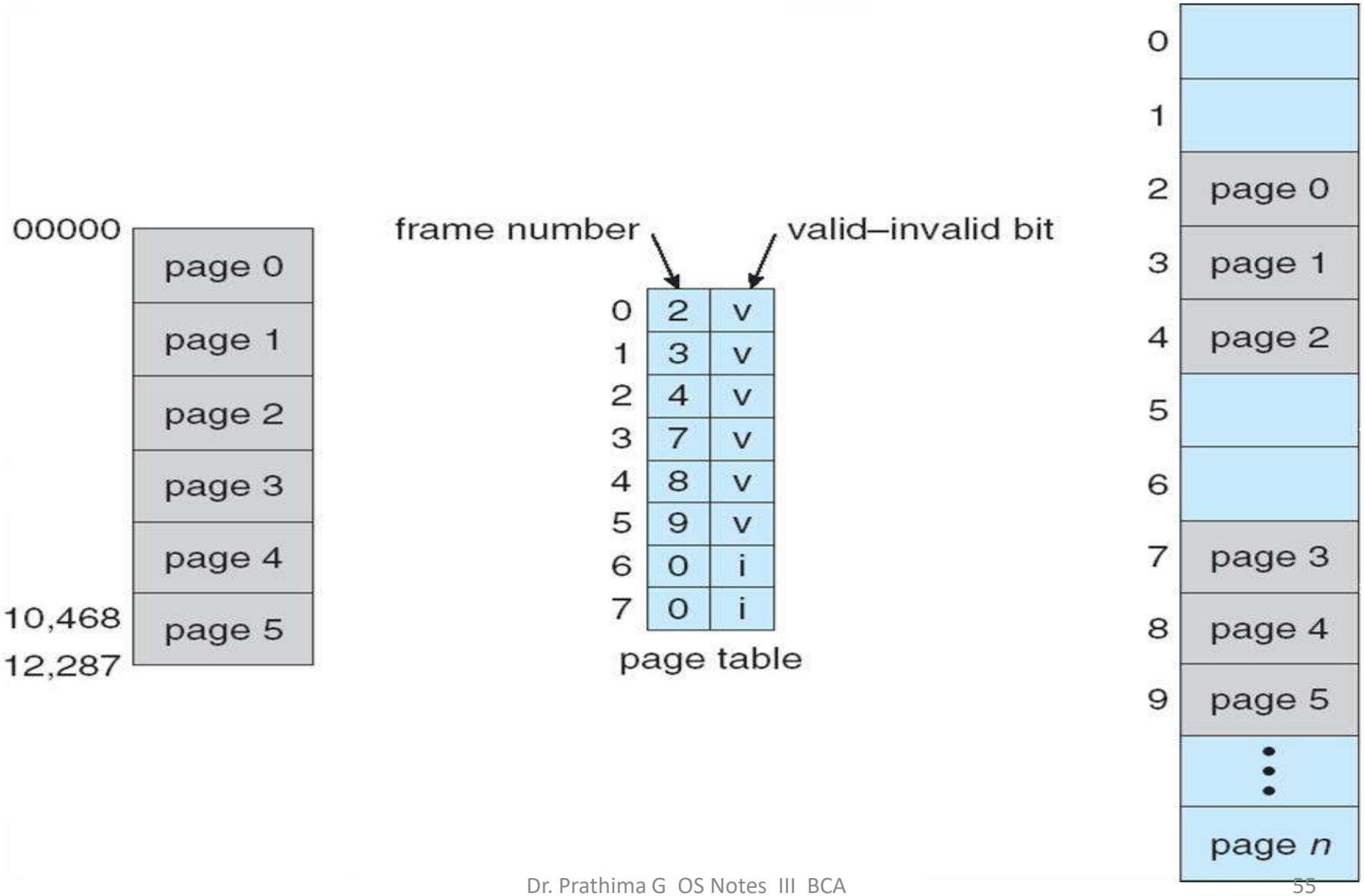
Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use PTLR
- Any violations result in a trap to the kernel

Example

- Suppose, for example, that in a system with a 14-bit address space (0 to 16383),
- we have a program that should use only addresses 0 to 10468.
- Given a page size of 2 KB, Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table.
- Any attempt to generate an address in pages 6 or 7, however, will find that the valid -invalid bit is set to invalid, and
- the computer will trap to flee operating system (invalid page reference).

Valid (v) or Invalid (i) Bit In A Page Table



Hardware Protection

- Some systems provide hardware**, in the form of a page table length register(PTLR) length to indicate the size of the page table.**

- value is **checked against every logical address to verify that the address is in the valid range for the process.**

- **Failure of this test causes an error trap to the operating**

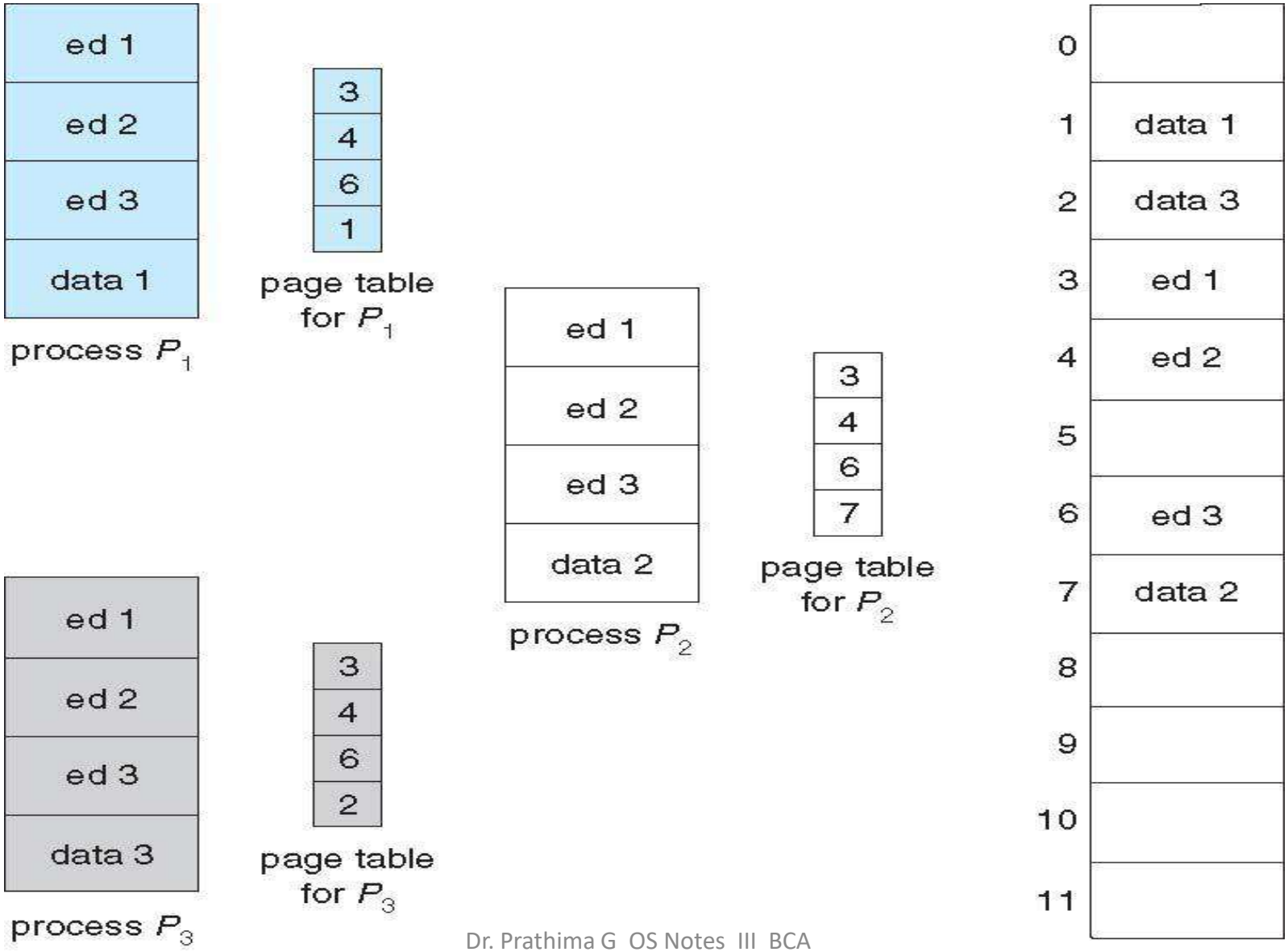
Shared Pages

- An advantage of paging is the possibility of sharing common code.
- This consideration is particularly important in a time-sharing environment.
- Consider a system that supports 40 users, each of whom executes a text editor.
- If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.

Shared Pages

- If the code is reentrant (or pure code) however, it can be shared, as shown in Figure.
- Here we see a three-page editor-each page 50 KB in size being shared among three processes.
- Each process has its own data page.
- Reentrant code is non-self-modifying code: it never changes during execution.
- Thus, two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data storage to hold the data for the process's execution.

Shared Pages Example



Shared Pages

- Only one copy of the editor need be kept in physical memory
- Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.
- Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user.
- The total space required is now 2150 KB instead of 8,000 KB-a significant savings.
- Other heavily used programs can also be shared -compilers, window systems, run-time libraries, database systems, and so on.
- To be sharable, the code must be reentrant.

Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don't want to allocate that contiguously in main memory
- One simple solution to this problem is to divide the page table into smaller pieces.
- We can accomplish this division in several ways

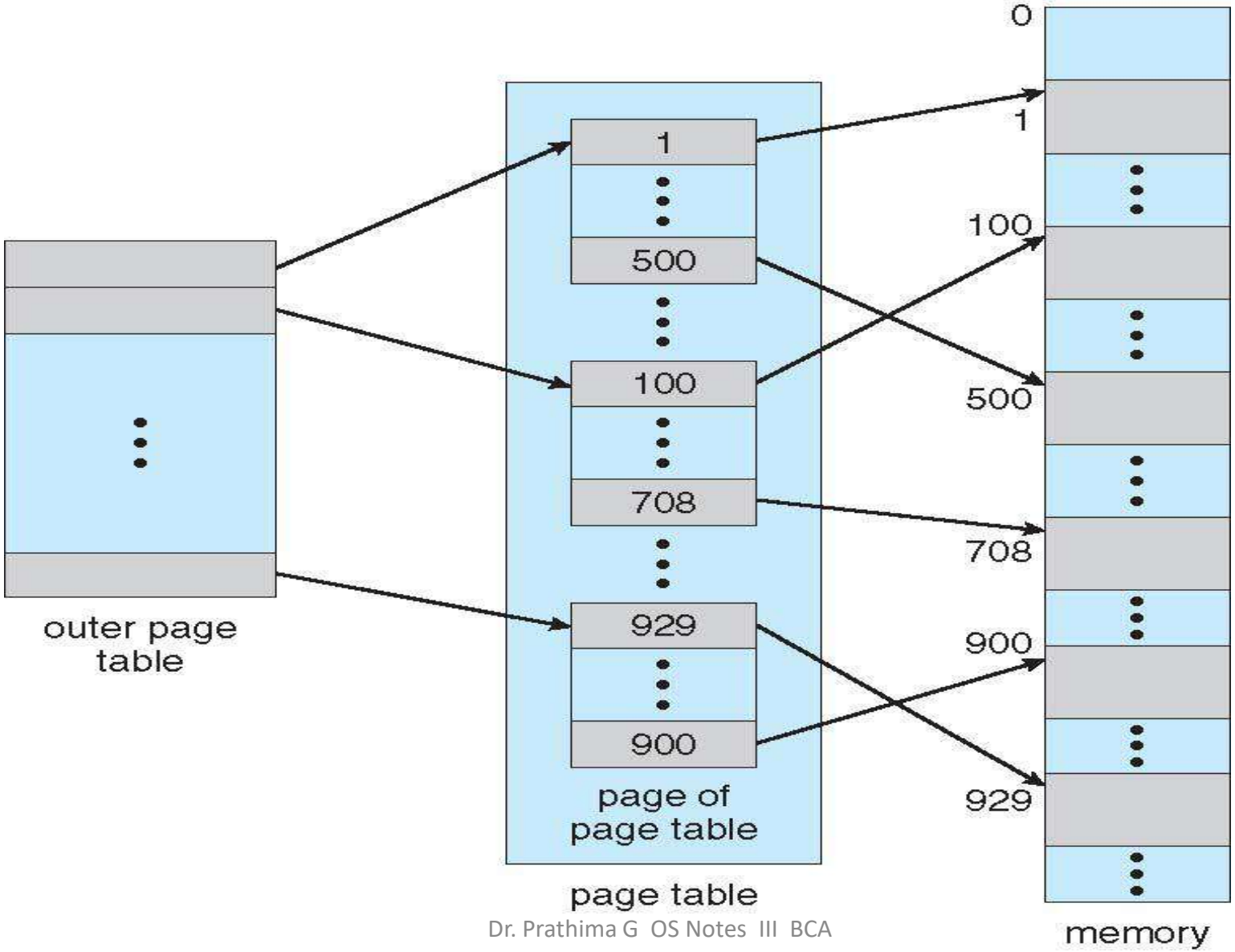
Structure of the Page Table

- Hierarchical Page Tables
- Hashed Page Tables
- Inverted Page Tables
- Hierarchical Paging-One way is to use a two-level paging algorithm,
- in which the page table itself is also paged .

Hierarchical Page Tables

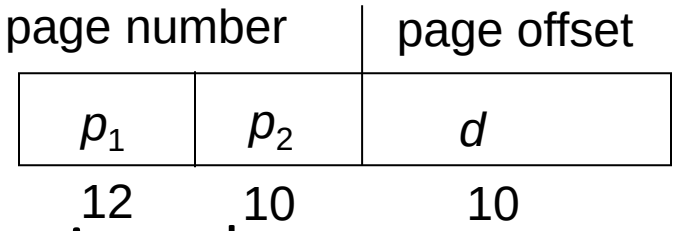
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme



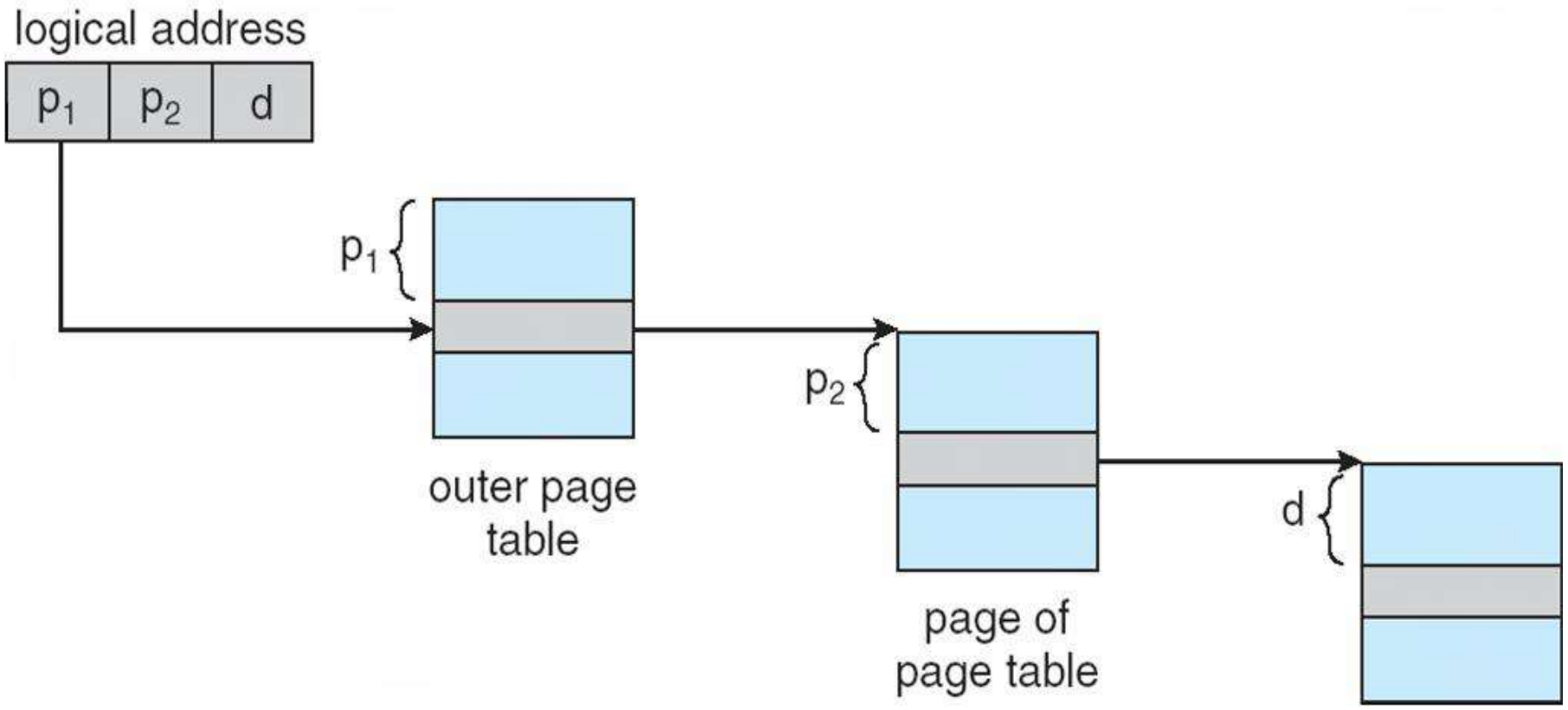
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



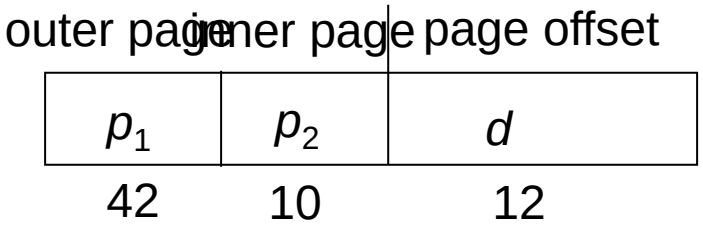
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table
- Known as **forward-mapped page table**

Address-Translation Scheme



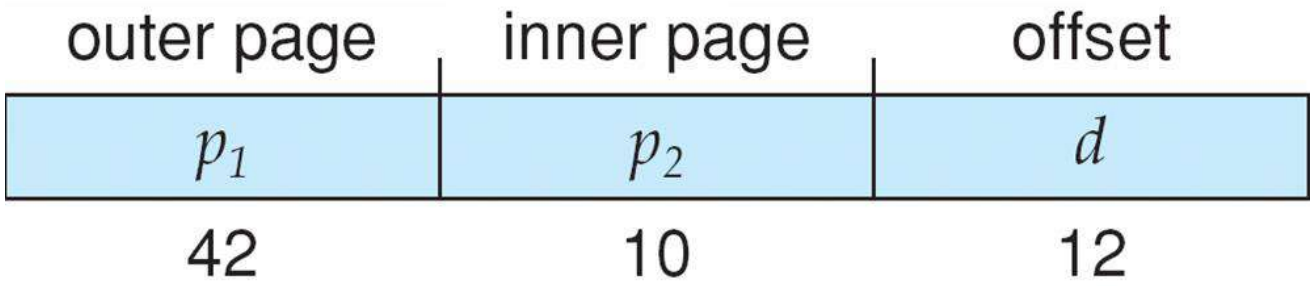
64-bit Logical Address Space

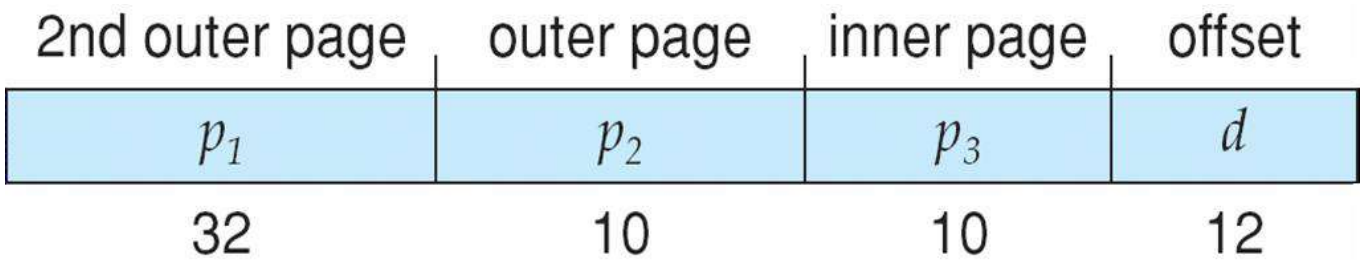
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

Three-level Paging Scheme





Hashed Page Tables

- Common in address spaces > 32 bits

- The virtual page number is hashed into a page table

  - **This page table contains a chain of elements hashing to the same location**

  - **Each entry in the hash table contains a linked list of elements that hash to the same location**

Hashed Page Tables

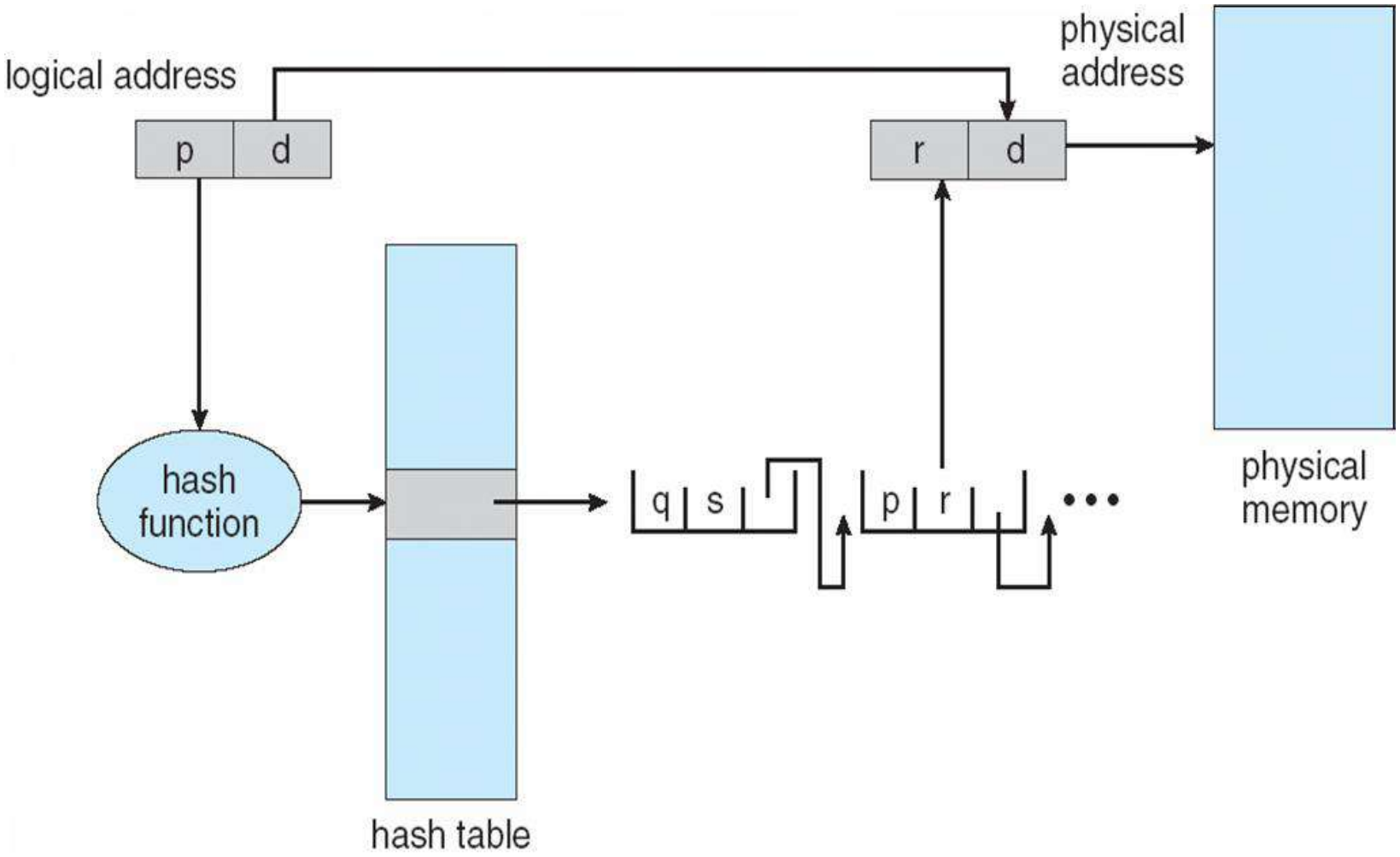
- Each element contains

(1) the virtual page number

(2) the value of the mapped page frame

(3) a pointer to the next element in the linked list

Hashed Page Table



Working

- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number is compared with field 1 in the first element in the linked list.
- If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.
-
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number

Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages

- An inverted page table has one entry for each real page (or frame) of memory

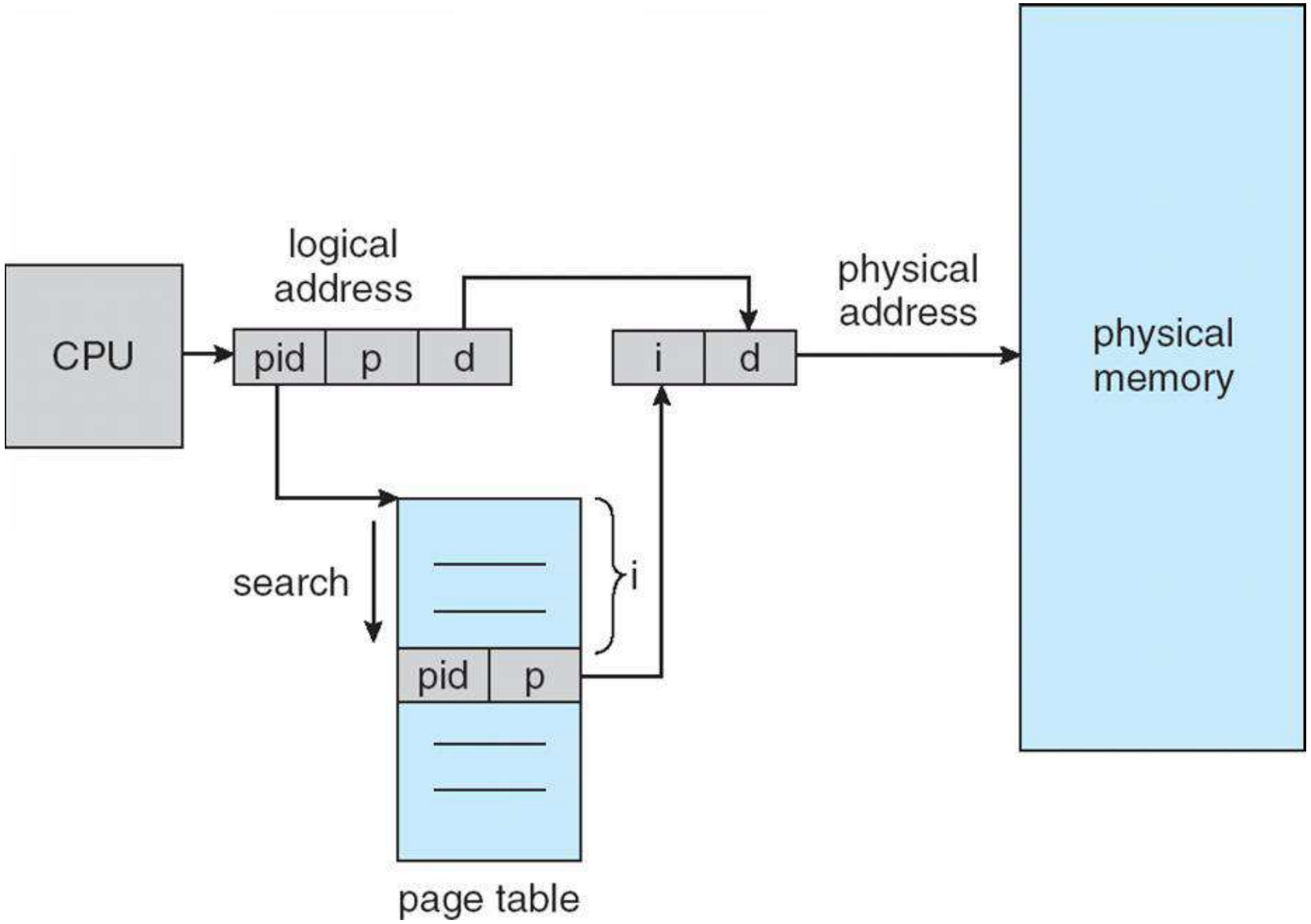
- Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page.

- Thus, only one page table is in the system, and it has only one entry for each page of physical memory

Inverted Page Table

- Each virtual address in the system consists of a triple:
- <process-id, page-number, offset>.
- Each inverted page-table entry is a pair <process-id, page-number>
- where the process-id assumes the role of the address-space identifier

Inverted Page Table Architecture



Inverted Page Table

- When a memory reference occurs,
- part of the virtual address, consisting of <process-id, pagenumber>, is presented to the memory subsystem.
- The inverted page table is then searched for a match.
- If a match is found-say, at entry i-then the physical address <i, offset> is generated.
- If no match is found, then an illegal address access has been attempted.

Inverted Page Table

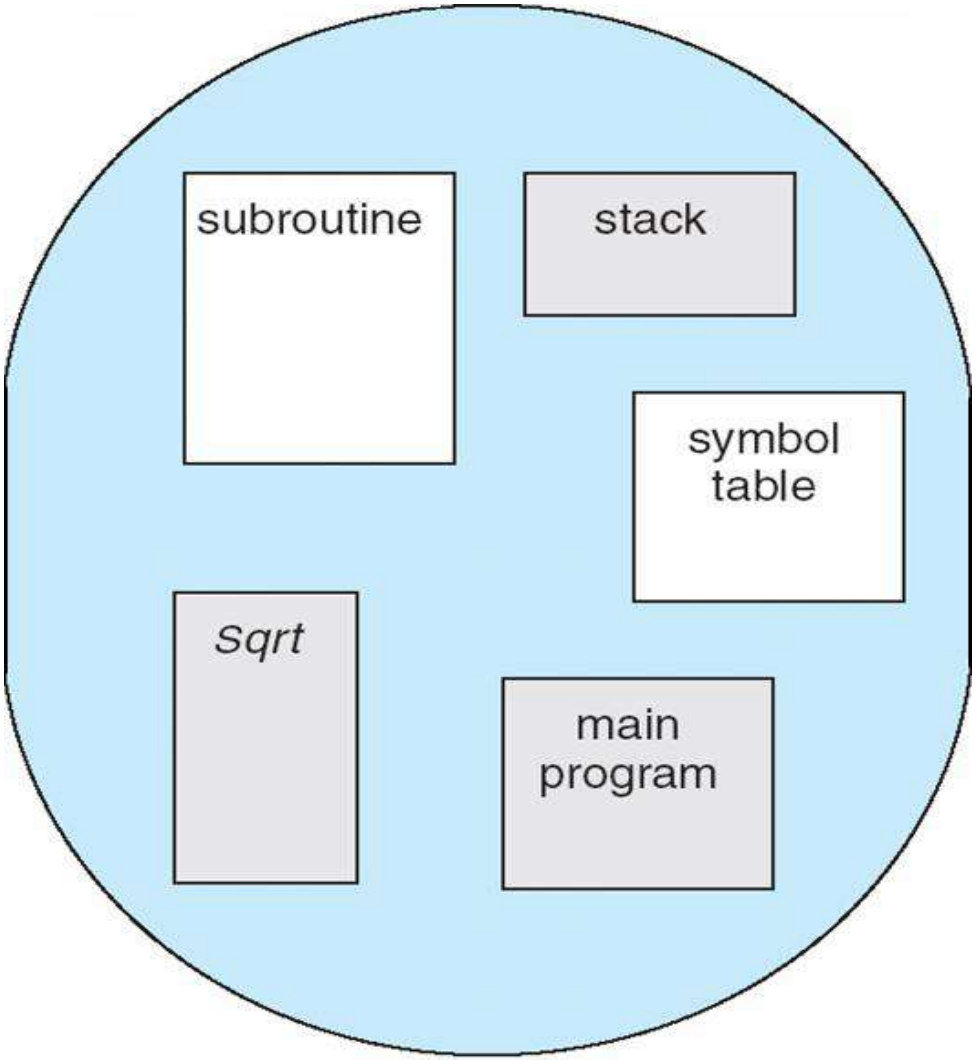
- Although this scheme decreases the amount of memory needed to store each page table,
- it increases the amount of time needed to search the table when a page reference occurs.
- Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched for a match.
- This search would take far too long.
- To alleviate this problem, we use a hash table

Segmentation

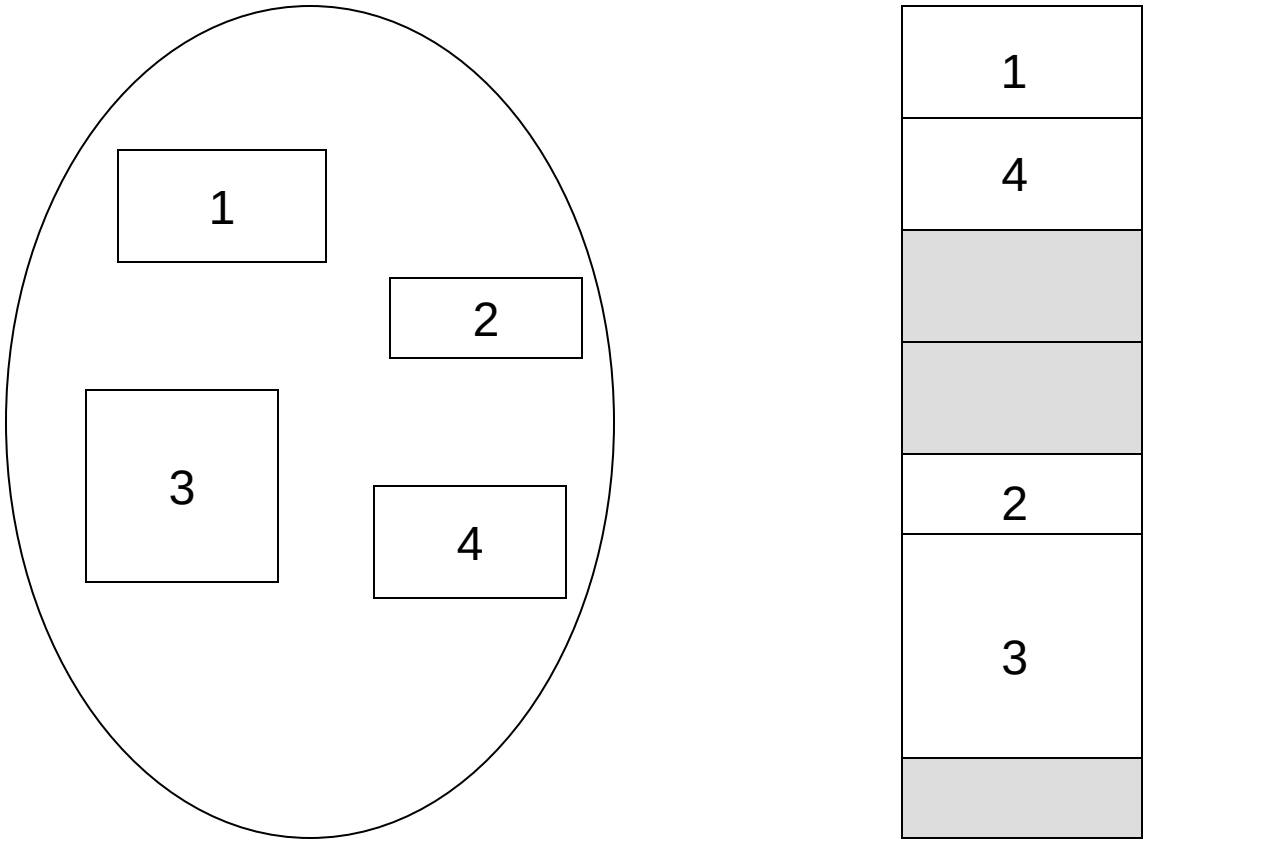
- **Memory-management scheme that supports user view of memory**
- A program is a collection of segments
  - A segment is a logical unit such as:

    main program
    procedure
    function
    method
    object
    local variables, global variables
    common block
    stack
    symbol table
    arrays

User's View of a Program



Logical View of Segmentation



Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

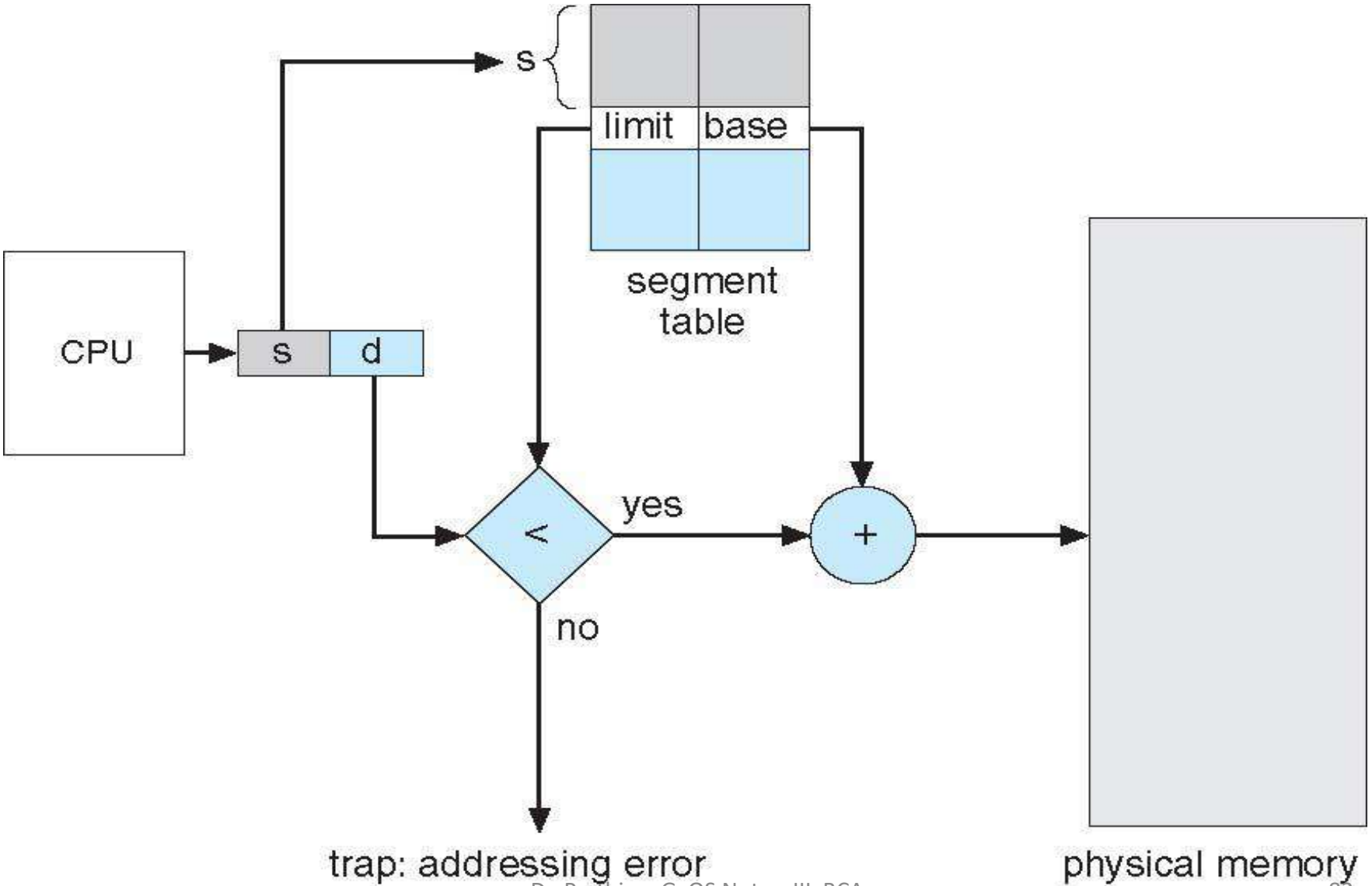
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment

- **Segment-table base register (STBR)** points to the segment table's location in memory

- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

Segmentation Hardware



Segmentation Architecture

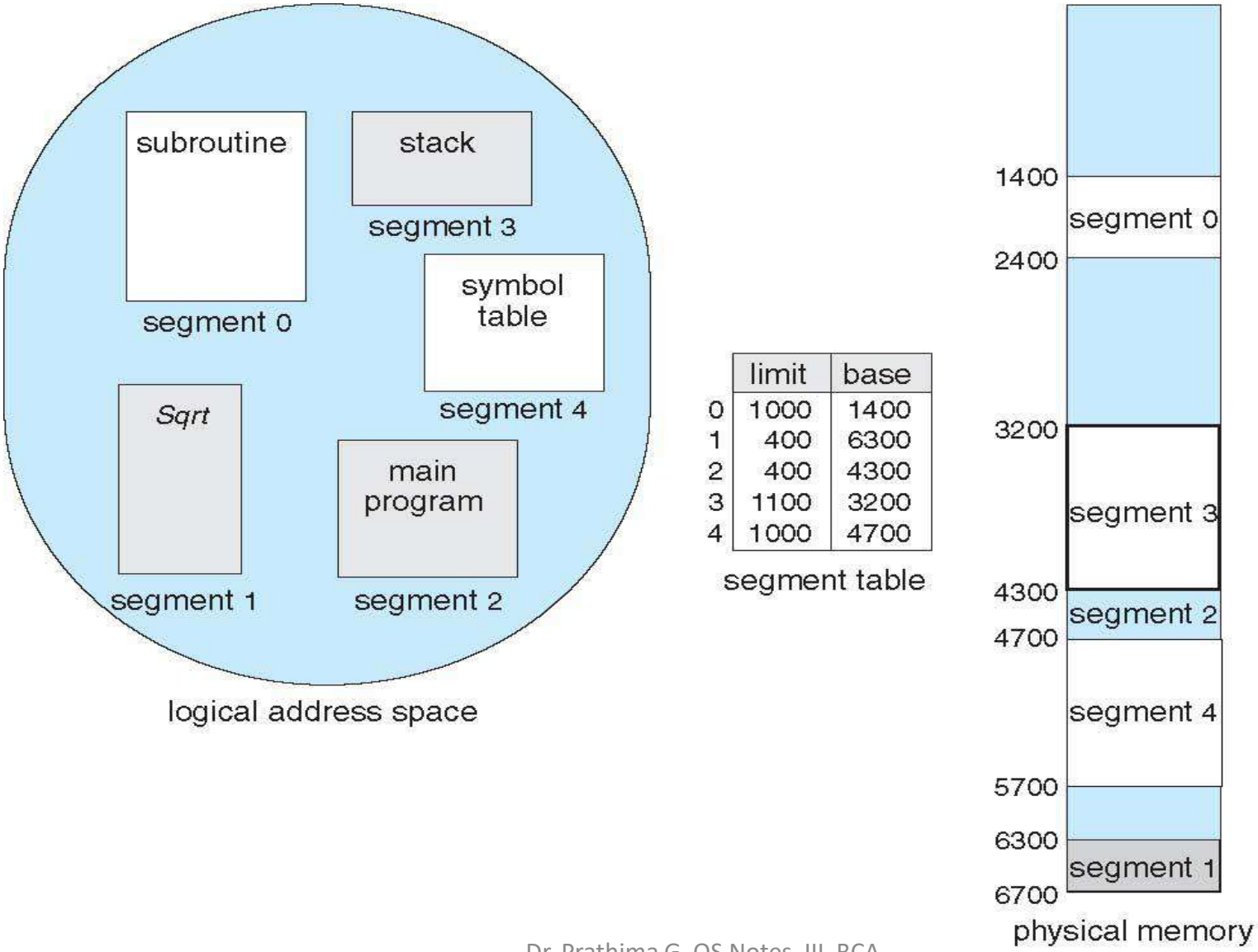
- Protection
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges

- Protection bits associated with segments; code sharing occurs at segment level

- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

- A segmentation example is shown in the following diagram

Example of Segmentation



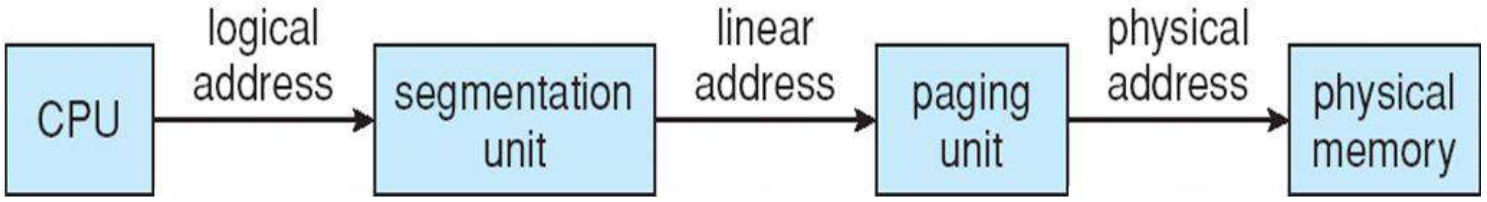
Example: The Intel Pentium

- Both paging and segmentation have advantages and disadvantages.
- In fact some architectures provide both.
- Intel Pentium architecture supports both pure segmentation and segmentation with paging

Example: The Intel Pentium

- In Pentium systems, the CPU generates logical addresses, which are given to the segmentation unit.
- The segmentation unit produces a linear address for each logical address.

- The linear address is then given to the paging unit, which in turn generates the physical address in main memory.
- Thus, the segmentation and paging units form the equivalent of the memory-management unit (MMU).



Pentium Segmentation

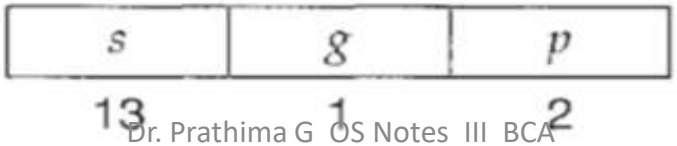
- Each segment can be 4 GB
- Up to 16 K segments per process
- Divided into two partitions
  - First partition of up to 8 K segments are private to process (kept in **local descriptor table LDT**)
  - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table GDT**)

Pentium Segmentation

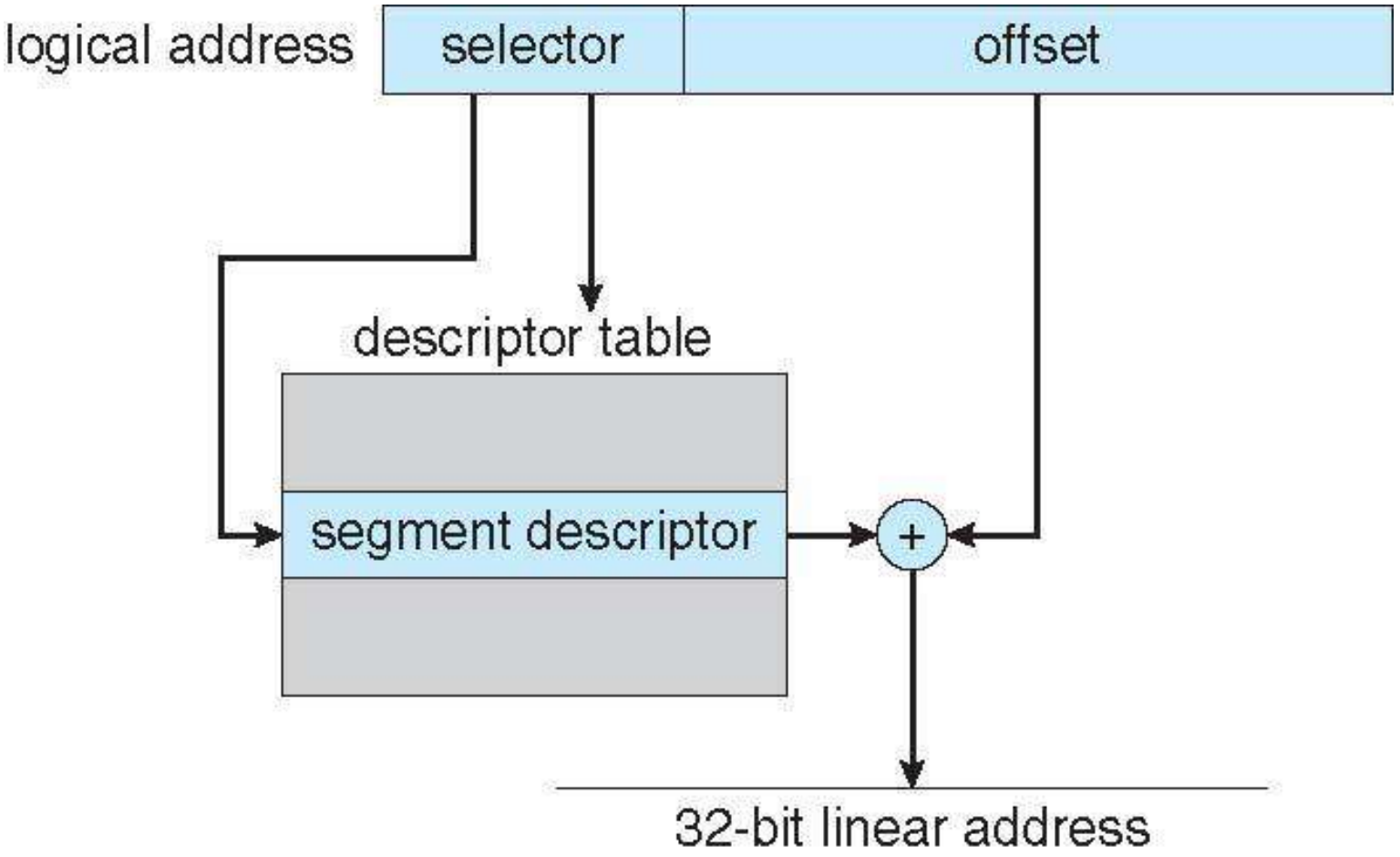
- The logical address is a pair of (selector, offset) where the selector of 16 bits

- In which s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection.

