Software architecture and design

What is Software Architecture

Definitions

- The software architecture of a program or computing system is the structure or structures of the system which comprise
  - The software components
  - The externally visible properties of those components
  - The relationships among the components

Architecture defined

Formal Definition

- IEEE 1471-2000
  - Software architecture is the **fundamental organization** of a system, embodied in its **components**, their **relationships** to each other and the environment, and the **principles** governing its design and evolution.

Architecture defined

- Software architecture **encompasses the set of significant decisions** about the organization of a software system
  - Selection of the structural elements and their interfaces by which a system is composed
  - Behavior as specified in collaborations among those elements
  - Composition of these structural and behavioral elements into larger subsystems
  - Architectural style that guides this organization

Architecture defined

- Perry and Wolf, 1992
  - A set of **architectural (or design)** **elements** that have a particular form
- Boehm et al., 1995
  - A software system architecture comprises
    - A collection of software and system **components, connections, and constraints**
    - A collection of **system stakeholders'** need statements
    - A **foundation** which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements

Architecture defined

- Architecture is not a single structure -- no single structure is the architecture

When to think about Architecture

Architecture is Early

- Architecture represents the set of earliest design decisions
  - Hardest to change
  - Most critical to get right
- Architecture is the first design artifact where a system’s quality attributes are addressed

Architecture Drives

Architecture Drives

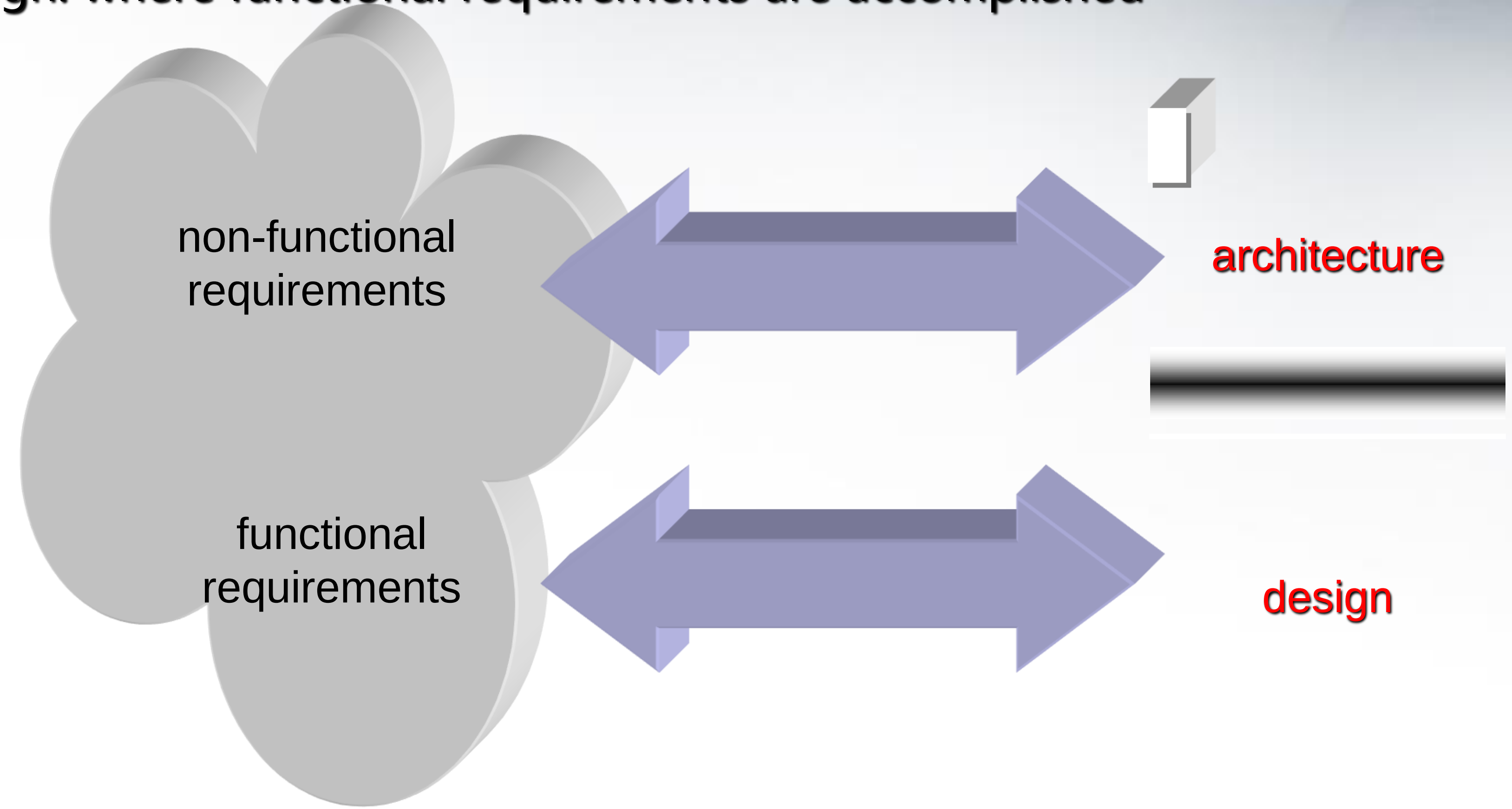
- Architecture not only serves as the blueprint for the system but also the project:
  - Team structure
  - Documentation organization
  - Work breakdown structure
  - Scheduling, planning, budgeting
  - Unit testing, integration
- Architecture establishes the communication and coordination mechanisms among components

Architecture vs. Design

Architecture vs. Design

Architecture: where non-functional decisions are cast, and functional requirements are partitioned

Design: where functional requirements are accomplished



non-functional requirements

architecture

functional requirements

design

Important : this is a general guideline – sometimes the borders are blurred

System Quality Attribute

- Performance
- Availability
- Usability
- Security

End User’s view

- Maintainability
- Portability
- Reusability
- Testability

Developer’s view

- Time To Market
- Cost and Benefits
- Projected life time
- Targeted Market
- Integration with Legacy System
- Roll back Schedule

Business Community view

A list of quality attributes exists in
ISO/IEC 9126-2001 Information Technology – Software Product Quality

Typical Descriptions of Software Architectures

Typical Descriptions of Software Architectures

- "Camelot is based on the client-server model and uses remote procedure calls both locally and remotely to provide communication among applications and servers." [Spector 87]
- "We have chosen a distributed, object-oriented approach to managing information." [Linton 87]
- "The easiest way to make the canonical sequential compiler into a concurrent compiler is to pipeline the execution of the compiler phases over a number of processors." [Seshadri 88]
- "The ARC network [follows] the general network architecture specified by the ISO in the Open Systems Interconnection Reference Model." [Paulk 85]

Architectural Design Level of Software

Architectural Design Level of Software

- Deals with the composition of software systems from module-scale elements
  - Gross decomposition of required function
    - What patterns of organization are useful?
    - Which organization fits the application best?
  - Assignment of function to design elements
    - What are the elements?
    - How do the elements interact?
  - Emergent system properties
    - Scaling and performance: Capacities, balance, schedules
    - Security
    - ...
  - Selection among design alternatives
    - Which implementations of elements will work best?

Issues for architecture and programs differ

Architectural Design Task

Issues for architecture and programs differ

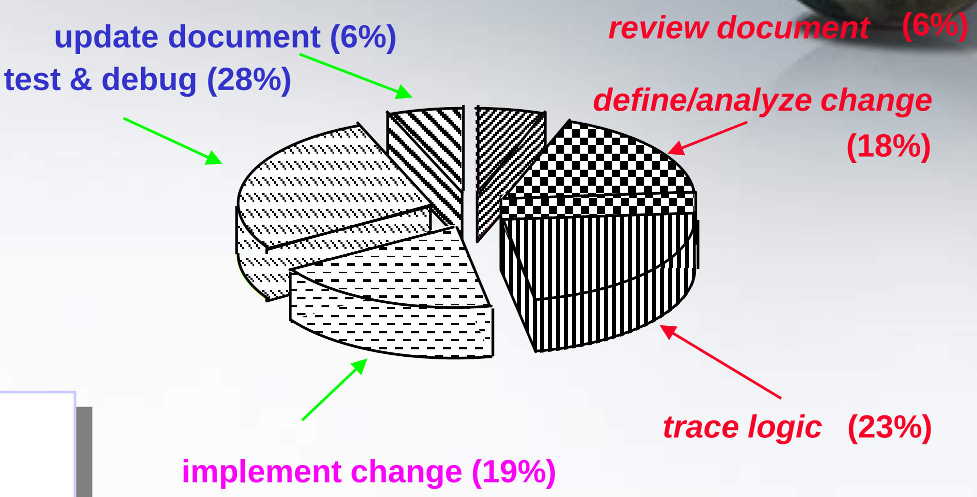
Architecture	*Programs*
interactions among parts	implementations of parts
structural properties	computational properties
declarative	operational
mostly static	mostly dynamic
system-level performance	algorithmic performance
outside module boundary	inside module boundary
composition of subsystems	copy code or call libraries