- The offset is a 32-bit number specifying the location of the byte (or word) within the segment



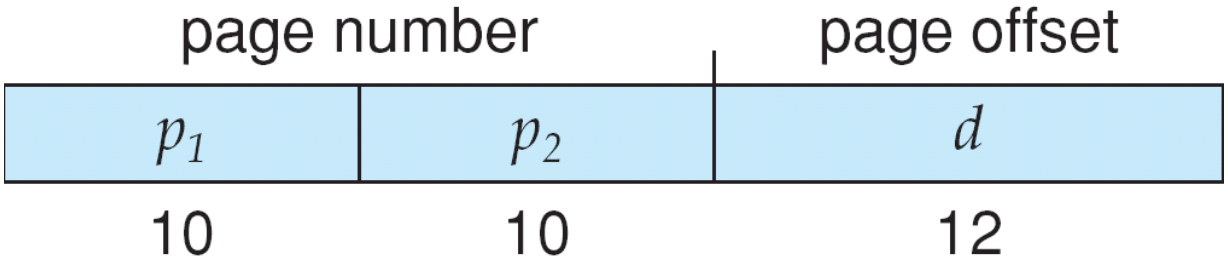
Intel Pentium Segmentation



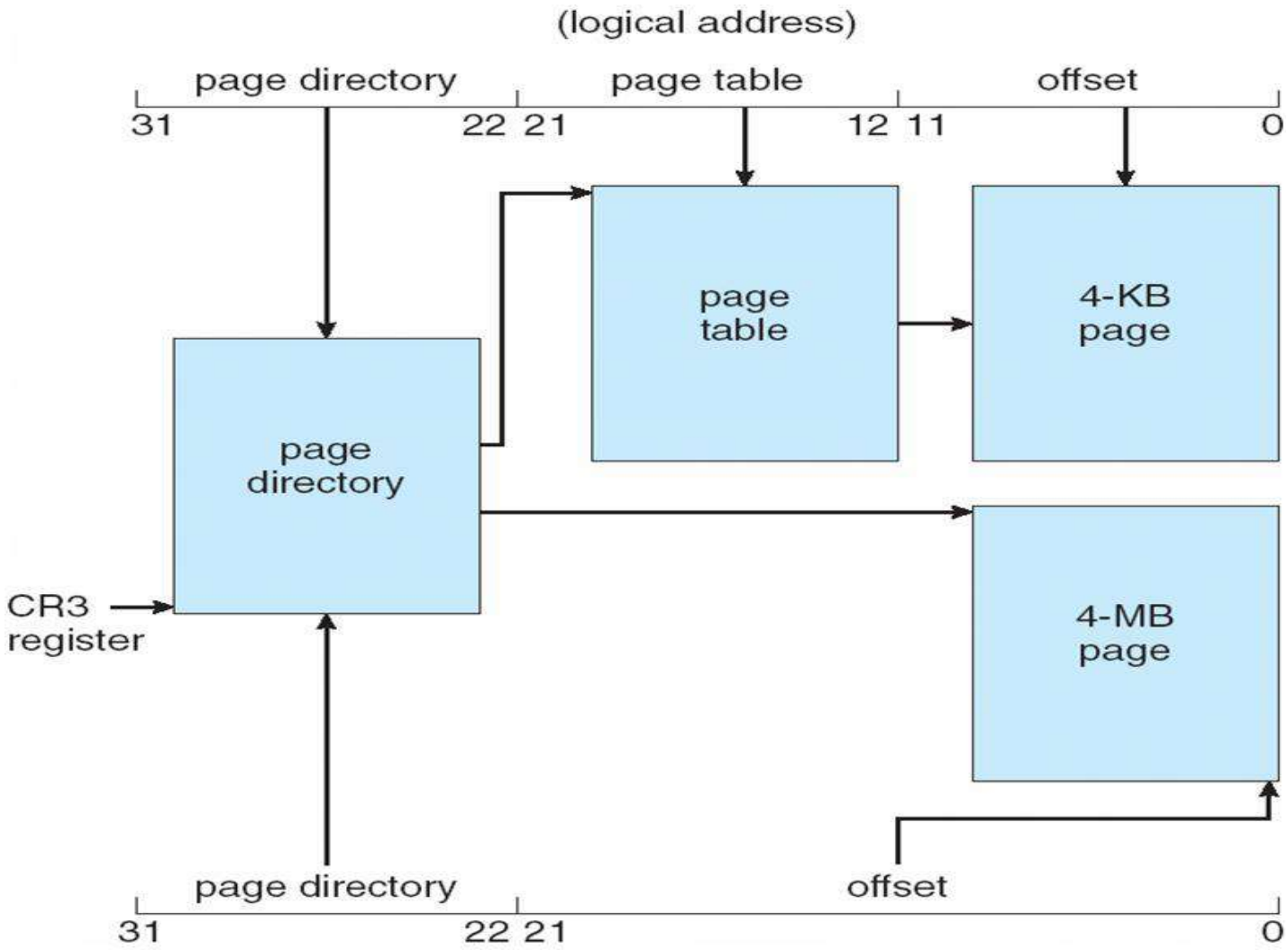
Pentium Paging

The Pentium architecture allows a page size of either 4 KB or 4 MB.

For 4-KB pages, the Pentium uses a two-level paging scheme in which the division of the 32-bit linear address is as follows:



Pentium Paging Architecture



Intel Pentium address translation Mechanism

- The 10 high-order bits reference an entry in the outer most page table, which the Pentium terms the page directory.

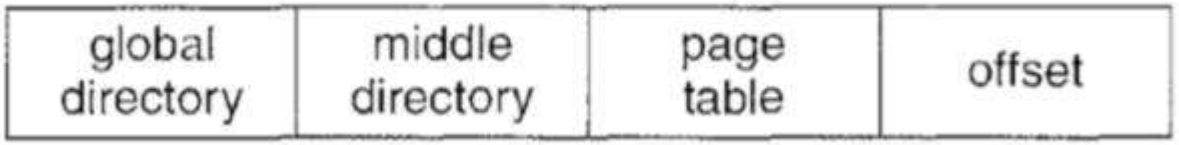
- The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address.

- Finally, the low-order bits 0-11 refer to the offset in the 4-KB page pointed to in the page table.

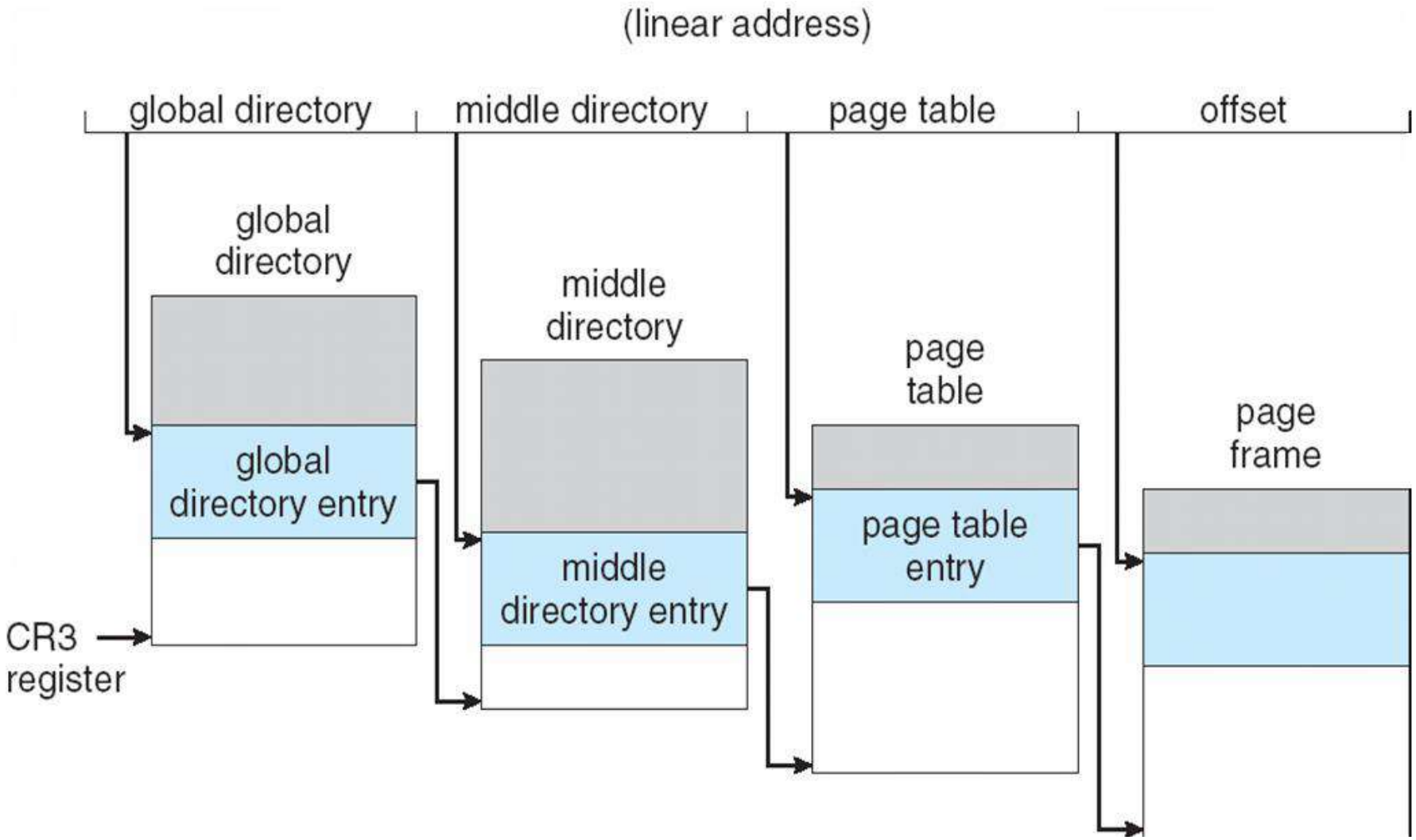
Linux on Pentium Machine

- On the Pentium, Linux uses only six segments:

- A segment for kernel code
- A segment for kernel data
- A segment for user code
- A segment for user data
- A task-state segment (TSS)
- A default LDT segment
- The segments for user code and user data are shared by all processes running in user mode



Three-level Paging in Linux



Virtual Memory

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Program and programs could be larger than physical memory

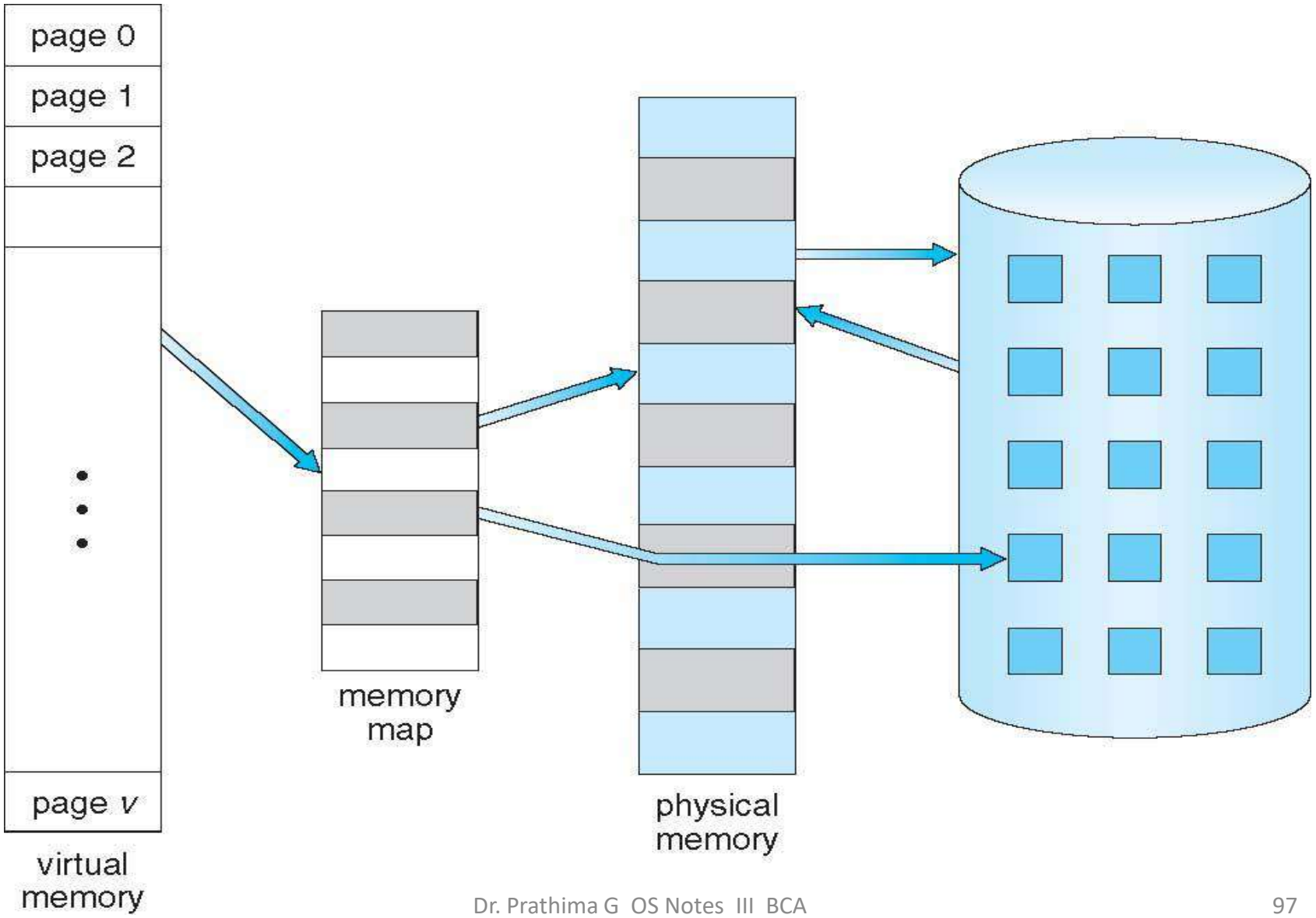
Virtual Memory

- **Virtual memory** – separation of user logical memory from physical memory

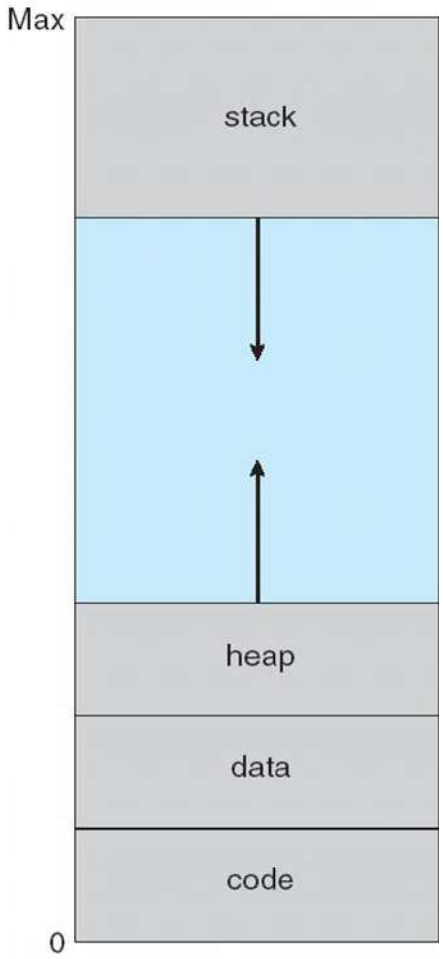
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory



Virtual-address Space



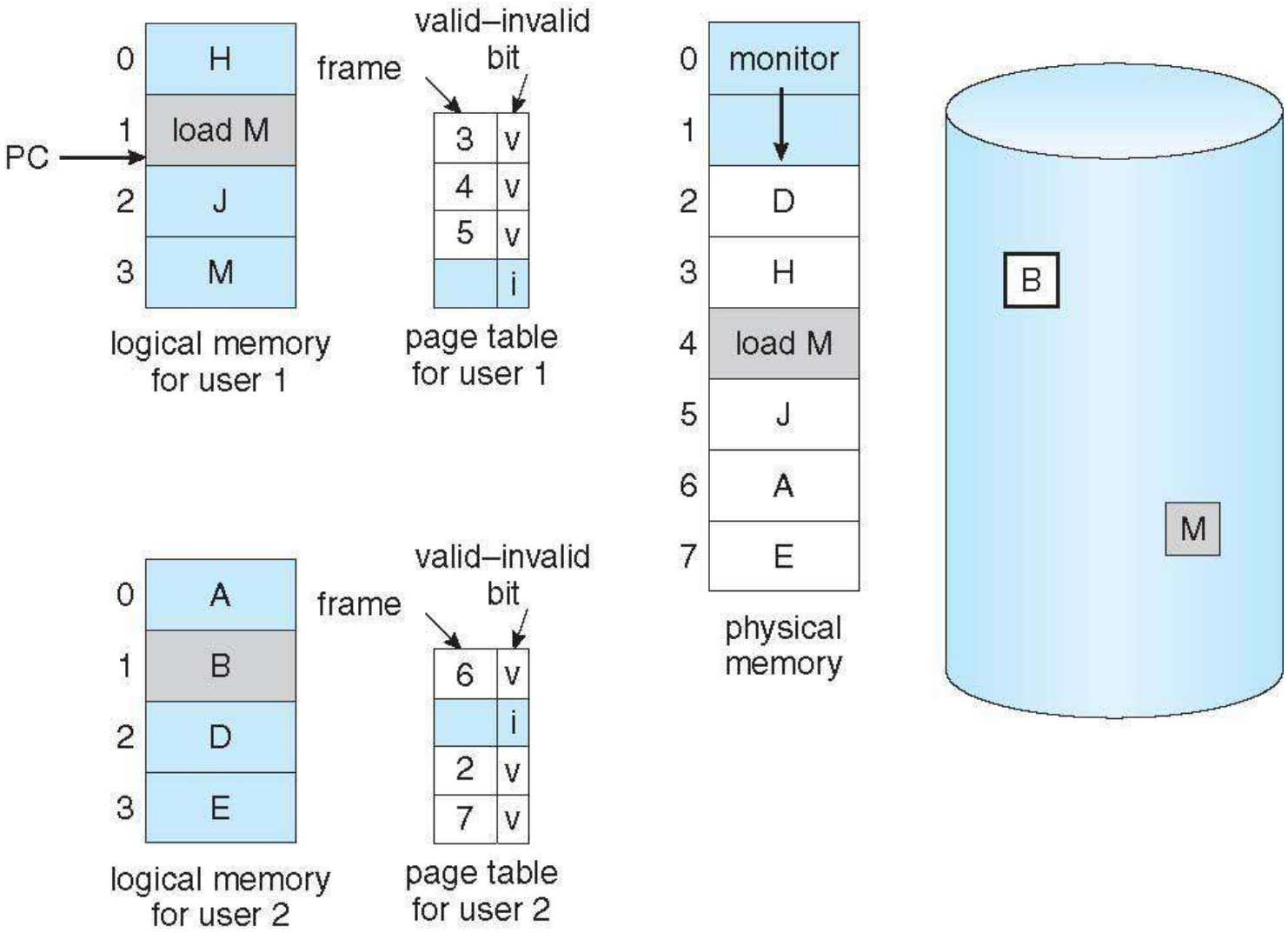
Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

- Page replacement completes separation between logical memory and physical memory –

- large virtual memory can be provided on a smaller physical memory

Need For Page Replacement



Basic Page Replacement

1. Find the location of the desired page on disk

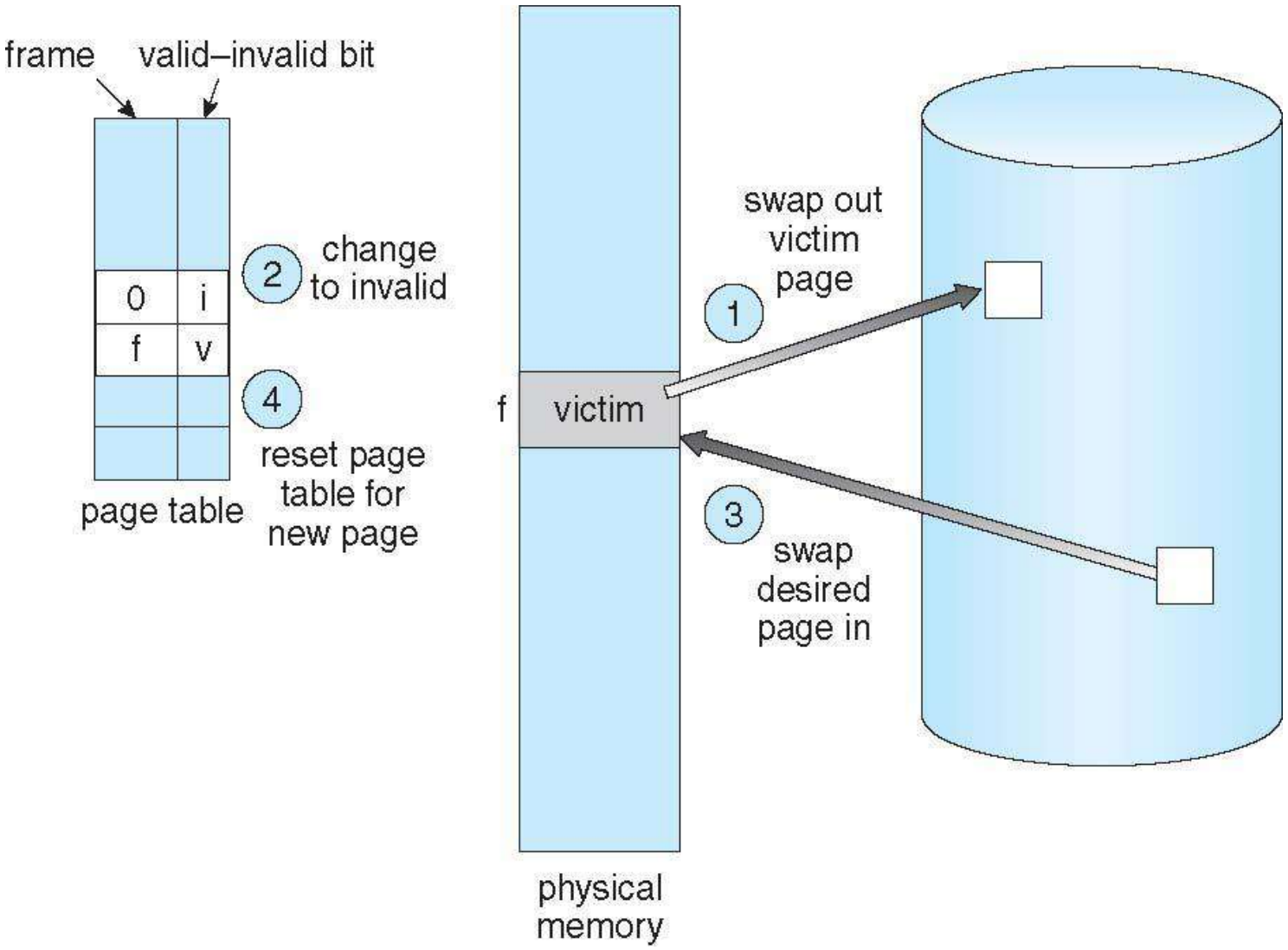
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing Effective Access Time