Promised Benefits of Architectural Modeling

Promised Benefits of Architectural Modeling

- Clarify design intentions
  - Intended architecture is often lost. It's mostly informal, it's hard to communicate anyhow.
- Provide basis for analysis in design
  - Engineering design entails performance prediction and design tuning. Routine practice.
- Improve maintenance
  - Over half of maintenance effort goes into figuring out just what's there.
- Address the hard questions
  - Even without formal methods, explicit architectural modelling can uncover fuzzy requirements, thinking, and design approaches

Promised Benefits of Architectural Modeling



Requirements

Architecture

Design

Code/Integ

Test/Accept

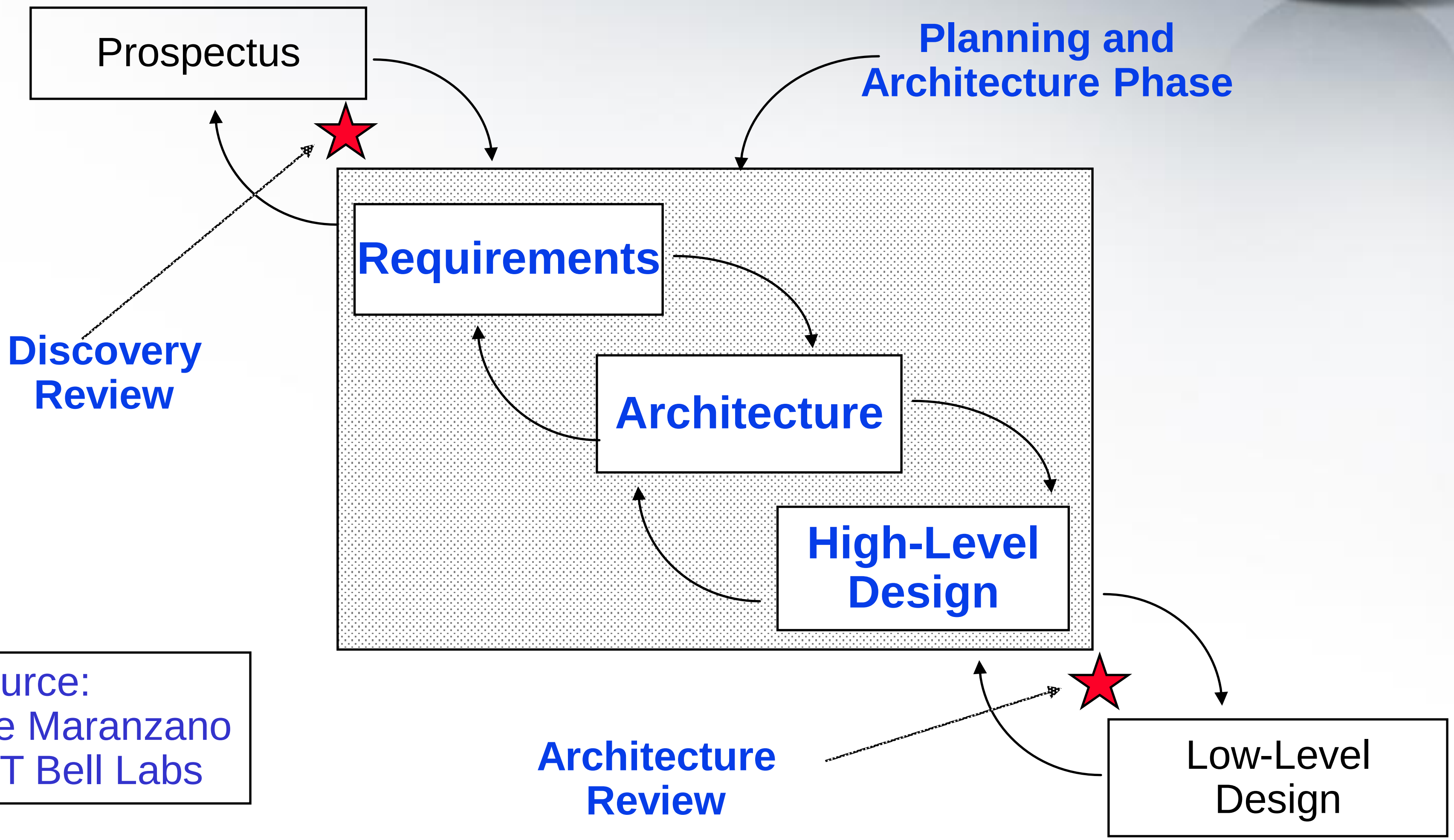
Maintenance

- Clarify intentions
- Make decisions and implications explicit
- Permit system-level analysis