Page Replacement



Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace

- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
- In all our examples, the reference string is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace

- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access

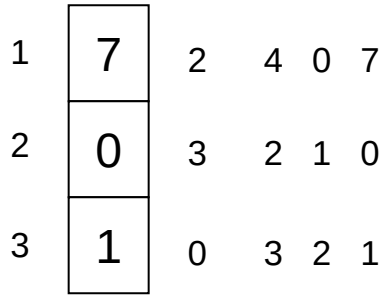
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
- In all our examples, the reference string is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

First-In-First-Out (FIFO) Algorithm

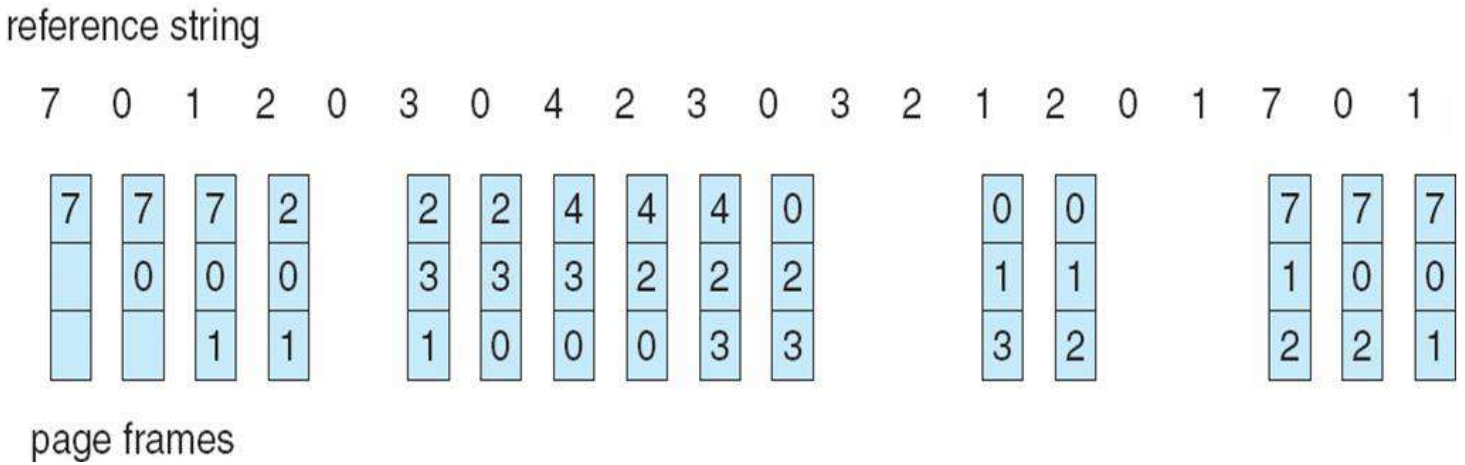
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

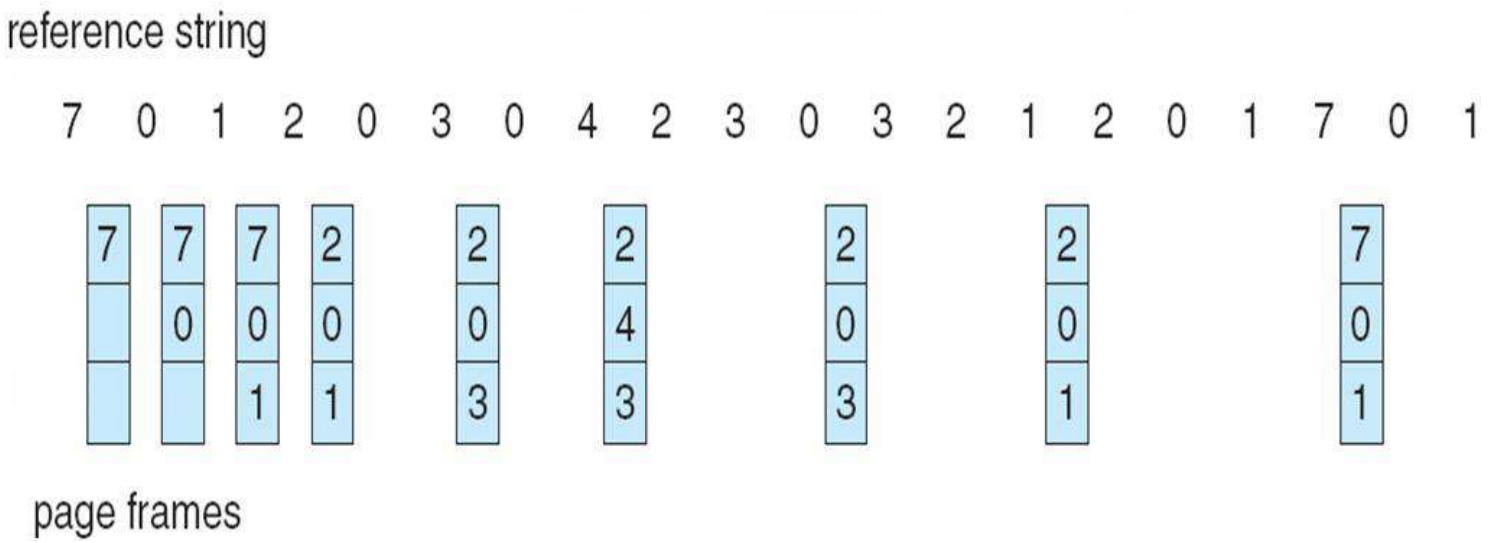
FIFO Page Replacement



Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example on the next slide
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

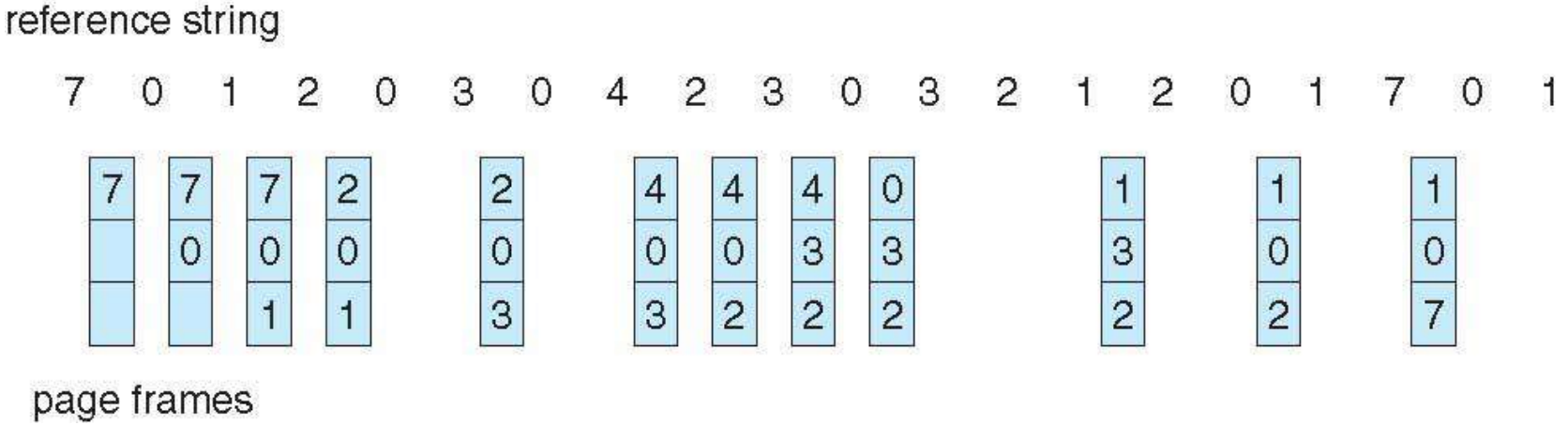
Optimal Page Replacement



No of page faults = 9

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used

Page Replacement Exercise

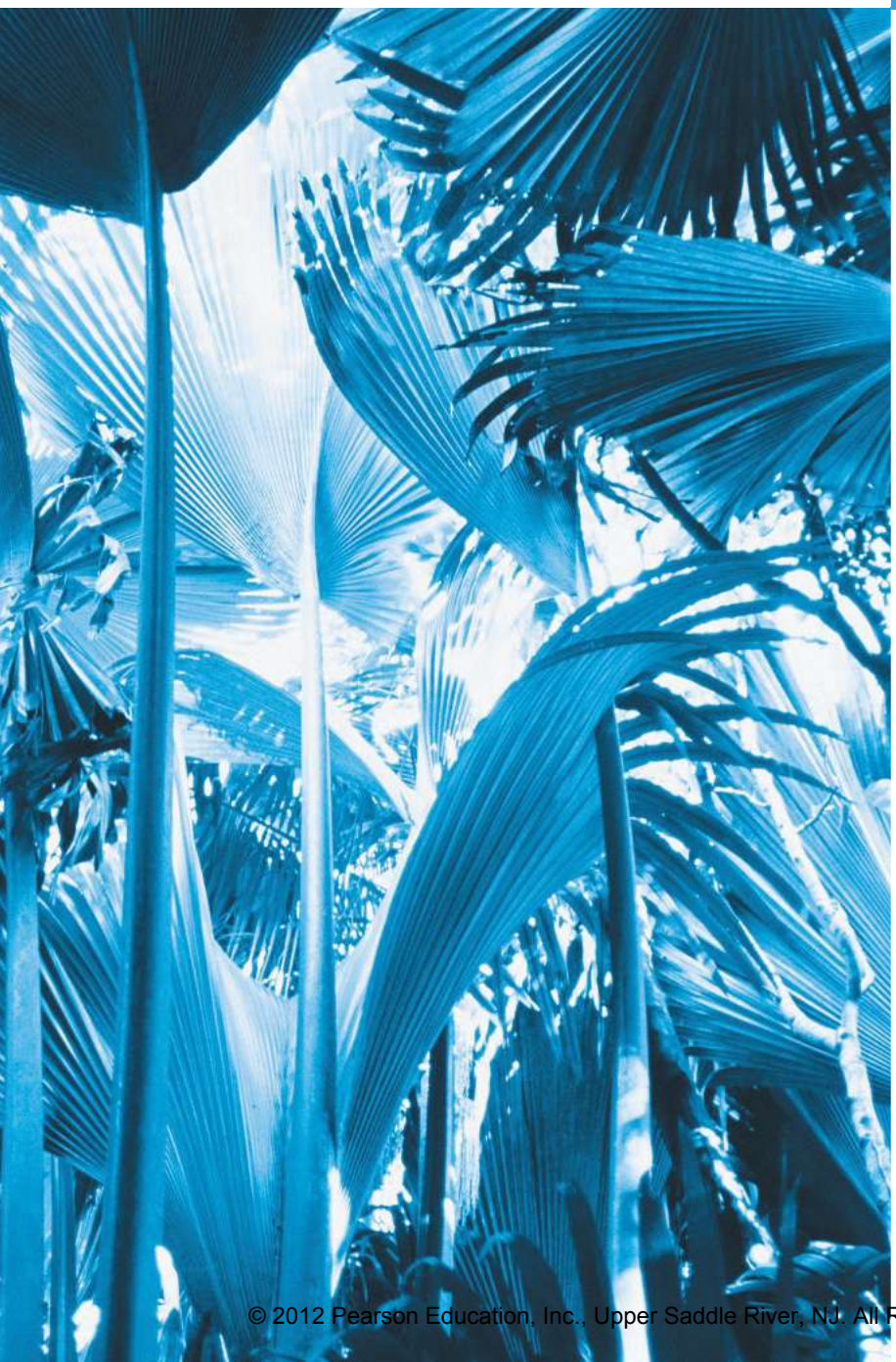
- How many page faults occur for FIFO, Optimal and LRU algorithms for the following reference string with four page frames?
- 1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

Page Replacement Exercise

- Consider the following page reference string:
- 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.
- How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames? Remember that all frames are initially empty, so your first unique pages will cost one fault each.
- LRU replacement
- FIFO replacement
- Optimal replacement

ATM Case Study, Part 1: Object-Oriented Design with the UML

25



Action speaks louder than words but not nearly as often.
—Mark Twain

Always design a thing by considering it in its next larger context.
—Eliel Saarinen

Oh, life is a glorious cycle of song.
—Dorothy Parker

The Wright brothers' design … allowed them to survive long enough to learn how to fly.
—Michael Potts

Objectives

In this chapter you'll learn:

- A simple object-oriented design methodology.
- What a requirements document is.
- To identify classes and class attributes from a requirements document.
- To identify objects' states, activities and operations from a requirements document.
- To determine the collaborations among objects in a system.
- To work with the UML's use case, class, state, activity, communication and sequence diagrams to graphically model an object-oriented system.

Outline

25.1 Introduction

Now we begin the optional portion of our object-oriented design and implementation case study. In this chapter and Chapter 26, you'll design and implement an object-oriented automated teller machine (ATM) software system. The case study provides you with a concise, carefully paced, complete design and implementation experience. You'll perform the steps of an object-oriented design (OOD) process using the UML while relating them to the object-oriented concepts discussed in Chapters 2–13. In this chapter, you'll work with six popular types of UML diagrams to graphically represent the design. In Chapter 26, you'll tune the design with inheritance and polymorphism, then fully implement the ATM in an 850-line C++ application (Section 26.4).

This is *not* an exercise; rather, it's an end-to-end learning experience that concludes with a detailed walkthrough of the *complete* C++ code that implements our design. It will acquaint you with the kinds of substantial problems encountered in industry.

These chapters can be studied as a continuous unit after you've completed the introduction to object-oriented programming in Chapters 2–13. Or, you can pace the sections after Chapters 3–7, 9 and 13. Each section of the case study begins with a note telling you the chapter after which it can be covered.

25.2 Introduction to Object-Oriented Analysis and Design

What if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or suppose you were asked to work on a team of 1000 software developers building the next U.S. air traffic control system. For projects so large and complex, you cannot simply sit down and start writing programs.

To create the best solutions, you should follow a process for **analyzing** your project's **requirements** (i.e., determining *what* the system should do) and developing a **design** that satisfies them (i.e., deciding *how* the system should do it). Ideally, you'd go through this process and carefully review the design (or have your design reviewed by other software professionals) before writing any code. If this process involves analyzing and designing your system from an object-oriented point of view, it's called an **object-oriented analysis and design (OOAD) process**. Analysis and design can save many hours by helping you to avoid an ill-planned system-development approach that has to be abandoned part of the way through its implementation, possibly wasting considerable time, money and effort. Small problems do not require an exhaustive OOAD process. It may be sufficient to write pseudocode before you begin writing C++ code.

As problems and the groups of people solving them increase in size, the methods of OOAD become more appropriate than pseudocode. Ideally, members of a group should agree on a strictly defined process for solving their problem and a uniform way of communicating the results of that process to one another. Although many different OOAD processes exist, a single graphical language for communicating the results of *any* OOAD process has come into wide use. This language, known as the Unified Modeling Language (UML), was developed in the mid-1990s under the initial direction of three software methodologists—Grady Booch, James Rumbaugh and Ivar Jacobson.

25.3 Examining the ATM Requirements Document

[*Note:* **This section can be studied after Chapter 3.**]
We begin our design process by presenting a **requirements document** that specifies the ATM system's overall purpose and *what* it must do. Throughout the case study, we refer to the requirements document to determine what functionality the system must include.

Requirements Document

A local bank intends to install a new automated teller machine (ATM) to allow users (i.e., bank customers) to perform basic financial transactions (Fig. 25.1). Each user can have only one account at the bank. ATM users should be able to *view their account balance*, *withdraw cash* (i.e., take money out of an account) and *deposit funds* (i.e., place money into an account).

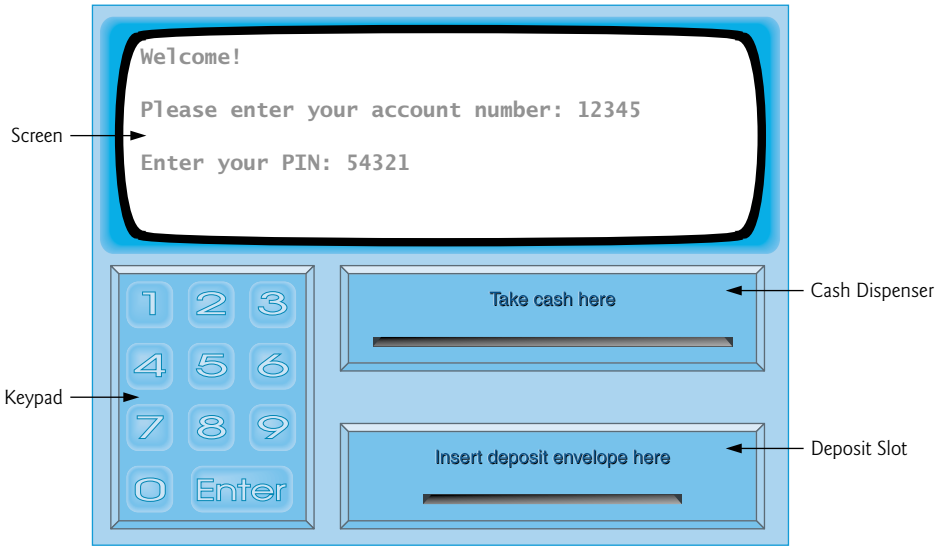


Fig. 25.1 | Automated teller machine user interface.

The user interface of the automated teller machine contains the following hardware components:

- a screen that displays messages to the user

- a keypad that receives numeric input from the user
- a cash dispenser that dispenses cash to the user and
- a deposit slot that receives deposit envelopes from the user.

The cash dispenser begins each day loaded with 500 $20 bills. [*Note:* Owing to the limited scope of this case study, certain elements of the ATM described here do not accurately mimic those of a real ATM. For example, a real ATM typically contains a device that reads a user's account number from an ATM card, whereas this ATM asks the user to type an account number using the keypad. A real ATM also usually prints a receipt at the end of a session, but all output from this ATM appears on the screen.]

The bank wants you to develop software to perform the financial transactions initiated by bank customers through the ATM. The bank will integrate the software with the ATM's hardware at a later time. The software should encapsulate the functionality of the hardware devices (e.g., cash dispenser, deposit slot) within software components, but it need not concern itself with how these devices perform their duties. The ATM hardware has not been developed yet, so instead of writing your software to run on the ATM, you should develop a first version of the software to run on a personal computer. This version should use the computer's monitor to simulate the ATM's screen, and the computer's keyboard to simulate the ATM's keypad.

An ATM session consists of authenticating a user (i.e., proving the user's identity) based on an account number and personal identification number (PIN), followed by creating and executing financial transactions. To authenticate a user and perform transactions, the ATM must interact with the bank's account information database. [*Note:* A database is an organized collection of data stored on a computer.] For each bank account, the database stores an account number, a PIN and a balance indicating the amount of money in the account. [*Note:* For simplicity, we assume that *the bank plans to build only one ATM, so we do not need to worry about multiple ATMs accessing this database at the same time. Furthermore, we assume that the bank does not make any changes to the information in the database while a user is accessing the ATM.* Also, any business system like an ATM faces reasonably complicated security issues that go well beyond the scope of a first- or second-semester computer science course. We make the simplifying assumption, however, that the bank trusts the ATM to access and manipulate the information in the database without significant security measures.]

Upon first approaching the ATM, the user should experience the following sequence of events (shown in Fig. 25.1):

1. The screen displays a welcome message and prompts the user to enter an account number.
2. The user enters a five-digit account number, using the keypad.
3. The screen prompts the user to enter the PIN (personal identification number) associated with the specified account number.
4. The user enters a five-digit PIN, using the keypad.
5. If the user enters a valid account number and the correct PIN for that account, the screen displays the main menu (Fig. 25.2). If the user enters an invalid account number or an incorrect PIN, the screen displays an appropriate message, then the ATM returns to *Step 1* to restart the authentication process.

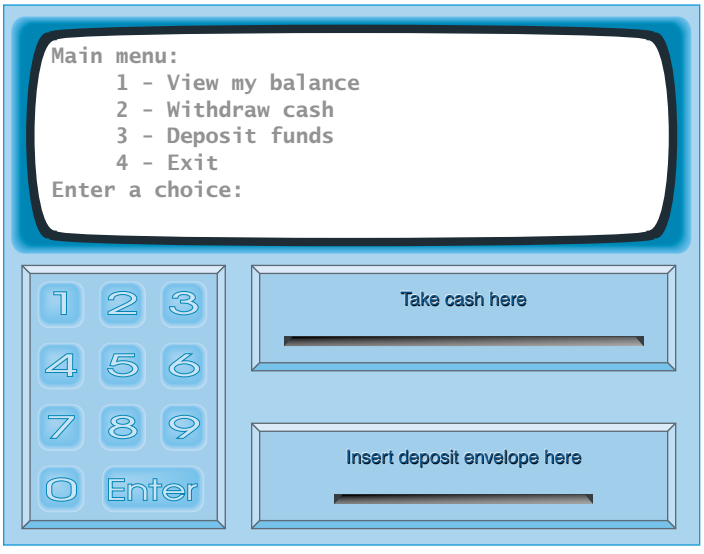


Fig. 25.2 | ATM main menu.

After the ATM authenticates the user, the main menu (Fig. 25.2) displays a numbered option for each of the three types of transactions: balance inquiry (option 1), withdrawal (option 2) and deposit (option 3). The main menu also displays an option that allows the user to exit the system (option 4). The user then chooses either to perform a transaction (by entering 1, 2 or 3) or to exit the system (by entering 4). If the user enters an invalid option, the screen displays an error message, then redisplays to the main menu.

If the user enters 1 to make a balance inquiry, the screen displays the user's account balance. To do so, the ATM must retrieve the balance from the bank's database.

The following actions occur when the user enters 2 to make a withdrawal:

1. The screen displays a menu (shown in Fig. 25.3) containing standard withdrawal amounts: $20 (option 1), $40 (option 2), $60 (option 3), $100 (option 4) and $200 (option 5). The menu also contains an option to allow the user to cancel the transaction (option 6).
2. The user enters a menu selection (1–6) using the keypad.
3. If the withdrawal amount chosen is greater than the user's account balance, the screen displays a message stating this and telling the user to choose a smaller amount. The ATM then returns to *Step 1*. If the withdrawal amount chosen is less than or equal to the user's account balance (i.e., an acceptable withdrawal amount), the ATM proceeds to *Step 4*. If the user chooses to cancel the transaction (option 6), the ATM displays the main menu (Fig. 25.2) and waits for user input.
4. If the cash dispenser contains enough cash to satisfy the request, the ATM proceeds to *Step 5*. Otherwise, the screen displays a message indicating the problem and telling the user to choose a smaller withdrawal amount. The ATM then returns to *Step 1*.

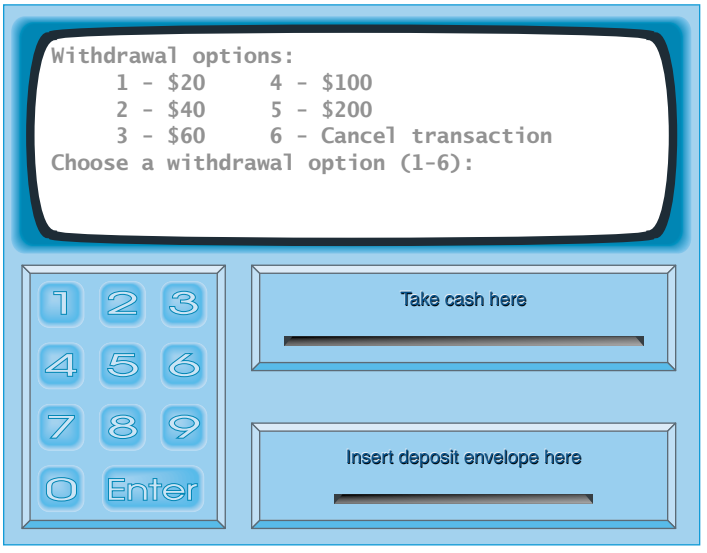


Fig. 25.3 | ATM withdrawal menu.

5. The ATM debits (i.e., subtracts) the withdrawal amount from the user's account balance in the bank's database.
6. The cash dispenser dispenses the desired amount of money to the user.
7. The screen displays a message reminding the user to take the money.

The following actions occur when the user enters 3 (while the main menu is displayed) to make a deposit:

1. The screen prompts the user to enter a deposit amount or to type 0 (zero) to cancel the transaction.
2. The user enters a deposit amount or 0, using the keypad. [*Note:* The keypad does not contain a decimal point or a dollar sign, so the user cannot type a real dollar amount (e.g., $1.25). Instead, the user must enter a deposit amount as a number of cents (e.g., 125). The ATM then divides this number by 100 to obtain a number representing a dollar amount (e.g., 125 ÷ 100 = 1.25).]
3. If the user specifies a deposit amount, the ATM proceeds to *Step 4*. If the user chooses to cancel the transaction (by entering 0), the ATM displays the main menu (Fig. 25.2) and waits for user input.
4. The screen displays a message telling the user to insert a deposit envelope into the deposit slot.
5. If the deposit slot receives a deposit envelope within two minutes, the ATM credits (i.e., adds) the deposit amount to the user's account balance in the bank's database. *This money is not immediately available for withdrawal. The bank first must physically verify the amount of cash in the deposit envelope, and any checks in the en-*

velope must clear (i.e., money must be transferred from the check writer's account to the check recipient's account). When either of these events occurs, the bank appropriately updates the user's balance stored in its database. This occurs independently of the ATM system. If the deposit slot does not receive a deposit envelope within this time period, the screen displays a message that the system has canceled the transaction due to inactivity. The ATM then displays the main menu and waits for user input.

After the system successfully executes a transaction, the system should redisplay the main menu (Fig. 25.2) so that the user can perform additional transactions. If the user chooses to exit the system (option 4), the screen should display a thank you message, then display the welcome message for the next user.

Analyzing the ATM System

The preceding statement is a simplified example of a requirements document. Typically, such a document is the result of a detailed **requirements gathering** process that might include interviews with potential users of the system and specialists in fields related to the system. For example, a systems analyst who is hired to prepare a requirements document for banking software (e.g., the ATM system described here) might interview financial experts to gain a better understanding of *what* the software must do. The analyst would use the information gained to compile a list of **system requirements** to guide systems designers.

The process of requirements gathering is a key task of the first stage of the software life cycle. The **software life cycle** specifies the stages through which software evolves from the time it's first conceived to the time it's retired from use. These stages typically include: analysis, design, implementation, testing and debugging, deployment, maintenance and retirement. Several software life-cycle models exist, each with its own preferences and specifications for when and how often software engineers should perform each of these stages. **Waterfall models** perform each stage once in succession, whereas **iterative models** may repeat one or more stages several times throughout a product's life cycle.

The analysis stage of the software life cycle focuses on defining the problem to be solved. When designing any system, one must certainly *solve the problem right*, but of equal importance, one must *solve the right problem*. Systems analysts collect the requirements that indicate the specific problem to solve. Our requirements document describes our ATM system in sufficient detail that you do not need to go through an extensive analysis stage—it has been done for you.

To capture what a proposed system should do, developers often employ a technique known as **use case modeling**. This process identifies the **use cases** of the system, each of which represents a different capability that the system provides to its clients. For example, ATMs typically have several use cases, such as "View Account Balance," "Withdraw Cash," "Deposit Funds," "Transfer Funds Between Accounts" and "Buy Postage Stamps." The simplified ATM system we build in this case study allows only the first three of these use cases (Fig. 25.4).

Each use case describes a typical scenario in which the user uses the system. You've already read descriptions of the ATM system's use cases in the requirements document; the lists of steps required to perform each type of transaction (i.e., balance inquiry, withdrawal and deposit) actually described the three use cases of our ATM—"View Account Balance," "Withdraw Cash" and "Deposit Funds."

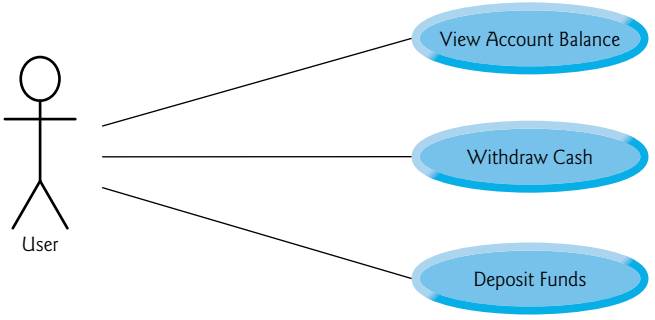


Fig. 25.4 | Use case diagram for the ATM system from the User's perspective.

Use Case Diagrams

We now introduce the first of several UML diagrams in our ATM case study. We create a **use case diagram** to model the interactions between a system's clients (in this case study, bank customers) and the system. The goal is to show the kinds of interactions users have with a system without providing the details—these are provided in other UML diagrams (which we present throughout the case study). Use case diagrams are often accompanied by informal text that describes the use cases in more detail—like the text that appears in the requirements document. Use case diagrams are produced during the analysis stage of the software life cycle. In larger systems, use case diagrams are simple but indispensable tools that help system designers remain focused on satisfying the users' needs.

Figure 25.4 shows the use case diagram for our ATM system. The stick figure represents an **actor**, which defines the roles that an external entity—such as a person or another system—plays when interacting with the system. For our automated teller machine, the actor is a User who can view an account balance, withdraw cash and deposit funds from the ATM. The User is not an actual person, but instead comprises the roles that a real person—when playing the part of a User—can play while interacting with the ATM. Note that a use case diagram can include multiple actors. For example, the use case diagram for a real bank's ATM system might also include an actor named Administrator who refills the cash dispenser each day.

We identify the actor in our system by examining the requirements document, which states, "ATM users should be able to view their account balance, withdraw cash and deposit funds." So, the actor in each of the three use cases is the User who interacts with the ATM. An external entity—a real person—plays the part of the User to perform financial transactions. Figure 25.4 shows one actor, whose name, User, appears below the actor in the diagram. The UML models each use case as an oval connected to an actor with a solid line.

Software engineers (more precisely, systems analysts) must analyze the requirements document or a set of use cases and design the system before programmers implement it. During the analysis stage, systems analysts focus on understanding the requirements document to produce a high-level specification that describes *what* the system is supposed to do. The output of the design stage—a **design specification**—should specify clearly *how* the system should be constructed to satisfy these requirements. In the next several sections, we perform the steps of a simple object-oriented design (OOD) process on the ATM

system to produce a design specification containing a collection of UML diagrams and supporting text. Recall that the UML is designed for use with any OOD process. Many such processes exist, the best known of which is the Rational Unified Process™ (RUP) developed by Rational Software Corporation (now a division of IBM). RUP is a rich process intended for designing "industrial strength" applications. For this case study, we present our own simplified design process.

Designing the ATM System

We now begin the ATM system's design. A **system** is a set of components that interact to solve a problem. To perform the ATM system's designated tasks, our ATM system has a user interface (Fig. 25.1), contains software that executes financial transactions and interacts with a database of bank account information. **System structure** describes the system's objects and their interrelationships. **System behavior** describes how the system changes as its objects interact with one another. Every system has both structure and behavior—designers must specify both. There are several distinct types of system structures and behaviors. For example, the interactions among objects in the system differ from those between the user and the system, yet both constitute a portion of the system behavior.

The UML 2 specifies 13 diagram types for documenting the models of systems. Each models a distinct characteristic of a system's structure or behavior—six diagrams relate to system structure; the remaining seven relate to system behavior. We list here only the six types of diagrams used in our case study—one of these (class diagrams) models system structure—the remaining five model system behavior. We overview the remaining seven UML diagram types in Appendix G, UML 2: Additional Diagram Types.

1. **Use case diagrams**, such as the one in Fig. 25.4, model the interactions between a system and its external entities (actors) in terms of use cases (system capabilities, such as "View Account Balance," "Withdraw Cash" and "Deposit Funds").
2. **Class diagrams**, which you'll study in Section 25.4, model the classes, or "building blocks," used in a system. Each noun or "thing" described in the requirements document is a candidate to be a class in the system (e.g., "account," "keypad"). Class diagrams help us specify the structural relationships between parts of the system. For example, the ATM system class diagram will specify that the ATM is physically composed of a screen, a keypad, a cash dispenser and a deposit slot.
3. **State machine diagrams**, which you'll study in Section 25.6, model the ways in which an object changes state. An object's **state** is indicated by the values of all the object's attributes at a given time. When an object changes state, that object may behave differently in the system. For example, after validating a user's PIN, the ATM transitions from the "user not authenticated" state to the "user authenticated" state, at which point the ATM allows the user to perform financial transactions (e.g., view account balance, withdraw cash, deposit funds).
4. **Activity diagrams**, which you'll also study in Section 25.6, model an object's **activity**—the object's workflow (sequence of events) during program execution. An activity diagram models the actions the object performs and specifies the order in which it performs these actions. For example, an activity diagram shows that the ATM must obtain the balance of the user's account (from the bank's account information database) before the screen can display the balance to the user.

5. **Communication diagrams** (called **collaboration diagrams** in earlier versions of the UML) model the interactions among objects in a system, with an emphasis on *what* interactions occur. You'll learn in Section 25.8 that these diagrams show which objects must interact to perform an ATM transaction. For example, the ATM must communicate with the bank's account information database to retrieve an account balance.
6. **Sequence diagrams** also model the interactions among the objects in a system, but unlike communication diagrams, they emphasize *when* interactions occur. You'll learn in Section 25.8 that these diagrams help show the order in which interactions occur in executing a financial transaction. For example, the screen prompts the user to enter a withdrawal amount before cash is dispensed.

In Section 25.4, we continue designing our ATM system by identifying the classes from the requirements document. We accomplish this by extracting key nouns and noun phrases from the requirements document. Using these classes, we develop our first draft of the class diagram that models the structure of our ATM system.

Web Resources

We've created an extensive UML Resource Center (`www.deitel.com/UML/`) that contains many links to additional information, including introductions, tutorials, blogs, books, certification, conferences, developer tools, documentation, e-books, FAQs, forums, groups, UML in C++, podcasts, security, tools, downloads, training courses, videos and more.

Self-Review Exercises for Section 25.3

25.1 Suppose we enabled a user of our ATM system to transfer money between two bank accounts. Modify the use case diagram of Fig. 25.4 to reflect this change.

25.2 _____ model the interactions among objects in a system with an emphasis on *when* these interactions occur.

a) Class diagrams
b) Sequence diagrams
c) Communication diagrams
d) Activity diagrams

25.3 Which of the following choices lists stages of a typical software life cycle in sequential order?

a) design, analysis, implementation, testing
b) design, analysis, testing, implementation
c) analysis, design, testing, implementation
d) analysis, design, implementation, testing

25.4 Identifying the Classes in the ATM Requirements Document

[*Note:* **This section can be studied after Chapter 3.**]

Now we begin designing the ATM system that we introduced in Section 25.3. In this section, we identify the classes that are needed to build the ATM system by analyzing the nouns and noun phrases that appear in the requirements document. We introduce UML class diagrams to model the relationships between these classes. This is an important first step in defining the structure of our system.

Identifying the Classes in a System

We begin our OOD process by identifying the classes required to build the ATM system. We'll eventually describe these classes using UML class diagrams and implement these classes in C++. First, we review the requirements document of Section 25.3 and find key nouns and noun phrases to help us identify classes that comprise the ATM system. We may decide that some of these nouns and noun phrases are attributes of other classes in the system. We may also conclude that some of the nouns do *not* correspond to parts of the system and thus should *not* be modeled at all. Additional classes may become apparent to us as we proceed through the design process.

Figure 25.5 lists the nouns and noun phrases in the requirements document. We list them from left to right in the order in which they appear in the requirements document. We list only the singular form of each noun or noun phrase.

Nouns and noun phrases in the requirements document			
bank	money / fund	account number	ATM
screen	PIN	user	keypad
bank database	customer	cash dispenser	balance inquiry
transaction	$20 bill / cash	withdrawal	account
deposit slot	deposit	balance	deposit envelope

Fig. 25.5 | Nouns and noun phrases in the requirements document.

We create classes only for the nouns and noun phrases that have significance in the ATM system. We don't need to model "bank" as a class, because it's not a part of the ATM system—the bank simply wants us to build the ATM. "Customer" and "user" also represent outside entities—they are important because they interact with our ATM system, but we do not need to model them as classes in the ATM software. Recall that we modeled an ATM user (i.e., a bank customer) as the actor in the use case diagram of Fig. 25.4.

We do not model "$20 bill" or "deposit envelope" as classes. These are physical objects in the real world, but they are *not* part of what's being automated. We can adequately represent the presence of bills in the system using an attribute of the class that models the cash dispenser. (We assign attributes to classes in Section 25.5.) For example, the cash dispenser maintains a count of the number of bills it contains. The requirements document doesn't say anything about what the system should do with deposit envelopes after it receives them. We can assume that acknowledging the receipt of an envelope—an operation performed by the class that models the deposit slot—is sufficient to represent the presence of an envelope in the system. (We assign operations to classes in Section 25.7.)

In our simplified ATM system, representing various amounts of "money," including an account's "balance," as attributes of other classes seems most appropriate. Likewise, the nouns "account number" and "PIN" represent significant information in the ATM system. They are important attributes of a bank account. They do *not*, however, exhibit behaviors. Thus, we can most appropriately model them as attributes of an account class.

Though the requirements document frequently describes a "transaction" in a general sense, we do not model the broad notion of a financial transaction at this time. Instead,

we model the three types of transactions (i.e., "balance inquiry," "withdrawal" and "deposit") as individual classes. These classes possess specific attributes needed for executing the transactions they represent. For example, a withdrawal needs to know the amount of money the user wants to withdraw. A balance inquiry, however, does not require any additional data. Furthermore, the three transaction classes exhibit unique behaviors. A withdrawal includes dispensing cash to the user, whereas a deposit involves receiving deposit envelopes from the user. *In Section 26.3, we "factor out" common features of all transactions into a general "transaction" class using the object-oriented concepts of abstract classes and inheritance.*

We determine the classes for our system based on the remaining nouns and noun phrases from Fig. 25.5. Each of these refers to one or more of the following:

- ATM
- screen
- keypad
- cash dispenser
- deposit slot
- account
- bank database
- balance inquiry
- withdrawal
- deposit

The elements of this list are likely to be classes we'll need to implement our system.

We can now model the classes in our system based on the list we've created. We capitalize class names in the design process—a UML convention—as we'll do when we write the actual C++ code that implements our design. If the name of a class contains more than one word, we run the words together and capitalize the first letter of each word (e.g., `MultipleWordName`). Using this convention, we create classes `ATM`, `Screen`, `Keypad`, `CashDispenser`, `DepositSlot`, `Account`, `BankDatabase`, `BalanceInquiry`, `Withdrawal` and `Deposit`. We construct our system using all of these classes as building blocks. Before we begin building the system, however, we must gain a better understanding of how the classes relate to one another.

Modeling Classes

The UML enables us to model, via **class diagrams**, the ATM system's classes and their interrelationships. Figure 25.6 represents class `ATM`. Each class is modeled as a rectangle with three compartments. The top compartment contains the name of the class, centered horizontally and in boldface. The middle compartment contains the class's attributes. (We discuss attributes in Section 25.5 and Section 25.6.) The bottom compartment contains the class's operations (discussed in Section 25.7). In Fig. 25.6 the middle and bottom compartments are empty, because we've not yet determined this class's attributes and operations.

Class diagrams also show the relationships among the classes of the system. Figure 25.7 shows how our classes `ATM` and `Withdrawal` relate to one another. For the moment, we choose to model only this subset of classes for simplicity; we present a more

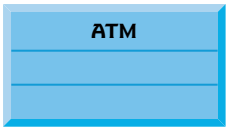


Fig. 25.6 | Representing a class in the UML using a class diagram.

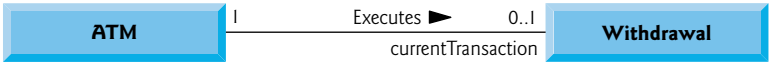


Fig. 25.7 | Class diagram showing an association among classes.

complete class diagram later in this section. Notice that the rectangles representing classes in this diagram are not subdivided into compartments. The UML allows the suppression of class attributes and operations in this manner, when appropriate, to create more readable diagrams. Such a diagram is said to be an **elided diagram**—one in which some information, such as the contents of the second and third compartments, is not modeled. We'll place information in these compartments in Section 25.5 and Section 25.7.

In Fig. 25.7, the solid line that connects the two classes represents an **association**—a relationship between classes. The numbers near each end of the line are **multiplicity** values, which indicate how many objects of each class participate in the association. In this case, following the line from one end to the other reveals that, at any given moment, one `ATM` object participates in an association with either zero or one `Withdrawal` objects—zero if the current user is not currently performing a transaction or has requested a different type of transaction, and one if the user has requested a withdrawal. The UML can model many types of multiplicity. Figure 25.8 lists and explains the multiplicity types.

Symbol	Meaning
0	None
1	One
m	An integer value
0..1	Zero or one
m, *n*	*m* or *n*
m..*n*	At least *m*, but not more than *n*
*	Any nonnegative integer (zero or more)
0..*	Zero or more (identical to *)
1..*	One or more

Fig. 25.8 | Multiplicity types.

An association can be named. For example, the word `Executes` above the line connecting classes `ATM` and `Withdrawal` in Fig. 25.7 indicates the name of that association. This part of the diagram reads "one object of class `ATM` executes zero or one objects of class

`Withdrawal`." Association names are directional, as indicated by the filled arrowhead—so it would be improper, for example, to read the preceding association from right to left as "zero or one objects of class `Withdrawal` execute one object of class `ATM`."

The word `currentTransaction` at the `Withdrawal` end of the association line in Fig. 25.7 is a **role name**, which identifies the role the `Withdrawal` object plays in its relationship with the `ATM`. A role name adds meaning to an association between classes by identifying the role a class plays in the context of an association. A class can play several roles in the same system. For example, in a school personnel system, a person may play the role of "professor" when relating to students. The same person may take on the role of "colleague" when participating in a relationship with another professor, and "coach" when coaching student athletes. In Fig. 25.7, the role name `currentTransaction` indicates that the `Withdrawal` object participating in the `Executes` association with an object of class `ATM` represents the transaction currently being processed by the ATM. In other contexts, a `Withdrawal` object may take on other roles (e.g., the previous transaction). Notice that we do *not* specify a role name for the `ATM` end of the `Executes` association. Role names in class diagrams are often omitted when the meaning of an association is clear without them.

In addition to indicating simple relationships, associations can specify more complex relationships, such as objects of one class being composed of objects of other classes. Consider a real-world automated teller machine. What "pieces" does a manufacturer put together to build a working ATM? Our requirements document tells us that the ATM is composed of a screen, a keypad, a cash dispenser and a deposit slot.

In Fig. 25.9, the **solid diamonds** attached to the association lines of class `ATM` indicate that class `ATM` has a **composition** relationship with classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`. Composition implies a whole/part relationship. The class that has the *composition symbol* (the solid diamond) on its end of the association line is the whole (in this case, `ATM`), and the classes on the other end of the association lines are the parts—in this case, classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`. The compositions in Fig. 25.9 indicate that an object of class `ATM` is formed from one object of class `Screen`, one object of class `CashDispenser`, one object of class `Keypad` and one object of class `DepositSlot`. The ATM *has a* screen, a keypad, a cash dispenser and a deposit slot. The ***has-a* relationship** defines composition. (We'll see in Section 26.3 that the *is-a* relationship defines inheritance.)

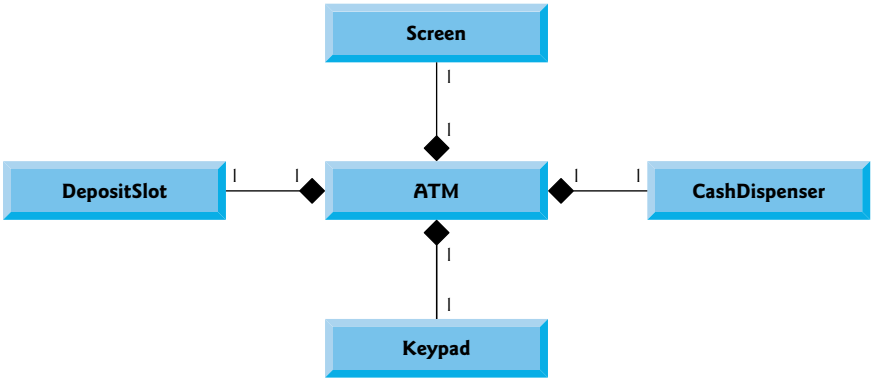


Fig. 25.9 | Class diagram showing composition relationships.

According to the UML specification, composition relationships have the following properties:

1. Only one class in the relationship can represent the whole (i.e., the diamond can be placed on only one end of the association line). For example, either the screen is part of the ATM or the ATM is part of the screen, but the screen and the ATM cannot both represent the whole in the relationship.
2. The parts in a composition relationship exist only as long as the whole, and the whole is responsible for creating and destroying its parts. For example, the act of constructing an ATM includes manufacturing its parts. Furthermore, if the ATM is destroyed, its screen, keypad, cash dispenser and deposit slot are also destroyed.
3. A part may belong to only one whole at a time, although the part may be removed and attached to another whole, which then assumes responsibility for the part.

The solid diamonds in our class diagrams indicate composition relationships that fulfill these three properties. If a *has-a* relationship does not satisfy one or more of these criteria, the UML specifies that hollow diamonds be attached to the ends of association lines to indicate **aggregation**—a weaker form of composition. For example, a personal computer and a computer monitor participate in an aggregation relationship—the computer *has a* monitor, but the two parts can exist independently, and the same monitor can be attached to multiple computers at once, thus violating the second and third properties of composition.

Figure 25.10 shows a class diagram for the ATM system. This diagram models most of the classes that we identified earlier in this section, as well as the associations between them that we can infer from the requirements document. [*Note:* Classes `BalanceInquiry` and `Deposit` participate in associations similar to those of class `Withdrawal`, so we've chosen to omit them from this diagram to keep it simple. In Section 26.3, we expand our class diagram to include all the classes in the ATM system.]

Figure 25.10 presents a graphical model of the structure of the ATM system. This class diagram includes classes `BankDatabase` and `Account` and several associations that were not present in either Fig. 25.7 or Fig. 25.9. The class diagram shows that class `ATM` has a **one-to-one relationship** with class `BankDatabase`—one `ATM` object authenticates users against one `BankDatabase` object. In Fig. 25.10, we also model the fact that the bank's database contains information about many accounts—one object of class `BankDatabase` participates in a composition relationship with zero or more objects of class `Account`. Recall from Fig. 25.8 that the multiplicity value 0..* at the `Account` end of the association between class `BankDatabase` and class `Account` indicates that zero or more objects of class `Account` take part in the association. Class `BankDatabase` has a **one-to-many relationship** with class `Account`—the `BankDatabase` contains many `Accounts`. Similarly, class `Account` has a **many-to-one relationship** with class `BankDatabase`—there can be many `Accounts` contained in the `BankDatabase`. [*Note:* Recall from Fig. 25.8 that the multiplicity value * is identical to 0..*. We include 0..* in our class diagrams for clarity.]