Reduce maintenance costs, directly and indirectly

Architectural Design Reviews

Architectural Design Reviews



Source:
Joe Maranzano
ATT Bell Labs

Architectural Structure: Components, Connectors, Systems

Architectural Structure: Components, Connectors, Systems

- Components
  - – computational elements
- Ports
  - – interface points for components
- Connectors
  - – interactions between components
- Roles
  - – interface points for connectors
- Systems
  - – graphs of component and connectors

Architectural Design Process

Architectural Design Process

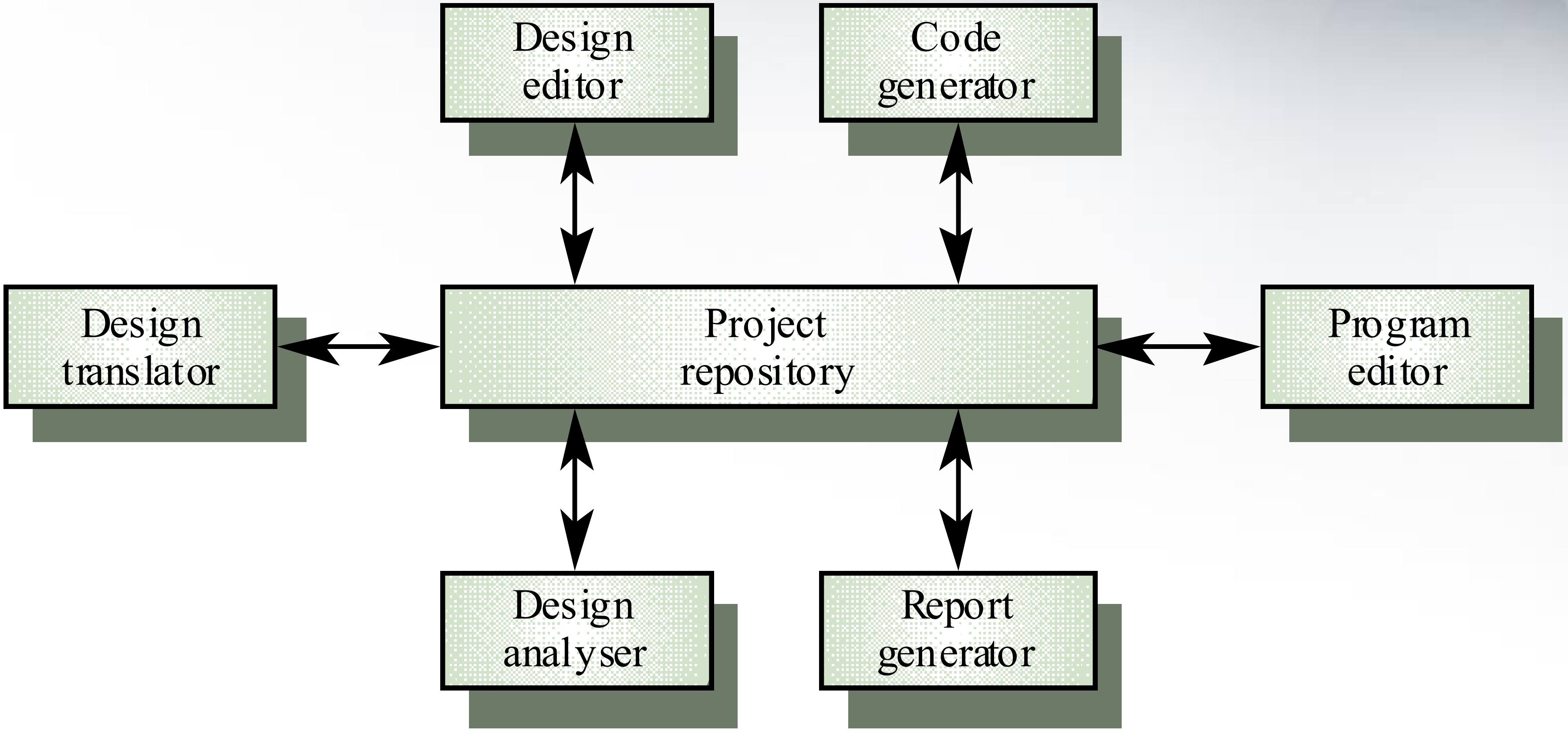
- **System structuring**
  - – system decomposed into several subsystems
  - – subsystem communication is established
- **Control modeling**
  - – model of control relationships among system components is established
- **Modular decomposition**
  - – identified subsystems decomposed into modules

Architectural Models

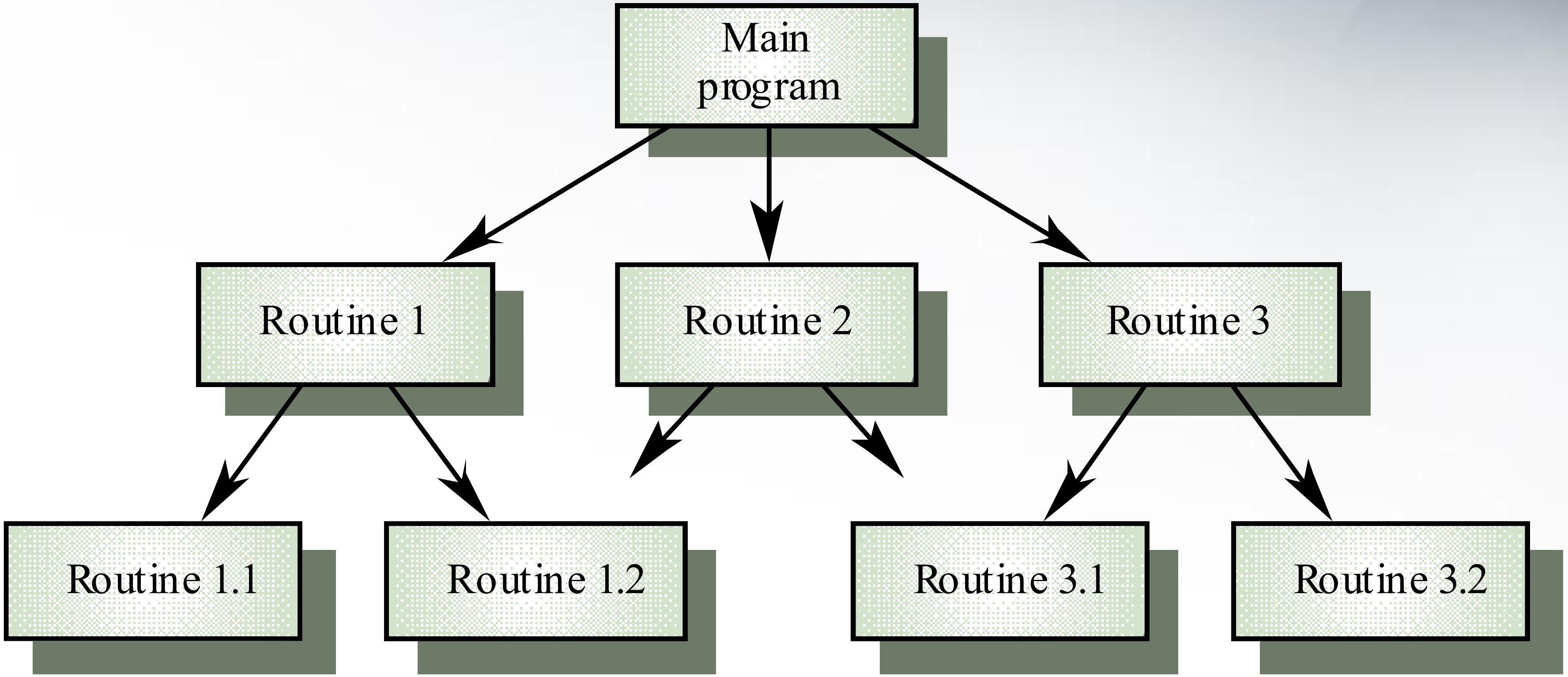
Architectural Models

- **Static structural model**
  - shows major system components
- **Dynamic process model**
  - shows process structure of the system
- **Interface model**
  - defines subsystem interfaces
- **Relationships model**
  - data flow or control flow diagrams