Figure 25.10 also indicates that if the user is performing a withdrawal, "one object of class `Withdrawal` accesses/modifies an account balance through one object of class `BankDatabase`." We could have created an association directly between class `Withdrawal` and class `Account`. The requirements document, however, states that the "ATM must interact with the bank's account information database" to perform transactions. A bank account

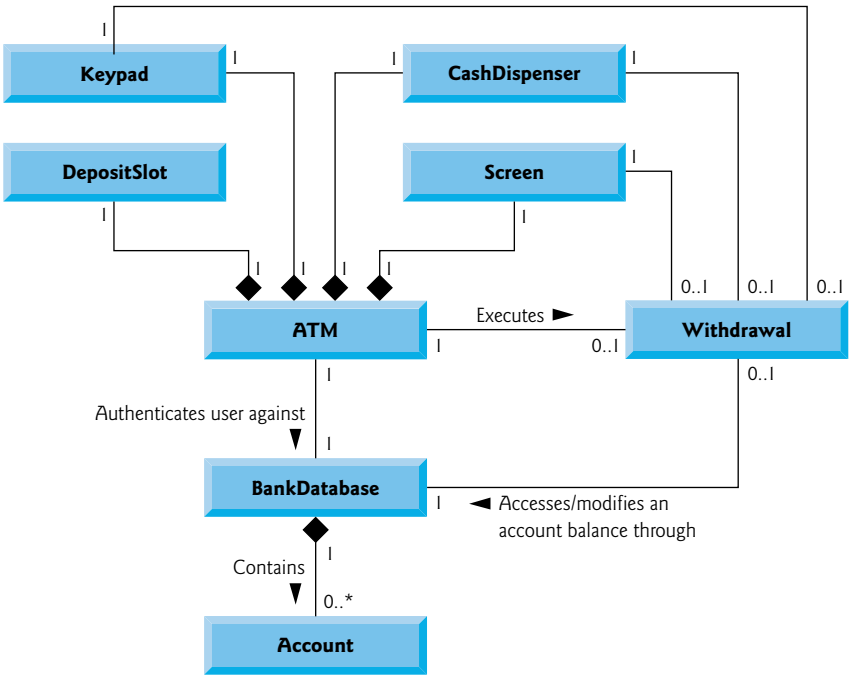


Fig. 25.10 | Class diagram for the ATM system model.

contains sensitive information, and systems engineers must always consider the security of personal data when designing a system. Thus, only the `BankDatabase` can access and manipulate an account directly. All other parts of the system must interact with the database to retrieve or update account information (e.g., an account balance).

The class diagram in Fig. 25.10 also models associations between class `Withdrawal` and classes `Screen`, `CashDispenser` and `Keypad`. A withdrawal transaction includes prompting the user to choose a withdrawal amount and receiving numeric input. These actions require the use of the screen and the keypad, respectively. Furthermore, dispensing cash to the user requires access to the cash dispenser.

Classes `BalanceInquiry` and `Deposit`, though not shown in Fig. 25.10, take part in several associations with the other classes of the ATM system. Like class `Withdrawal`, each of these classes associates with classes `ATM` and `BankDatabase`. An object of class `BalanceInquiry` also associates with an object of class `Screen` to display the balance of an account to the user. Class `Deposit` associates with classes `Screen`, `Keypad` and `DepositSlot`. Like withdrawals, deposit transactions require use of the screen and the keypad to display prompts and receive input, respectively. To receive deposit envelopes, an object of class `Deposit` accesses the deposit slot.

We've now identified the classes in our ATM system (although we may discover others as we proceed with the design and implementation). In Section 25.5, we determine the attributes for each of these classes, and in Section 25.6, we use these attributes to examine how the system changes over time. In Section 25.7, we determine the operations of the classes in our system.

Self-Review Exercises for Section 25.4

25.4 Suppose we have a class `Car` that represents a car. Think of some of the different pieces that a manufacturer would put together to produce a whole car. Create a class diagram (similar to Fig. 25.9) that models some of the composition relationships of class `Car`.

25.5 Suppose we have a class `File` that represents an electronic document in a stand-alone, non-networked computer represented by class `Computer`. What sort of association exists between class `Computer` and class `File`?

a) Class `Computer` has a *one-to-one* relationship with class `File`.
b) Class `Computer` has a *many-to-one* relationship with class `File`.
c) Class `Computer` has a *one-to-many* relationship with class `File`.
d) Class `Computer` has a *many-to-many* relationship with class `File`.

25.6 State whether the following statement is *true* or *false*, and if *false*, explain why: A UML diagram in which a class's second and third compartments are not modeled is said to be an elided diagram.

25.7 Modify the class diagram of Fig. 25.10 to include class `Deposit` instead of class `Withdrawal`.

25.5 Identifying Class Attributes

[*Note:* **This section can be studied after Chapter 4.**]
In Section 25.4, we began the first stage of an object-oriented design (OOD) for our ATM system—analyzing the requirements document and identifying the classes needed to implement the system. We listed the *nouns* and *noun phrases* in the requirements document and identified a separate class for each one that plays a significant role in the ATM system. We then modeled the classes and their relationships in a UML class diagram (Fig. 25.10).

Classes have attributes (data) and operations (behaviors). Class attributes are implemented in C++ programs as data members, and class operations are implemented as member functions. In this section, we determine many of the attributes needed in the ATM system. In Section 25.6, we examine how these attributes represent an object's state. In Section 25.7, we determine class operations.

Identifying Attributes

Consider the attributes of some real-world objects: A person's attributes include height, weight and whether the person is left-handed, right-handed or ambidextrous. A radio's attributes include its station setting, its volume setting and its AM or FM setting. A car's attributes include its speedometer and odometer readings, the amount of gas in its tank and what gear it's in. A personal computer's attributes include its manufacturer (e.g., Dell, HP, Apple or IBM), type of screen (e.g., LCD or CRT), main memory size and hard disk size.

We can identify many attributes of the classes in our system by looking for descriptive words and phrases in the requirements document. For each one we find that plays a significant role in the ATM system, we create an attribute and assign it to one or more of the classes identified in Section 25.4. We also create attributes to represent any additional data that a class may need, as such needs become apparent throughout the design process.

Figure 25.11 lists the words or phrases from the requirements document that describe each class. We formed this list by reading the requirements document and identifying any words or phrases that refer to characteristics of the classes in the system. For example, the requirements document describes the steps taken to obtain a "withdrawal amount," so we list "amount" next to class `Withdrawal`.

Figure 25.11 leads us to create one attribute of class `ATM`. Class `ATM` maintains information about the state of the ATM. The phrase "user is authenticated" describes a state of the ATM (we introduce states in Section 25.6), so we include `userAuthenticated` as a **`Boolean` attribute** (i.e., an attribute that has a value of either `true` or `false`). The UML `Boolean` type is equivalent to the `bool` type in C++. This attribute indicates whether the ATM has successfully authenticated the current user—`userAuthenticated` must be `true` for the system to allow the user to perform transactions and access account information. This attribute helps ensure the security of the data in the system.

Class	Descriptive words and phrases
`ATM`	user is authenticated
`BalanceInquiry`	account number
`Withdrawal`	account number amount
`Deposit`	account number amount
`BankDatabase`	[no descriptive words or phrases]
`Account`	account number PIN balance
`Screen`	[no descriptive words or phrases]
`Keypad`	[no descriptive words or phrases]
`CashDispenser`	begins each day loaded with 500 $20 bills
`DepositSlot`	[no descriptive words or phrases]

Fig. 25.11 | Descriptive words and phrases from the ATM requirements.

Classes `BalanceInquiry`, `Withdrawal` and `Deposit` share one attribute. Each transaction involves an "account number" that corresponds to the account of the user making the transaction. We assign an integer attribute `accountNumber` to each transaction class to identify the account to which an object of the class applies.

Descriptive words and phrases in the requirements document also suggest some differences in the attributes required by each transaction class. The requirements document indicates that to withdraw cash or deposit funds, users must enter a specific "amount" of money to be withdrawn or deposited, respectively. Thus, we assign to classes `Withdrawal` and `Deposit` an attribute `amount` to store the value supplied by the user. The amounts of money related to a withdrawal and a deposit are defining characteristics of these transactions that the system requires for them to take place. Class `BalanceInquiry`, however, needs no additional data to perform its task—it requires only an account number to indicate the account whose balance should be retrieved.

Class `Account` has several attributes. The requirements document states that each bank account has an "account number" and "PIN," which the system uses for identifying accounts

and authenticating users. We assign to class `Account` two integer attributes: `accountNumber` and `pin`. The requirements document also specifies that an account maintains a "balance" of the amount of money in the account and that money the user deposits does not become available for a withdrawal until the bank verifies the amount of cash in the deposit envelope, and any checks in the envelope clear. An account must still record the amount of money that a user deposits, however. Therefore, we decide that an account should represent a balance using two attributes of UML type `Double`: `availableBalance` and `totalBalance`. Attribute `availableBalance` tracks the amount of money that a user can withdraw from the account. Attribute `totalBalance` refers to the total amount of money that the user has "on deposit" (i.e., the amount of money available, plus the amount waiting to be verified or cleared). For example, suppose an ATM user deposits $50.00 into an empty account. The `totalBalance` attribute would increase to $50.00 to record the deposit, but the `availableBalance` would remain at $0. [*Note:* We assume that the bank updates the `availableBalance` attribute of an `Account` soon after the ATM transaction occurs, in response to confirming that $50 worth of cash or checks was found in the deposit envelope. We assume that this update occurs through a transaction that a bank employee performs using some piece of bank software other than the ATM. Thus, we do not discuss this transaction in our case study.]

Class `CashDispenser` has one attribute. The requirements document states that the cash dispenser "begins each day loaded with 500 $20 bills." The cash dispenser must keep track of the number of bills it contains to determine whether enough cash is on hand to satisfy withdrawal requests. We assign to class `CashDispenser` an integer attribute `count`, which is initially set to `500`.

For real problems in industry, there is no guarantee that requirements specifications will be rich enough and precise enough for the object-oriented systems designer to determine all the attributes or even all the classes. The need for additional (or fewer) classes, attributes and behaviors may become clear as the design process proceeds. As we progress through this case study, we too will continue to add, modify and delete information about the classes in our system.

Modeling Attributes

The class diagram in Fig. 25.12 lists some of the attributes for the classes in our system—the descriptive words and phrases in Fig. 25.11 helped us identify these attributes. For simplicity, Fig. 25.12 does not show the associations among classes—we showed these in Fig. 25.10. This is a common practice of systems designers when designs are being developed. Recall from Section 25.4 that in the UML, a class's attributes are placed in the middle compartment of the class's rectangle. We list each attribute's name and type separated by a colon (`:`), followed in some cases by an equal sign (`=`) and an initial value.

Consider the `userAuthenticated` attribute of class `ATM`:

```
userAuthenticated : Boolean = false
```

This attribute declaration contains three pieces of information about the attribute. The **attribute name** is `userAuthenticated`. The **attribute type** is `Boolean`. In C++, an attribute can be represented by a fundamental type, such as `bool`, `int` or `double`, or a class type. We've chosen to model only primitive-type attributes in Fig. 25.12—we discuss the reasoning behind this decision shortly. [*Note:* Figure 25.12 lists UML data types for the attributes. When we implement the system, we'll associate the UML types `Boolean`, `Integer` and `Double` with the C++ fundamental types `bool`, `int` and `double`, respectively.]

Fig. 25.12 | Classes with attributes.

We can also indicate an *initial value* for an attribute. The `userAuthenticated` attribute in class `ATM` has an initial value of `false`. This indicates that the system initially does not consider the user to be authenticated. If an attribute has no initial value specified, only its name and type (separated by a colon) are shown. For example, the `accountNumber` attribute of class `BalanceInquiry` is an `Integer`. Here we show no initial value, because the value of this attribute is a number that we do not yet know—it will be determined at execution time based on the account number entered by the current ATM user.

Figure 25.12 does not include any attributes for classes `Screen`, `Keypad` and `DepositSlot`. These are important components of our system, for which our design process simply has not yet revealed any attributes. We may still discover some, however, in the remaining design phases or when we implement these classes in C++. This is perfectly normal for the iterative process of software engineering.

Software Engineering Observation 25.1

At the early stages in the design process, classes often lack attributes (and operations). Such classes should not *be eliminated, however, because attributes (and operations) may become evident in the later phases of design and implementation.*

Figure 25.12 also does not include attributes for class `BankDatabase`. Recall that attributes can be represented by either fundamental types or class types. We've chosen to

include only fundamental-type attributes in the class diagram in Fig. 25.12 (and in similar class diagrams throughout the case study). A class-type attribute is modeled more clearly as an association (in particular, a composition) between the class with the attribute and the class of the object of which the attribute is an instance. For example, the class diagram in Fig. 25.10 indicates that class `BankDatabase` participates in a composition relationship with zero or more `Account` objects. From this composition, we can determine that when we implement the ATM system in C++, we'll be required to create an attribute of class `BankDatabase` to hold zero or more `Account` objects. Similarly, we'll assign attributes to class `ATM` that correspond to its composition relationships with classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`. These composition-based attributes would be redundant if modeled in Fig. 25.12, because the compositions modeled in Fig. 25.10 already convey the fact that the database contains information about zero or more accounts and that an ATM is composed of a screen, keypad, cash dispenser and deposit slot. Software developers typically model these whole/part relationships as compositions rather than as attributes required to implement the relationships.

The class diagram in Fig. 25.12 provides a solid basis for the structure of our model, but the diagram is not complete. In Section 25.6, we identify the states and activities of the objects in the model, and in Section 25.7 we identify the operations that the objects perform. As we present more of the UML and object-oriented design, we'll continue to strengthen the structure of our model.

Self-Review Exercises for Section 25.5

25.8 We typically identify the attributes of the classes in our system by analyzing the _____ in the requirements document.

a) nouns and noun phrases
b) descriptive words and phrases
c) verbs and verb phrases
d) All of the above.

25.9 Which of the following is *not* an attribute of an airplane?

a) length
b) wingspan
c) fly
d) number of seats

25.10 Describe the meaning of the following attribute declaration of class `CashDispenser` in the class diagram in Fig. 25.12:

```
count : Integer = 500
```

25.6 Identifying Objects' States and Activities

[*Note:* **This section can be studied after Chapter 5.**]

In Section 25.5, we identified many of the class attributes needed to implement the ATM system and added them to the class diagram in Fig. 25.12. In this section, we show how these attributes represent an object's *state*. We identify some key states that our objects may occupy and discuss how objects change state in response to various events occurring in the system. We also discuss the workflow, or activities, that objects perform in the ATM system. We present the activities of `BalanceInquiry` and `Withdrawal` transaction objects in this section, as they represent two of the key activities in the ATM system.

State Machine Diagrams

Each object in a system goes through a series of discrete states. An object's current state is indicated by the values of the object's attributes at a given time. **State machine diagrams** (commonly called **state diagrams**) model key states of an object and show under what circumstances the object changes state. Unlike the class diagrams presented in earlier case study sections, which focused primarily on the *structure* of the system, state diagrams model some of the *behavior* of the system.

Figure 25.13 is a simple state diagram that models some of the states of an object of class `ATM`. The UML represents each state in a state diagram as a **rounded rectangle** with the name of the state placed inside it. A **solid circle** with an attached stick arrowhead designates the **initial state**. Recall that we modeled this state information as the `Boolean` attribute `userAuthenticated` in the class diagram of Fig. 25.12. This attribute is initialized to `false`, or the "User not authenticated" state, according to the state diagram.

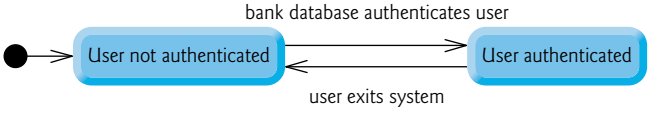


Fig. 25.13 | State diagram for the `ATM` object.

The arrows with stick arrowheads indicate **transitions** between states. An object can transition from one state to another in response to various *events* that occur in the system. The name or description of the event that causes a transition is written near the line that corresponds to the transition. For example, the `ATM` object changes from the "User not authenticated" state to the "User authenticated" state *after* the database authenticates the user. Recall from the requirements document that the database authenticates a user by comparing the account number and PIN entered by the user with those of the corresponding account in the database. If the database indicates that the user has entered a valid account number and the correct PIN, the `ATM` object transitions to the "User authenticated" state and changes its `userAuthenticated` attribute to a value of `true`. When the user exits the system by choosing the "exit" option from the main menu, the `ATM` object returns to the "User not authenticated" state in preparation for the next ATM user.

Software Engineering Observation 25.2

Software designers do not generally create state diagrams showing every possible state and state transition for all attributes—there are simply too many of them. State diagrams typically show only the most important or complex states and state transitions.

Activity Diagrams

Like a state diagram, an activity diagram models aspects of system *behavior*. Unlike a state diagram, an activity diagram models an object's workflow (sequence of events) during program execution. An activity diagram models the actions the object will perform and in what order. Recall that we used UML activity diagrams to illustrate the flow of control for the control statements presented in Chapters 4 and 5.

Figure 25.14 models the actions involved in executing a `BalanceInquiry` transaction. We assume that a `BalanceInquiry` object has been initialized and assigned a valid account

number (that of the current user), so the object knows which balance to retrieve. The diagram includes the actions that occur after the user selects a balance inquiry from the main menu and before the ATM returns the user to the main menu—a `BalanceInquiry` object does not perform or initiate these actions, so we do not model them here. The diagram begins with retrieving the available balance of the user's account from the database. Next, the `BalanceInquiry` retrieves the total balance of the account. Finally, the transaction displays the balances on the screen. This action completes the execution of the transaction.

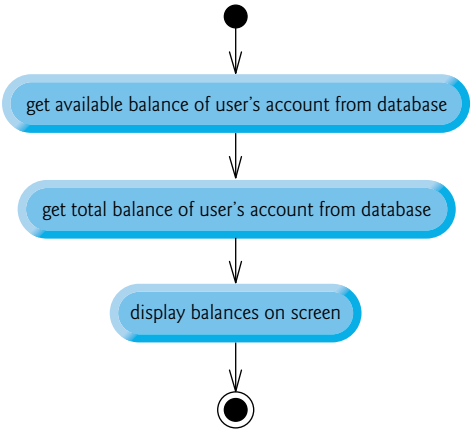


Fig. 25.14 | Activity diagram for a `BalanceInquiry` transaction.

The UML represents an action in an activity diagram as an action state modeled by a rectangle with its left and right sides replaced by arcs curving outward. Each action state contains an action expression—for example, "get available balance of user's account from database"—that specifies an action to be performed. An arrow with a stick arrowhead connects two action states, indicating the order in which the actions represented by the action states occur. The solid circle (at the top of Fig. 25.14) represents the activity's initial state—the beginning of the workflow before the object performs the modeled actions. In this case, the transaction first executes the "get available balance of user's account from database" action expression. Second, the transaction retrieves the total balance. Finally, the transaction displays both balances on the screen. The solid circle enclosed in an open circle (at the bottom of Fig. 25.14) represents the final state—the end of the workflow after the object performs the modeled actions.

Figure 25.15 shows an activity diagram for a `Withdrawal` transaction. We assume that a `Withdrawal` object has been assigned a valid account number. We do not model the user selecting a withdrawal from the main menu or the ATM returning the user to the main menu because these are not actions performed by a `Withdrawal` object. The transaction first displays a menu of standard withdrawal amounts (Fig. 25.3) and an option to cancel the transaction. The transaction then inputs a menu selection from the user. The activity flow now arrives at a decision symbol. This point determines the next action based on the associated guard conditions. If the user cancels the transaction, the system displays an appropriate message. Next, the cancellation flow reaches a merge symbol, where this activity flow joins the transaction's other possible activity flows (which we discuss shortly).

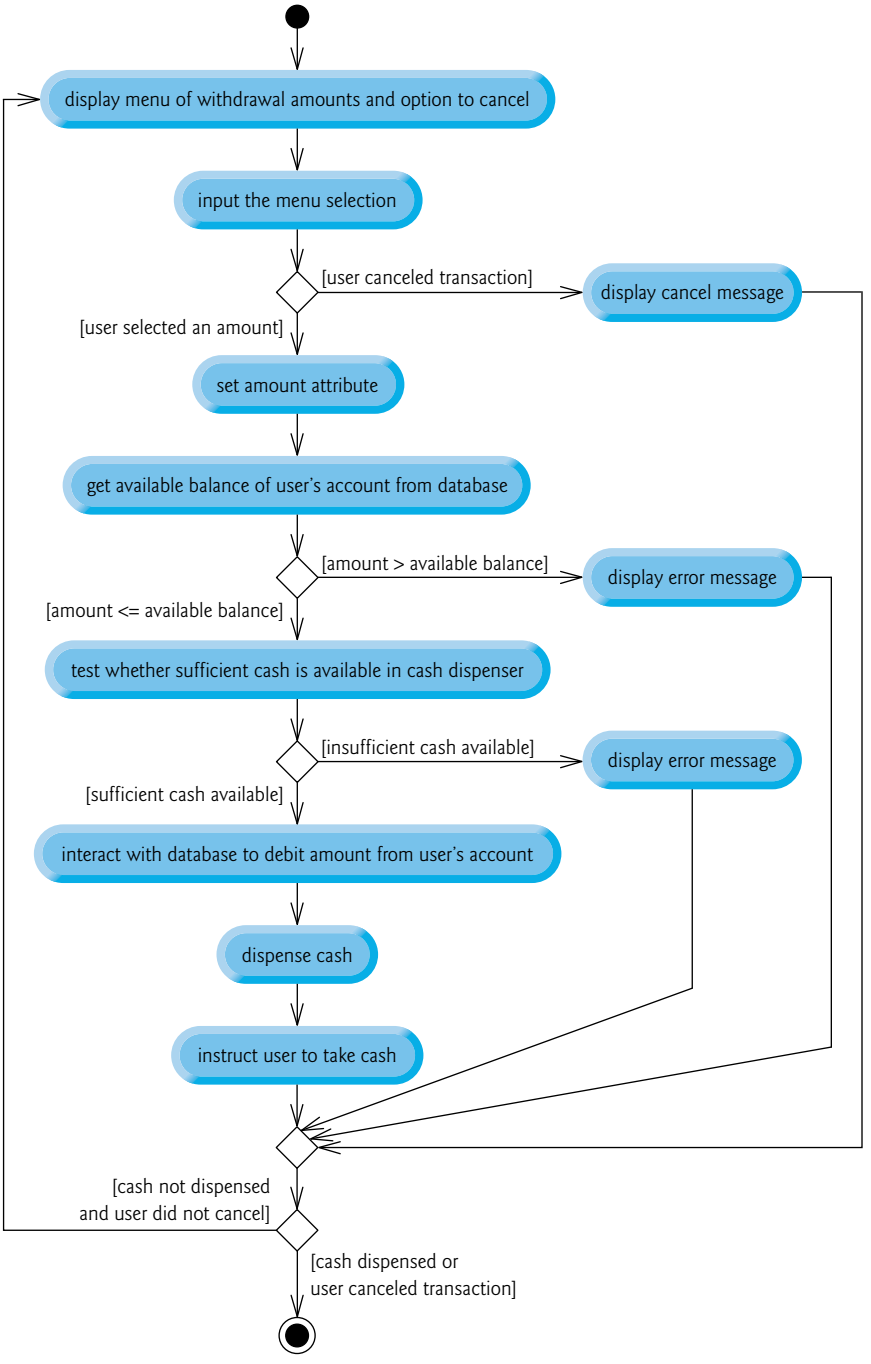


Fig. 25.15 | Activity diagram for a `Withdrawal` transaction.

A merge can have *any* number of incoming transition arrows, but only *one* outgoing transition arrow. The decision at the bottom of the diagram determines whether the transaction should repeat from the beginning. When the user has canceled the transaction, the guard condition "cash dispensed or user canceled transaction" is true, so control transitions to the activity's final state.

If the user selects a withdrawal amount from the menu, `the` transaction sets `amount` (an attribute of class `Withdrawal` originally modeled in Fig. 25.12) to the value chosen by the user. The transaction next gets the available balance of the user's account (i.e., the `availableBalance` attribute of the user's `Account` object) from the database. The activity flow then arrives at another decision. If the requested withdrawal amount exceeds the user's available balance, the system displays an appropriate error message informing the user of the problem. Control then merges with the other activity flows before reaching the decision at the bottom of the diagram. The guard decision "cash not dispensed and user did not cancel" is true, so the activity flow returns to the top of the diagram, and the transaction prompts the user to input a new amount.

If the requested withdrawal amount is less than or equal to the user's available balance, the transaction tests whether the cash dispenser has enough cash to satisfy the withdrawal request. If it does not, the transaction displays an appropriate error message and passes through the merge before reaching the final decision. Cash was not dispensed, so the activity flow returns to the beginning of the activity diagram, and the transaction prompts the user to choose a new amount. If sufficient cash is available, the transaction interacts with the database to debit the withdrawal amount from the user's account (i.e., subtract the amount from both the `availableBalance` and `totalBalance` attributes of the user's `Account` object). The transaction then dispenses the desired amount of cash and instructs the user to take the cash that is dispensed. The main flow of activity next merges with the two error flows and the cancellation flow. In this case, cash was dispensed, so the activity flow reaches the final state.