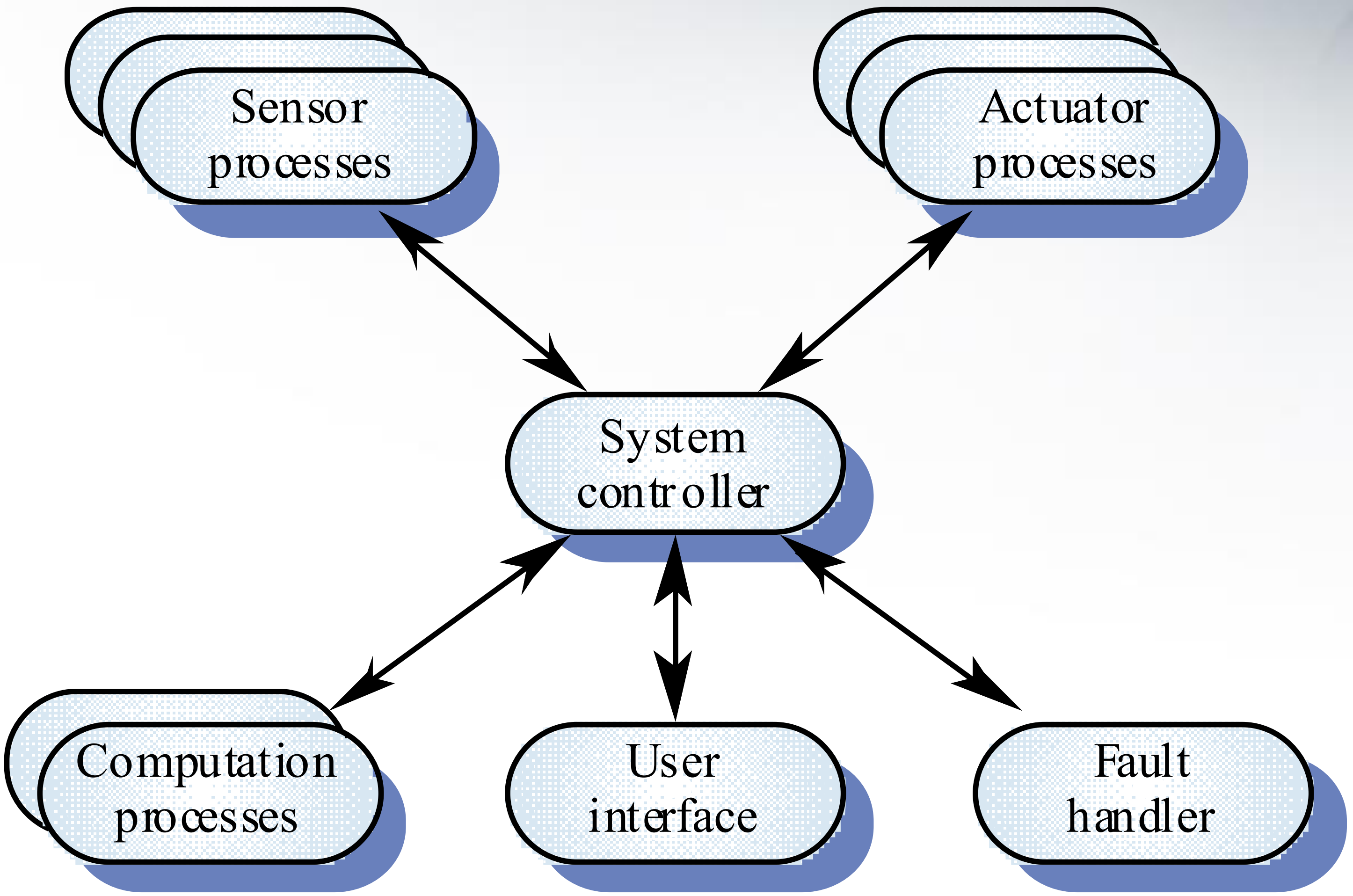
CASE Repository Model



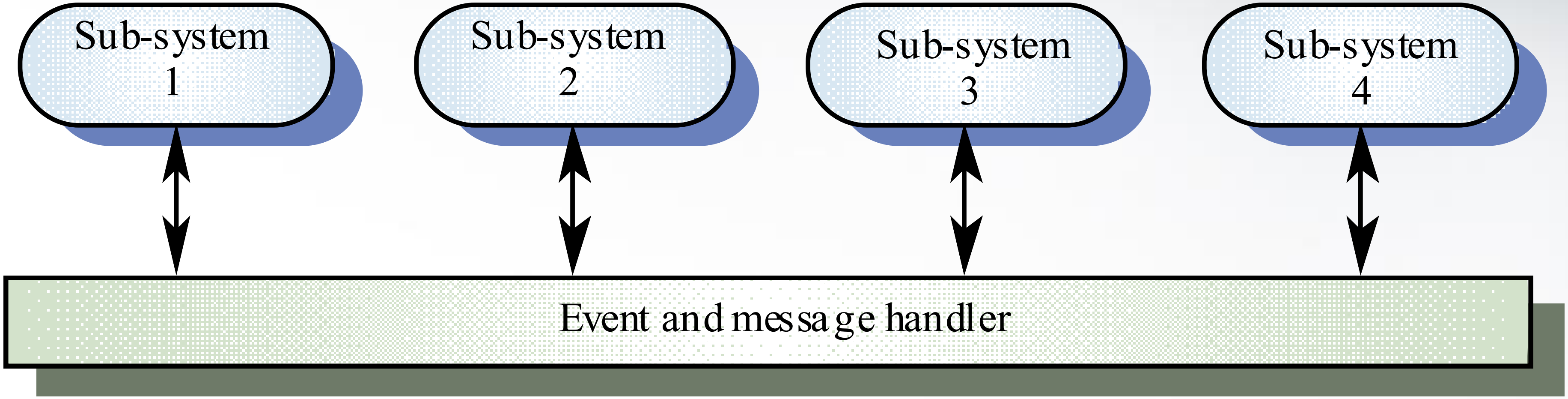