We've taken the first steps in modeling the ATM system's behavior and have shown how an object's attributes participate in the object's activities. In Section 25.7, we investigate the operations of our classes to create a more complete model of the system's behavior.

Self-Review Exercises for Section 25.6

25.11 State whether the following statement is *true* or *false*, and if *false*, explain why: State diagrams model structural aspects of a system.

25.12 An activity diagram models the _____ that an object performs and the order in which it performs them.

a) actions
b) attributes
c) states
d) state transitions

25.13 Based on the requirements document, create an activity diagram for a deposit transaction.

25.7 Identifying Class Operations

[*Note:* **This section can be studied after Chapter 6.**]

In Sections 25.4–25.6, we performed the first few steps in the object-oriented design of our ATM system. In Section 25.4, we identified the classes that we'll need to implement

and we created our first class diagram. In Section 25.5, we described some attributes of our classes. In Section 25.6, we examined object states and modeled object state transitions and activities. Now, we determine some of the class operations (or behaviors) needed to implement the ATM system.

Identifying Operations

An *operation* is a service that objects of a class provide to clients of the class. Consider the operations of some real-world objects. A radio's operations include setting its station and volume (typically invoked by a person adjusting the radio's controls). A car's operations include accelerating (invoked by the driver pressing the accelerator pedal), decelerating (invoked by the driver pressing the brake pedal or releasing the gas pedal), turning and shifting gears. Software objects can offer operations as well—for example, a software graphics object might offer operations for drawing a circle, drawing a line, drawing a square and the like. A spreadsheet software object might offer operations like printing the spreadsheet, totaling the elements in a row or column and graphing information in the spreadsheet as a bar chart or pie chart.

We can derive many of the operations of each class by examining the key verbs and verb phrases in the requirements document. We then relate each of these to particular classes in our system (Fig. 25.16). The verb phrases in Fig. 25.16 help us determine the operations of each class.

Class	Verbs and verb phrases
`ATM`	executes financial transactions
`BalanceInquiry`	[none in the requirements document]
`Withdrawal`	[none in the requirements document]
`Deposit`	[none in the requirements document]
`BankDatabase`	authenticates a user, retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
`Account`	retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
`Screen`	displays a message to the user
`Keypad`	receives numeric input from the user
`CashDispenser`	dispenses cash, indicates whether it contains enough cash to satisfy a withdrawal request
`DepositSlot`	receives a deposit envelope

Fig. 25.16 | Verbs and verb phrases for each class in the ATM system.

Modeling Operations

To identify operations, we examine the verb phrases listed for each class in Fig. 25.16. The "executes financial transactions" phrase associated with class `ATM` implies that class `ATM` instructs transactions to execute. Therefore, classes `BalanceInquiry`, `Withdrawal` and `Deposit` each need an operation to provide this service to the ATM. We place this operation

(which we've named `execute`) in the third compartment of the three transaction classes in the updated class diagram of Fig. 25.17. During an ATM session, the `ATM` object will invoke the `execute` operation of each transaction object to tell it to execute.

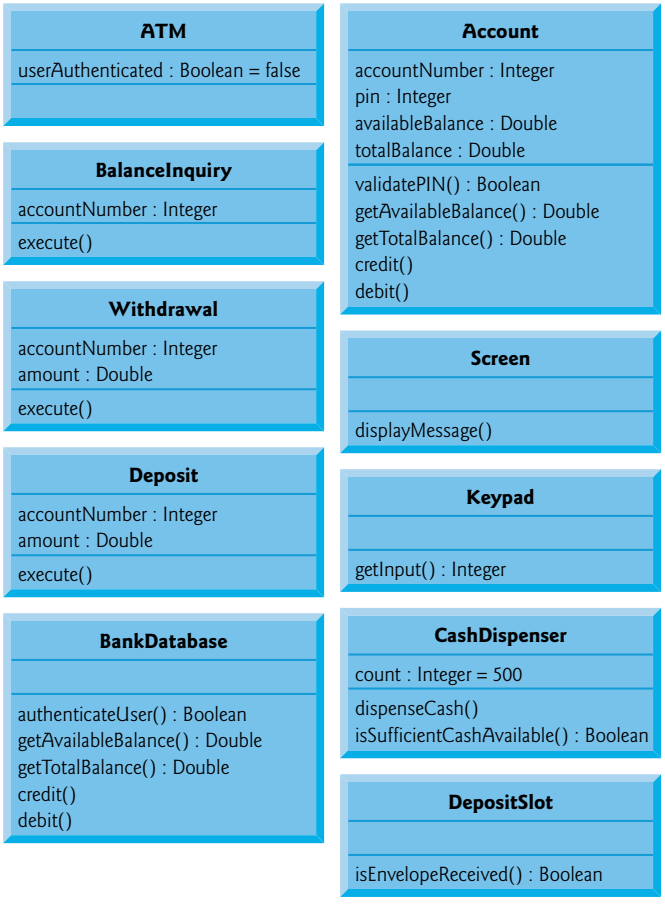


Fig. 25.17 | Classes in the ATM system with attributes and operations.

The UML represents operations (which are implemented as member functions in C++) by listing the operation name, followed by a comma-separated list of parameters in parentheses, a colon and the return type:

operationName(*parameter1*, *parameter2*, ..., *parameterN*) : *return type*

Each parameter in the comma-separated parameter list consists of a parameter name, followed by a colon and the parameter type:

parameterName : *parameterType*

For the moment, we do not list the operations' parameters—we'll identify and model the parameters of some of the operations shortly. For some, we do not yet know the return

types, so we also omit them from the diagram. These omissions are perfectly normal at this point. As our design and implementation proceed, we'll add the remaining return types.

Operations of Class `BankDatabase` *and Class* `Account`

Figure 25.16 lists the phrase "authenticates a user" next to class `BankDatabase`—the database is the object that contains the account information necessary to determine whether the account number and PIN entered by a user match those of an account held at the bank. Therefore, class `BankDatabase` needs an operation that provides an authentication service to the ATM. We place the operation `authenticateUser` in the third compartment of class `BankDatabase` (Fig. 25.17). However, an object of class `Account`, not class `BankDatabase`, stores the account number and PIN that must be accessed to authenticate a user, so class `Account` must provide a service to validate a PIN obtained through user input against a PIN stored in an `Account` object. Therefore, we add a `validatePIN` operation to class `Account`. We specify return type `Boolean` for the `authenticateUser` and `validatePIN` operations. Each operation returns a value indicating either that the operation was successful in performing its task (i.e., a return value of `true`) or that it was not (i.e., a return value of `false`).

Figure 25.16 lists several additional verb phrases for class `BankDatabase`: "retrieves an account balance," "credits a deposit amount to an account" and "debits a withdrawal amount from an account." Like "authenticates a user," these remaining phrases refer to services that the database must provide to the ATM, because the database holds all the account data used to authenticate a user and perform ATM transactions. However, objects of class `Account` actually perform the operations to which these phrases refer. Thus, we assign an operation to both class `BankDatabase` and class `Account` to correspond to each of these phrases. Recall from Section 25.4 that, because a bank account contains sensitive information, we do *not* allow the ATM to access accounts directly. The database acts as an intermediary between the ATM and the account data, thus preventing unauthorized access. As we'll see in Section 25.8, class `ATM` invokes the operations of class `BankDatabase`, each of which in turn invokes the operation with the same name in class `Account`.

The phrase "retrieves an account balance" suggests that classes `BankDatabase` and `Account` each need a `getBalance` operation. However, recall that we created two attributes in class `Account` to represent a balance—`availableBalance` and `totalBalance`. A balance inquiry requires access to both balance attributes so that it can display them to the user, but a withdrawal needs to check only the value of `availableBalance`. To allow objects in the system to obtain each balance attribute individually, we add operations `getAvailableBalance` and `getTotalBalance` to the third compartment of classes `BankDatabase` and `Account` (Fig. 25.17). We specify a return type of `Double` for each of these operations, because the balance attributes which they retrieve are of type `Double`.

The phrases "credits a deposit amount to an account" and "debits a withdrawal amount from an account" indicate that classes `BankDatabase` and `Account` must perform operations to update an account during a deposit and withdrawal, respectively. We therefore assign `credit` and `debit` operations to classes `BankDatabase` and `Account`. You may recall that crediting an account (as in a deposit) adds an amount only to the `totalBalance` attribute. Debiting an account (as in a withdrawal), on the other hand, subtracts the amount from both balance attributes. We hide these implementation details inside class `Account`. This is a good example of encapsulation and information hiding.

If this were a real ATM system, classes `BankDatabase` and `Account` would also provide a set of operations to allow another banking system to update a user's account balance after

either confirming or rejecting all or part of a deposit. Operation `confirmDepositAmount`, for example, would add an amount to the `availableBalance` attribute, thus making deposited funds available for withdrawal. Operation `rejectDepositAmount` would subtract an amount from the `totalBalance` attribute to indicate that a specified amount, which had recently been deposited through the ATM and added to the `totalBalance`, was not found in the deposit envelope. The bank would invoke this operation after determining either that the user failed to include the correct amount of cash or that any checks did not clear (i.e, they "bounced"). While adding these operations would make our system more complete, we do not include them in our class diagrams or our implementation because they are beyond the scope of the case study.

Operations of Class `Screen`

Class `Screen` "displays a message to the user" at various times in an ATM session. All visual output occurs through the screen of the ATM. The requirements document describes many types of messages (e.g., a welcome message, an error message, a thank you message) that the screen displays to the user. The requirements document also indicates that the screen displays prompts and menus to the user. However, a prompt is really just a message describing what the user should input next, and a menu is essentially a type of prompt consisting of a series of messages (i.e., menu options) displayed consecutively. Therefore, rather than assign class `Screen` an individual operation to display each type of message, prompt and menu, we simply create one operation that can display any message specified by a parameter. We place this operation (`displayMessage`) in the third compartment of class `Screen` in our class diagram (Fig. 25.17). We do not worry about the parameter of this operation at this time—we model the parameter later in this section.

Operations of Class `Keypad`

From the phrase "receives numeric input from the user" listed by class `Keypad` in Fig. 25.16, we conclude that class `Keypad` should perform a `getInput` operation. Because the ATM's keypad, unlike a computer keyboard, contains only the numbers 0–9, we specify that this operation returns an integer value. Recall from the requirements document that in different situations the user may be required to enter a different type of number (e.g., an account number, a PIN, the number of a menu option, a deposit amount as a number of cents). Class `Keypad` simply obtains a numeric value for a client of the class—it does *not* determine whether the value meets any specific criteria. Any class that uses this operation must verify that the user enters appropriate numbers, and if not, display error messages via class `Screen`). [*Note:* When we implement the system, we simulate the ATM's keypad with a computer keyboard, and for simplicity we assume that the user does not enter nonnumeric input using keys on the computer keyboard that do not appear on the ATM's keypad.]

Operations of Class `CashDispenser` *and Class* `DepositSlot`

Figure 25.16 lists "dispenses cash" for class `CashDispenser`. Therefore, we create operation `dispenseCash` and list it under class `CashDispenser` in Fig. 25.17. Class `CashDispenser` also "indicates whether it contains enough cash to satisfy a withdrawal request." Thus, we include `isSufficientCashAvailable`, an operation that returns a value of UML type `Boolean`, in class `CashDispenser`. Figure 25.16 also lists "receives a deposit envelope" for class `DepositSlot`. The deposit slot must indicate whether it received an envelope, so we place an operation `isEnvelopeReceived`, which returns a `Boolean` value, in the third

compartment of class `DepositSlot`. [*Note:* A real hardware deposit slot would most likely send the ATM a signal to indicate that an envelope was received. We simulate this behavior, however, with an operation in class `DepositSlot` that class `ATM` can invoke to find out whether the deposit slot received an envelope.]

Operations of Class `ATM`

We do not list any operations for class `ATM` at this time. We are not yet aware of any services that class `ATM` provides to other classes in the system. When we implement the system with C++ code, however, operations of this class, and additional operations of the other classes in the system, may emerge.

Identifying and Modeling Operation Parameters

So far, we've not been concerned with the parameters of our operations—we've attempted to gain only a basic understanding of the operations of each class. Let's now take a closer look at some operation parameters. We identify an operation's parameters by examining what data the operation requires to perform its assigned task.

Consider the `authenticateUser` operation of class `BankDatabase`. To authenticate a user, this operation must know the account number and PIN supplied by the user. Thus we specify that operation `authenticateUser` takes integer parameters `userAccountNumber` and `userPIN`, which the operation must compare to the account number and PIN of an `Account` object in the database. We prefix these parameter names with "user" to avoid confusion between the operation's parameter names and the attribute names that belong to class `Account`. We list these parameters in the class diagram in Fig. 25.18 that models only class `BankDatabase`. [*Note:* It's perfectly normal to model only one class in a class diagram. In this case, we are most concerned with examining the parameters of this one class in particular, so we omit the other classes. In class diagrams later in the case study, in which parameters are no longer the focus of our attention, we omit the parameters to save space. Remember, however, that the operations listed in these diagrams still have parameters.]

Recall that the UML models each parameter in an operation's comma-separated parameter list by listing the parameter name, followed by a colon and the parameter type (in UML notation). Figure 25.18 thus specifies that operation `authenticateUser` takes two parameters—`userAccountNumber` and `userPIN`, both of type `Integer`. When we implement the system in C++, we'll represent these parameters with `int` values.

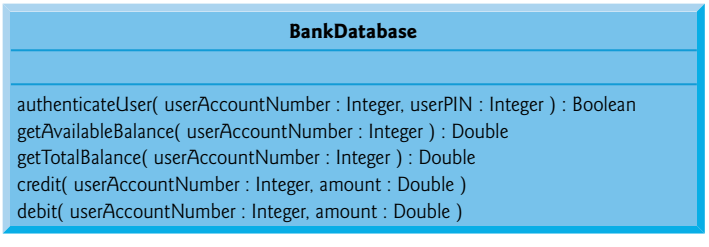


Fig. 25.18 | Class `BankDatabase` with operation parameters.

Class `BankDatabase` operations `getAvailableBalance`, `getTotalBalance`, `credit` and `debit` also each require a `userAccountNumber` parameter to identify the account to

which the database must apply the operations, so we include these parameters in the class diagram of Fig. 25.18. In addition, operations `credit` and `debit` each require a `Double` parameter `amount` to specify the amount of money to be credited or debited, respectively.

The class diagram in Fig. 25.19 models the parameters of class `Account`'s operations. Operation `validatePIN` requires only a `userPIN` parameter, which contains the user-specified PIN to be compared with the PIN associated with the account. Like their counterparts in class `BankDatabase`, operations `credit` and `debit` in class `Account` each require a `Double` parameter `amount` that indicates the amount of money involved in the operation. Operations `getAvailableBalance` and `getTotalBalance` in class `Account` require no additional data to perform their tasks. Class `Account`'s operations do not require an account number parameter—each of these operations can be invoked only on a specific `Account` object, so including a parameter to specify an `Account` is unnecessary.

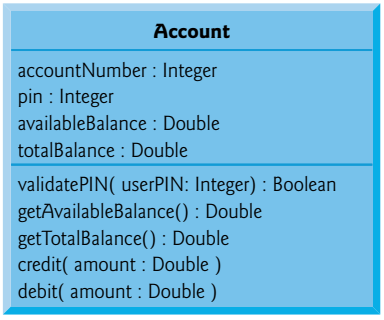


Fig. 25.19 | Class `Account` with operation parameters.

Figure 25.20 models class `Screen` with a parameter specified for operation `displayMessage`. This operation requires only a `String` parameter `message` that indicates the text to be displayed. Recall that the parameter types listed in our class diagrams are in UML notation, so the `String` type listed in Fig. 25.20 refers to the UML type. When we implement the system in C++, we'll in fact use a C++ `string` object to represent this parameter.

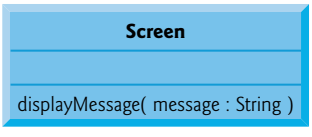


Fig. 25.20 | Class `Screen` with operation parameters.

The class diagram in Fig. 25.21 specifies that operation `dispenseCash` of class `CashDispenser` takes a `Double` parameter `amount` to indicate the amount of cash (in dollars) to be dispensed. Operation `isSufficientCashAvailable` also takes a `Double` parameter `amount` to indicate the amount of cash in question.

We do not discuss parameters for operation `execute` of classes `BalanceInquiry`, `Withdrawal` and `Deposit`, operation `getInput` of class `Keypad` and operation `isEnvelopeReceived` of class `DepositSlot`. At this point in our design process, we cannot determine

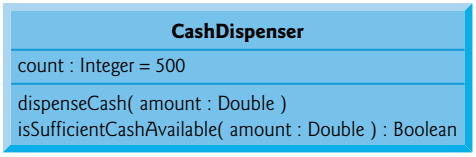


Fig. 25.21 | Class `CashDispenser` with operation parameters.

whether these operations require additional data to perform their tasks, so we leave their parameter lists empty. As we progress through the case study, we may decide to add parameters to these operations.

In this section, we've determined many of the operations performed by the classes in the ATM system. We've identified the parameters and return types of some of the operations. As we continue our design process, the number of operations belonging to each class may vary—we might find that new operations are needed or that some current operations are unnecessary—and we might determine that some of our class operations need additional parameters and different return types.

Self-Review Exercises for Section 25.7

25.14 Which of the following is not a behavior?
a) reading data from a file
b) printing output
c) text output
d) obtaining input from the user

25.15 If you were to add to the ATM system an operation that returns the `amount` attribute of class `Withdrawal`, how and where would you specify this operation in the class diagram of Fig. 25.17?

25.16 Describe the meaning of the following operation listing that might appear in a class diagram for an object-oriented design of a calculator:

```
add( x : Integer, y : Integer ) : Integer
```

25.8 Indicating Collaboration Among Objects

[*Note:* **This section can be studied after Chapter 7.**]

In this section, we concentrate on the collaborations (interactions) among objects in our ATM system. When two objects communicate with each other to accomplish a task, they are said to **collaborate**—they do this by invoking one another's operations. A **collaboration** consists of an object of one class sending a **message** to an object of another class. Messages are sent in C++ via member-function calls.

In Section 25.7, we determined many of the *operations* of the system's classes. Next, we concentrate on the *messages* that invoke these operations. To identify the collaborations, we return to the requirements document in Section 25.3. Recall that this document specifies the range of activities that occur during an ATM session (e.g., authenticating a user, performing transactions). The steps used to describe how the system must perform each of these tasks are our first indication of the collaborations in our system. As we proceed through this and the remaining sections, we may discover additional collaborations.

Identifying the Collaborations in a System

We identify the collaborations in the system by carefully reading the requirements document sections that specify what the ATM should do to authenticate a user and to perform each transaction type. For each action or step described, we decide which objects in our system must interact to achieve the desired result. We identify one object as the *sending object* (i.e., the object that sends the message) and another as the *receiving object* (i.e., the object that offers that operation to clients of the class). We then select one of the receiving object's operations (identified in Section 25.7) that must be invoked by the sending object to produce the proper behavior. For example, the ATM displays a welcome message when idle. We know that an object of class `Screen` displays a message to the user via its `displayMessage` operation. Thus, we decide that the system can display a welcome message by employing a collaboration between the `ATM` and the `Screen` in which the `ATM` sends a `displayMessage` message to the `Screen` by invoking the `displayMessage` operation of class `Screen`. [*Note:* To avoid repeating the phrase "an object of class…," we refer to each object simply by using its class name preceded by an article ("a," "an" or "the")—for example, "the `ATM`" refers to an object of class `ATM`.]

Figure 25.22 lists the collaborations that can be derived from the requirements document. For each sending object, we list the collaborations in the order in which they are discussed in the requirements document. We list each collaboration involving a unique sender, message and recipient only once, even though the collaboration may occur several times during an ATM session. For example, the first row in Fig. 25.22 indicates that the `ATM` collaborates with the `Screen` whenever the `ATM` needs to display a message to the user.

An object of class...	sends the message...	to an object of class...
`ATM`	`displayMessage`	`Screen`
	`getInput`	`Keypad`
	`authenticateUser`	`BankDatabase`
	`execute`	`BalanceInquiry`
	`execute`	`Withdrawal`
	`execute`	`Deposit`
`BalanceInquiry`	`getAvailableBalance`	`BankDatabase`
	`getTotalBalance`	`BankDatabase`
	`displayMessage`	`Screen`
`Withdrawal`	`displayMessage`	`Screen`
	`getInput`	`Keypad`
	`getAvailableBalance`	`BankDatabase`
	`isSufficientCashAvailable`	`CashDispenser`
	`debit`	`BankDatabase`
	`dispenseCash`	`CashDispenser`
`Deposit`	`displayMessage`	`Screen`
	`getInput`	`Keypad`
	`isEnvelopeReceived`	`DepositSlot`
	`credit`	`BankDatabase`

Fig. 25.22 | Collaborations in the ATM system. (Part 1 of 2.)

An object of class...	sends the message...	to an object of class...
BankDatabase	validatePIN	Account
	getAvailableBalance	Account
	getTotalBalance	Account
	debit	Account
	credit	Account

Fig. 25.22 | Collaborations in the ATM system. (Part 2 of 2.)

Let's consider the collaborations in Fig. 25.22. Before allowing a user to perform any transactions, the ATM must prompt the user to enter an account number, then to enter a PIN. It accomplishes each of these tasks by sending a `displayMessage` message to the `Screen`. Both of these actions refer to the same collaboration between the `ATM` and the `Screen`, which is already listed in Fig. 25.22. The `ATM` obtains input in response to a prompt by sending a `getInput` message to the `Keypad`. Next, the ATM must determine whether the user-specified account number and PIN match those of an account in the database. It does so by sending an `authenticateUser` message to the `BankDatabase`. Recall that the `BankDatabase` cannot authenticate a user directly—only the user's `Account` (i.e., the `Account` that contains the account number specified by the user) can access the user's PIN to authenticate the user. Figure 25.22 therefore lists a collaboration in which the `BankDatabase` sends a `validatePIN` message to an `Account`.

After the user is authenticated, the `ATM` displays the main menu by sending a series of `displayMessage` messages to the `Screen` and obtains input containing a menu selection by sending a `getInput` message to the `Keypad`. We've already accounted for these collaborations. After the user chooses a type of transaction to perform, the `ATM` executes the transaction by sending an `execute` message to an object of the appropriate transaction class (i.e., a `BalanceInquiry`, a `Withdrawal` or a `Deposit`). For example, if the user chooses to perform a balance inquiry, the `ATM` sends an `execute` message to a `BalanceInquiry`.

Further examination of the requirements document reveals the collaborations involved in executing each transaction type. A `BalanceInquiry` retrieves the amount of money available in the user's account by sending a `getAvailableBalance` message to the `BankDatabase`, which responds by sending a `getAvailableBalance` message to the user's `Account`. Similarly, the `BalanceInquiry` retrieves the amount of money on deposit by sending a `getTotalBalance` message to the `BankDatabase`, which sends the same message to the user's `Account`. To display both measures of the user's balance at the same time, the `BalanceInquiry` sends a `displayMessage` message to the `Screen`.

A `Withdrawal` sends the `Screen` several `displayMessage` messages to display a menu of standard withdrawal amounts (i.e., $20, $40, $60, $100, $200). The `Withdrawal` sends the `Keypad` a `getInput` message to obtain the user's menu selection, then determines whether the requested withdrawal amount is less than or equal to the user's account balance. The `Withdrawal` can obtain the amount of money available in the account by sending the `BankDatabase` a `getAvailableBalance` message. The `Withdrawal` then tests whether the cash dispenser contains enough cash by sending the `CashDispenser` an `isSufficientCashAvailable` message. A `Withdrawal` sends the `BankDatabase` a `debit`

message to decrease the user's account balance. The `BankDatabase` sends the same message to the appropriate `Account`. Recall that debiting funds from an `Account` decreases both the `totalBalance` and the `availableBalance`. To dispense the requested amount of cash, the `Withdrawal` sends the `CashDispenser` a `dispenseCash` message. Finally, the `Withdrawal` sends a `displayMessage` message to the `Screen`, instructing the user to take the cash.