Call-Return Model



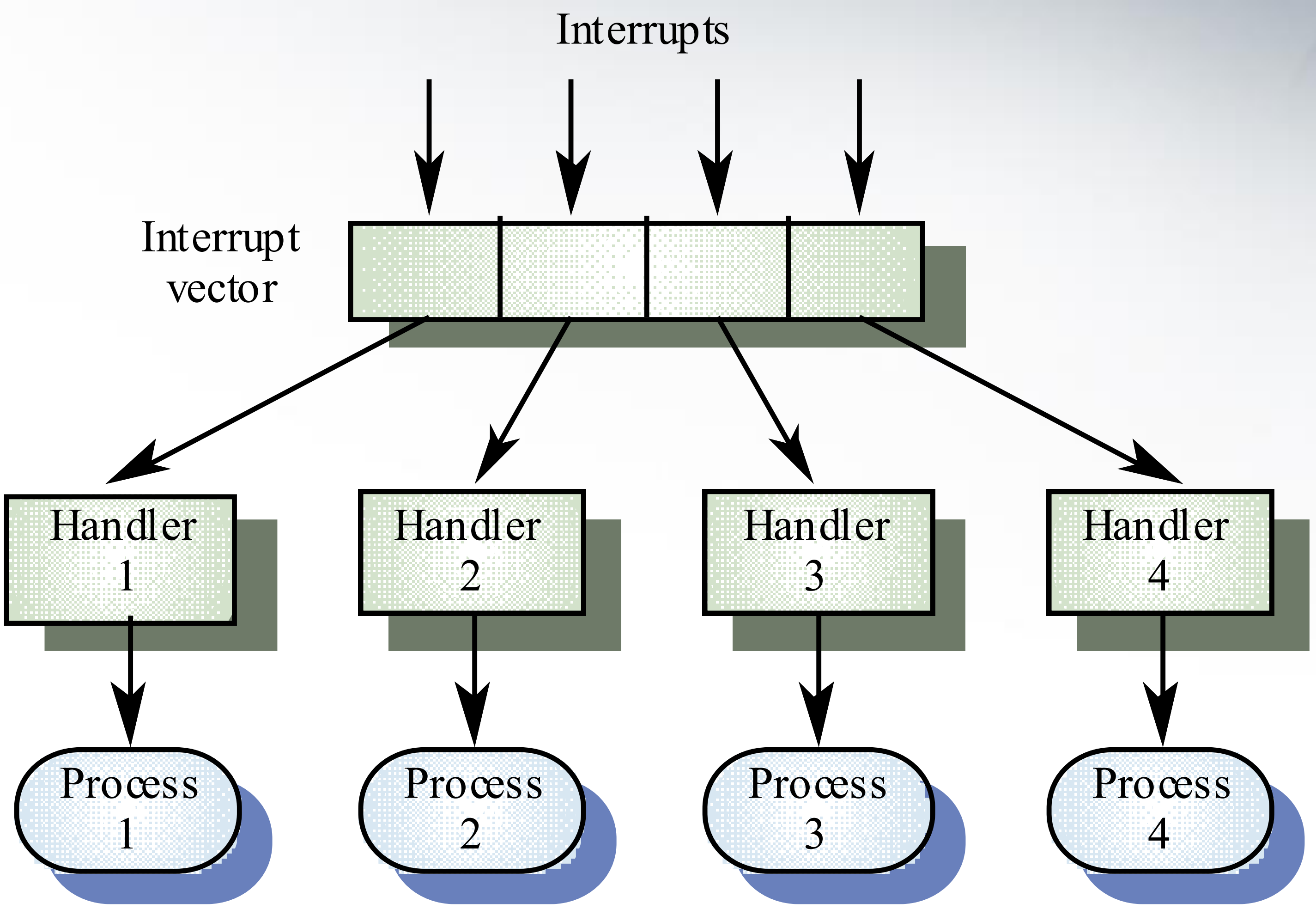
Real-Time System