A `Deposit` responds to an `execute` message first by sending a `displayMessage` message to the `Screen` to prompt the user for a deposit amount. The `Deposit` sends a `getInput` message to the `Keypad` to obtain the user's input. The `Deposit` then sends a `displayMessage` message to the `Screen` to tell the user to insert a deposit envelope. To determine whether the deposit slot received an incoming deposit envelope, the `Deposit` sends an `isEnvelopeReceived` message to the `DepositSlot`. The `Deposit` updates the user's account by sending a `credit` message to the `BankDatabase`, which subsequently sends a `credit` message to the user's `Account`. Recall that crediting funds to an `Account` increases the `totalBalance` but not the `availableBalance`.

Interaction Diagrams

Now that we've identified possible collaborations between the objects in our ATM system, let's graphically model these interactions using the UML. Several types of **interaction diagrams** model the behavior of a system by showing how objects interact with one another. The **communication diagram** emphasizes which objects participate in collaborations. [*Note:* Communication diagrams were called **collaboration diagrams** in earlier versions of the UML.] Like the communication diagram, the **sequence diagram** shows collaborations among objects, but it emphasizes *when* messages are sent between objects *over time*.

Communication Diagrams

Figure 25.23 shows a communication diagram that models the `ATM` executing a `BalanceInquiry`. Objects are modeled in the UML as rectangles containing names in the form `objectName : ClassName`. In this example, which involves only one object of each type, we disregard the object name and list only a colon followed by the class name. [*Note:* Specifying the name of each object in a communication diagram is recommended when modeling multiple objects of the same type.] Communicating objects are connected with solid lines, and messages are passed between objects along these lines in the direction shown by arrows. The name of the message, which appears next to the arrow, is the name of an operation (i.e., a member function) belonging to the receiving object—think of the name as a service that the receiving object provides to sending objects (its "clients").

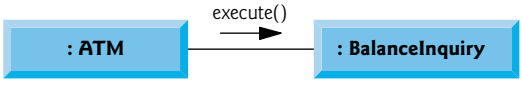


Fig. 25.23 | Communication diagram of the ATM executing a balance inquiry.

The solid filled arrow in Fig. 25.23 represents a message—or **synchronous call**—in the UML and a function call in C++. This arrow indicates that the flow of control is from the sending object (the `ATM`) to the receiving object (a `BalanceInquiry`). *Since this is a synchronous call, the sending object may not send another message, or do anything at all, until the receiving object processes the message and returns control to the sending object—the sender just*

waits. For example, in Fig. 25.23, the `ATM` calls member function `execute` of a `BalanceInquiry` and may not send another message until `execute` has finished and returns control to the `ATM`. [*Note:* If this were an **asynchronous call**, represented by a stick arrowhead, the sending object would not have to wait for the receiving object to return control—it would continue sending additional messages immediately following the asynchronous call. Asynchronous calls often can be implemented in C++ using platform-specific libraries provided with your compiler. Such techniques are beyond the scope of this book.]

Sequence of Messages in a Communication Diagram

Figure 25.24 shows a communication diagram that models the interactions among objects in the system when an object of class `BalanceInquiry` executes. We assume that the object's `accountNumber` attribute contains the account number of the current user. The collaborations in Fig. 25.24 begin after the `ATM` sends an `execute` message to a `BalanceInquiry` (i.e., the interaction modeled in Fig. 25.23). The number to the left of a message name indicates the order in which the message is passed. The **sequence of messages** in a communication diagram progresses in numerical order from least to greatest. In this diagram, the numbering starts with message `1` and ends with message `3`. The `BalanceInquiry` first sends a `getAvailableBalance` message to the `BankDatabase` (message `1`), then sends a `getTotalBalance` message to the `BankDatabase` (message `2`). Within the parentheses following a message name, we can specify a comma-separated list of the names of the parameters sent with the message (i.e., arguments in a C++ function call)—the `BalanceInquiry` passes attribute `accountNumber` with its messages to the `BankDatabase` to indicate which `Account`'s balance information to retrieve. Recall from Fig. 25.18 that operations `getAvailableBalance` and `getTotalBalance` of class `BankDatabase` each require a parameter to identify an account. The `BalanceInquiry` next displays the `availableBalance` and the `totalBalance` to the user by passing a `displayMessage` message to the `Screen` (message `3`) that includes a parameter indicating the `message` to be displayed.

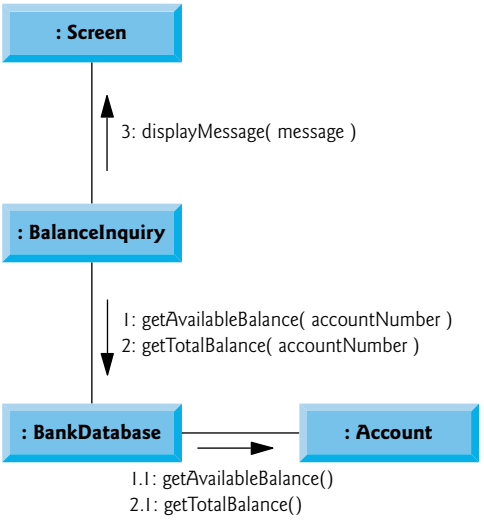


Fig. 25.24 | Communication diagram for executing a balance inquiry.

Figure 25.24 models two additional messages passing from the `BankDatabase` to an `Account` (message `1.1` and message `2.1`). To provide the `ATM` with the two balances of the user's `Account` (as requested by messages `1` and `2`), the `BankDatabase` must pass a `getAvailableBalance` and a `getTotalBalance` message to the user's `Account`. Messages passed within the handling of another message are called **nested messages**. The UML recommends using a decimal numbering scheme to indicate nested messages. For example, message `1.1` is the first message nested in message `1`—the `BankDatabase` passes a `getAvailableBalance` message while processing `BankDatabase`'s message of the same name. [*Note:* If the `BankDatabase` needed to pass a second nested message while processing message `1`, the second message would be numbered `1.2`.] A message may be passed only when all the nested messages from the previous message have been passed—e.g., the `BalanceInquiry` passes message `3` only after messages `2` and `2.1` have been passed, in that order.

The nested numbering scheme used in communication diagrams helps clarify precisely when and in what context each message is passed. For example, if we numbered the messages in Fig. 25.24 using a flat numbering scheme (i.e., `1`, `2`, `3`, `4`, `5`), someone looking at the diagram might not be able to determine that `BankDatabase` passes the `getAvailableBalance` message (message `1.1`) to an `Account` *during* the `BankDatabase`'s processing of message `1`, as opposed to *after* completing the processing of message `1`. The nested decimal numbers make it clear that the second `getAvailableBalance` message (message `1.1`) is passed to an `Account` within the handling of the first `getAvailableBalance` message (message `1`) by the `BankDatabase`.

Sequence Diagrams

Communication diagrams emphasize the participants in collaborations but model their timing a bit awkwardly. A sequence diagram helps model the timing of collaborations more clearly. Figure 25.25 shows a sequence diagram modeling the sequence of interactions that occur when a `Withdrawal` executes. The dotted line extending down from an object's rectangle is that object's **lifeline**, which represents the progression of time. Actions typically occur along an object's lifeline in *chronological order* from top to bottom—an action near the top typically happens before one near the bottom.

Message passing in sequence diagrams is similar to message passing in communication diagrams. A solid arrow with a filled arrowhead extending from the sending object to the receiving object represents a message between two objects. The arrowhead points to an activation on the receiving object's lifeline. An **activation**, shown as a thin vertical rectangle, indicates that an object is executing. When an object returns control, a return message, represented as a dashed line with a stick arrowhead, extends from the activation of the object returning control to the activation of the object that initially sent the message. To eliminate clutter, we omit the return-message arrows—the UML allows this practice to make diagrams more readable. Like communication diagrams, sequence diagrams can indicate message parameters between the parentheses following a message name.

The sequence of messages in Fig. 25.25 begins when a `Withdrawal` prompts the user to choose a withdrawal amount by sending a `displayMessage` message to the `Screen`. The `Withdrawal` then sends a `getInput` message to the `Keypad`, which obtains input from the user. We've already modeled the control logic involved in a `Withdrawal` in the activity diagram of Fig. 25.15, so we do not show this logic in the sequence diagram of Fig. 25.25. Instead, we model the best-case scenario in which the balance of the user's account is greater than or equal to the chosen withdrawal amount, and the cash dispenser contains a

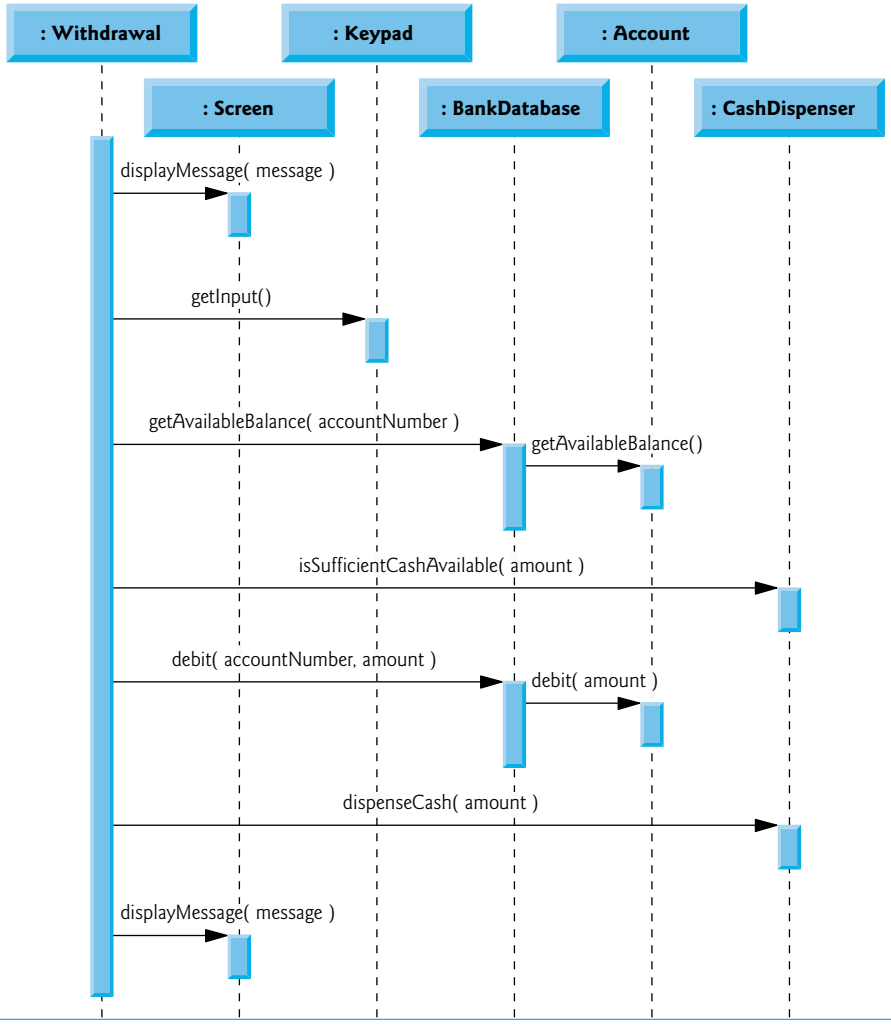


Fig. 25.25 | Sequence diagram that models a `Withdrawal` executing.

sufficient amount of cash to satisfy the request. For information on how to model control logic in a sequence diagram, please refer to the web resources at the end of Section 25.3.

After obtaining a withdrawal amount, the `Withdrawal` sends a `getAvailableBalance` message to the `BankDatabase`, which in turn sends a `getAvailableBalance` message to the user's `Account`. Assuming that the user's account has enough money available to permit the transaction, the `Withdrawal` next sends an `isSufficientCashAvailable` message to the `CashDispenser`. Assuming that there is enough cash available, the `Withdrawal` decreases the balance of the user's account (i.e., both the `totalBalance` and the `availableBalance`) by sending a `debit` message to the `BankDatabase`. The `BankDatabase` responds by sending a `debit` message to the user's `Account`. Finally, the `Withdrawal` sends a `dispenseCash` message to the `CashDispenser` and a `displayMessage` message to the `Screen`, telling the user to remove the cash from the machine.

We've identified the collaborations among the ATM system's objects and modeled some of them using UML interaction diagrams—both communication diagrams and sequence diagrams. In Section 26.2, we enhance the structure of our model to complete a preliminary object-oriented design, then we implement the ATM system in C++.

Self-Review Exercises for Section 25.8

25.17 A(n) _____ consists of an object of one class sending a message to an object of another class.
a) association
b) aggregation
c) collaboration
d) composition

25.18 Which form of interaction diagram emphasizes *what* collaborations occur? Which form emphasizes *when* collaborations occur?

25.19 Create a sequence diagram that models the interactions among objects in the ATM system that occur when a `Deposit` executes successfully, and explain the sequence of messages modeled by the diagram.

25.9 Wrap-Up

In this chapter, you learned how to work from a detailed requirements document to develop an object-oriented design. You worked with six popular types of UML diagrams to graphically model an object-oriented automated teller machine software system. In Section 26.3, we tune the design using inheritance, then completely implement the design in an 850-line C++ application.

Answers to Self-Review Exercises

25.1 Figure 25.26 shows a use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

25.2 b.

25.3 d.

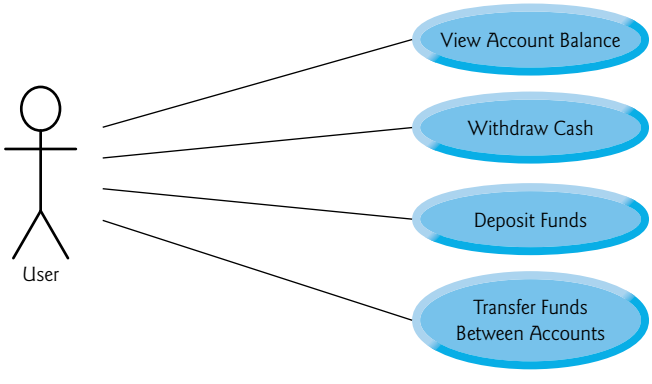


Fig. 25.26 | Use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

25.4 [*Note:* Answers may vary.] Figure 25.27 presents a class diagram that shows some of the composition relationships of a class `Car`.

25.5 c. [*Note:* In a computer network, this relationship could be many-to-many.]

25.6 True.

25.7 Figure 25.28 presents an ATM class diagram including class `Deposit` instead of class `Withdrawal`. Note that `Deposit` does not access `CashDispenser`, but does access `DepositSlot`.

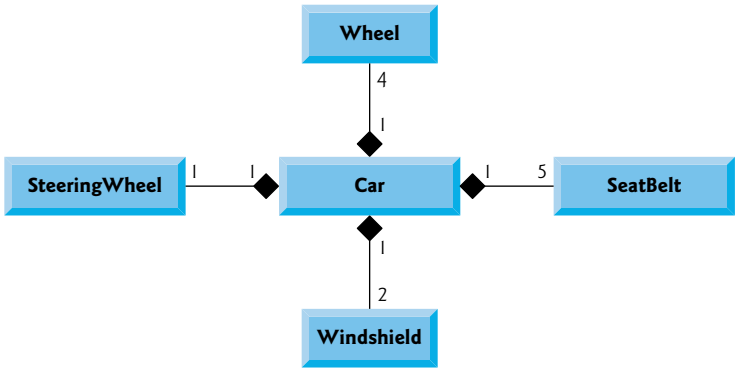


Fig. 25.27 | Class diagram showing composition relationships of a class `Car`.

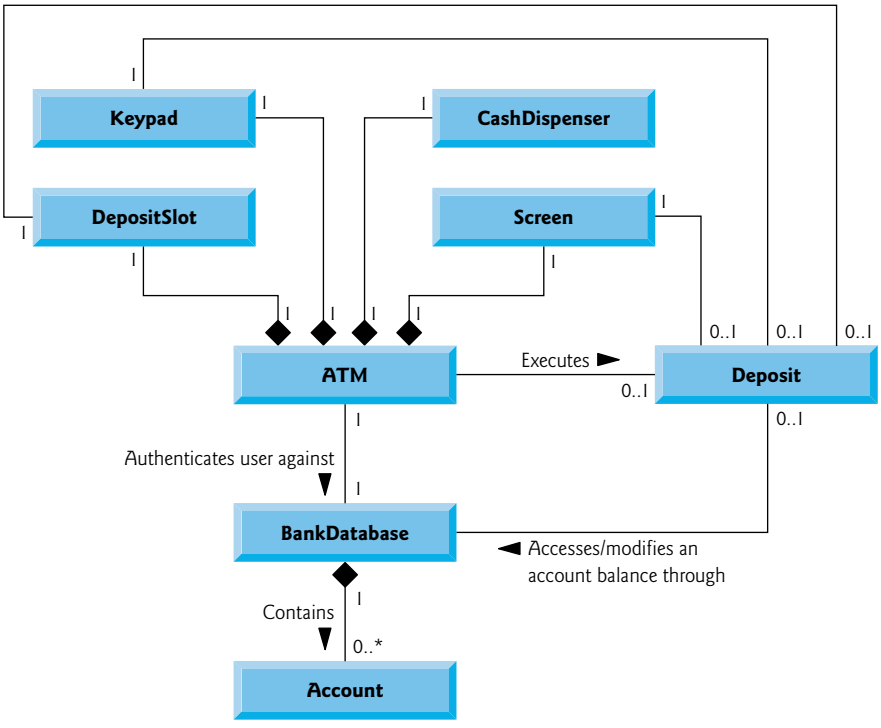


Fig. 25.28 | Class diagram for the ATM system model including class `Deposit`.

25.8 b.

25.9 c. Fly is an operation or behavior of an airplane, not an attribute.

25.10 This indicates that `count` is an `Integer` with an initial value of `500`. This attribute keeps track of the number of bills available in the `CashDispenser` at any given time.

25.11 False. State diagrams model some of the behavior of a system.

25.12 a.

25.13 Figure 25.29's activity diagram models the actions that occur after the user chooses the deposit option from the main menu and before the ATM returns the user to the main menu. Recall that part of receiving a deposit amount from the user involves converting an integer number of cents to a dollar amount. Also recall that crediting a deposit amount to an account involves increasing only the `totalBalance` attribute of the user's `Account` object. The bank updates the `availableBal-`

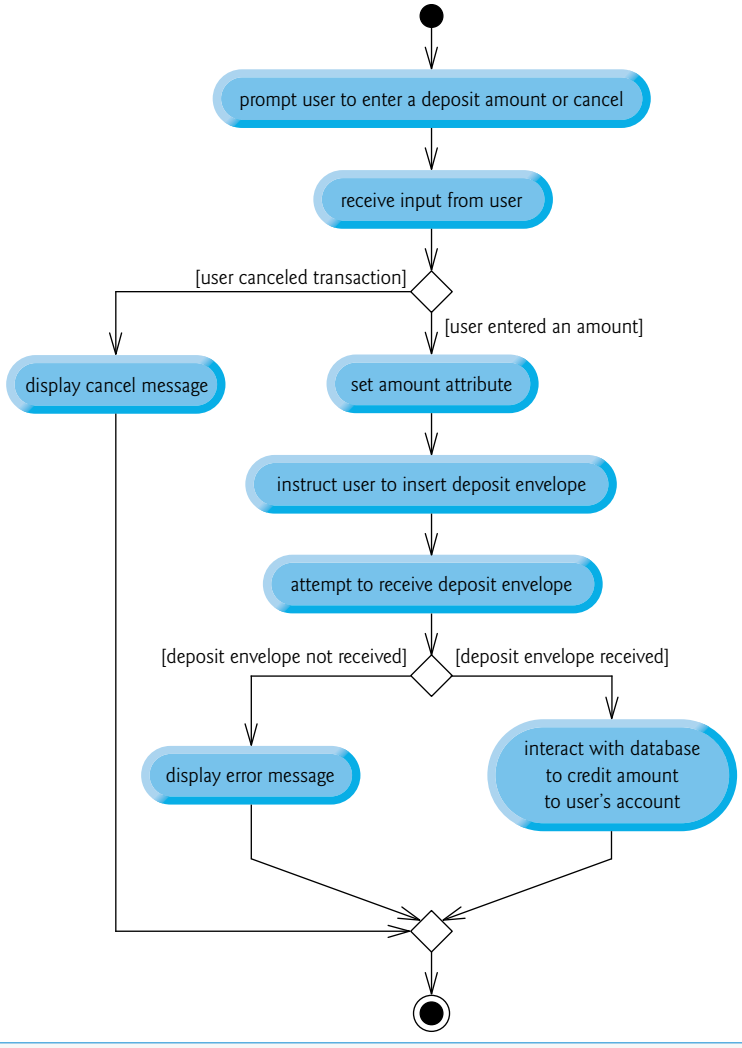


Fig. 25.29 | Activity diagram for a `Deposit` transaction.

ance attribute of the user's `Account` object only after confirming the amount of cash in the deposit envelope and after the enclosed checks clear—this occurs independently of the ATM system.

25.14 c.

25.15 To specify an operation that retrieves the `amount` attribute of class `Withdrawal`, the following operation would be placed in the operation (i.e., third) compartment of class `Withdrawal`:

```
getAmount( ) : Double
```

25.16 This is an operation named `add` that takes integers `x` and `y` as parameters and returns an integer value.

25.17 c.

25.18 Communication diagrams emphasize *what* collaborations occur. Sequence diagrams emphasize *when* collaborations occur.

25.19 Figure 25.30 presents a sequence diagram that models the interactions between objects that occur when a `Deposit` executes successfully. A `Deposit` first sends a `displayMessage` message to the `Screen` to ask the user to enter a deposit amount. Next, it sends a `getInput` message to the `Keypad` to receive input from the user. Then, it instructs the user to insert a deposit envelope by sending a `displayMessage` message to the `Screen`. It then sends an `isEnvelopeReceived` message to the `DepositSlot` to confirm that the deposit envelope has been received. Finally, it increases the `totalBalance` attribute (but not the `availableBalance` attribute) of the user's `Account` by sending a `credit` message to the `BankDatabase`. The `BankDatabase` responds by sending the same message to the user's `Account`.

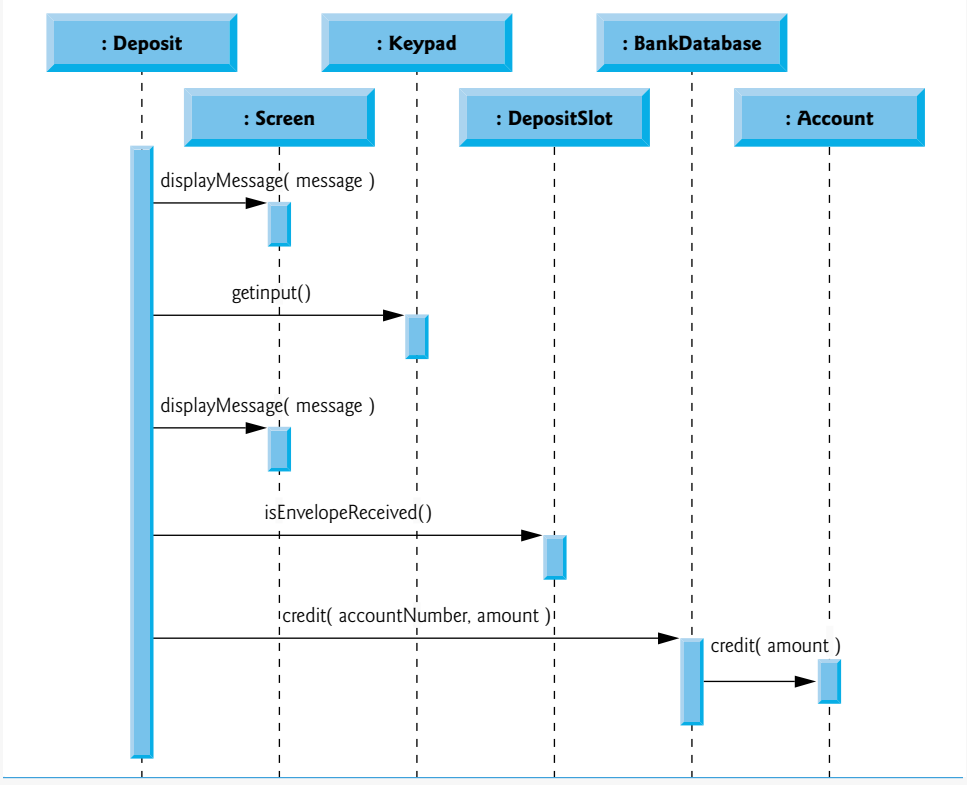


Fig. 25.30 | Sequence diagram that models a `Deposit` executing.