Control Model



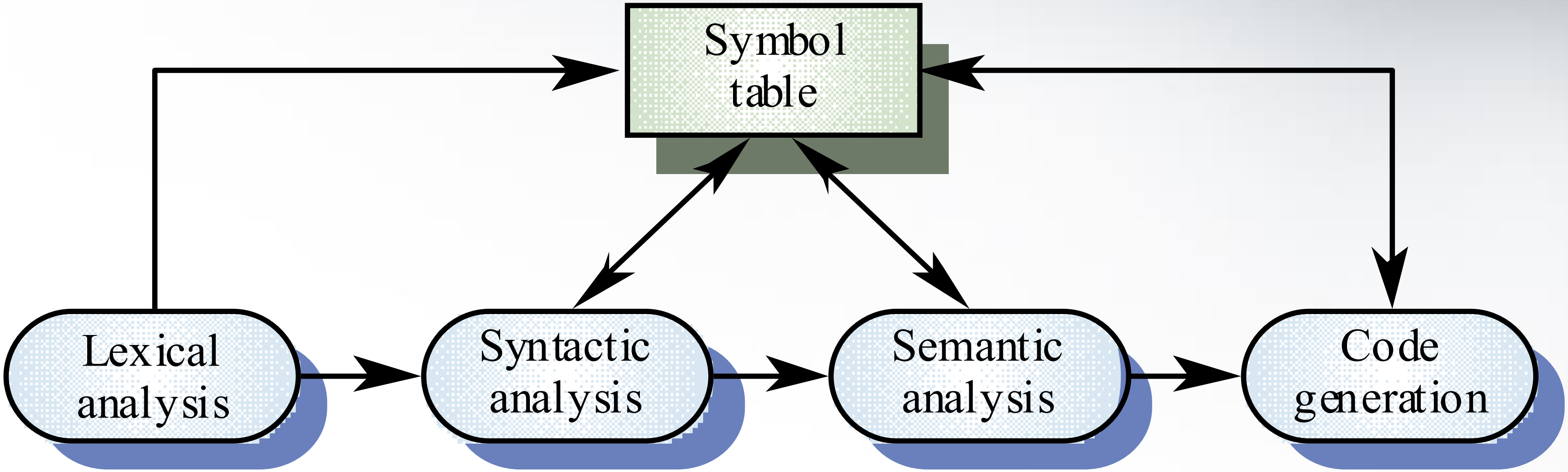
Selective Broadcasting Model



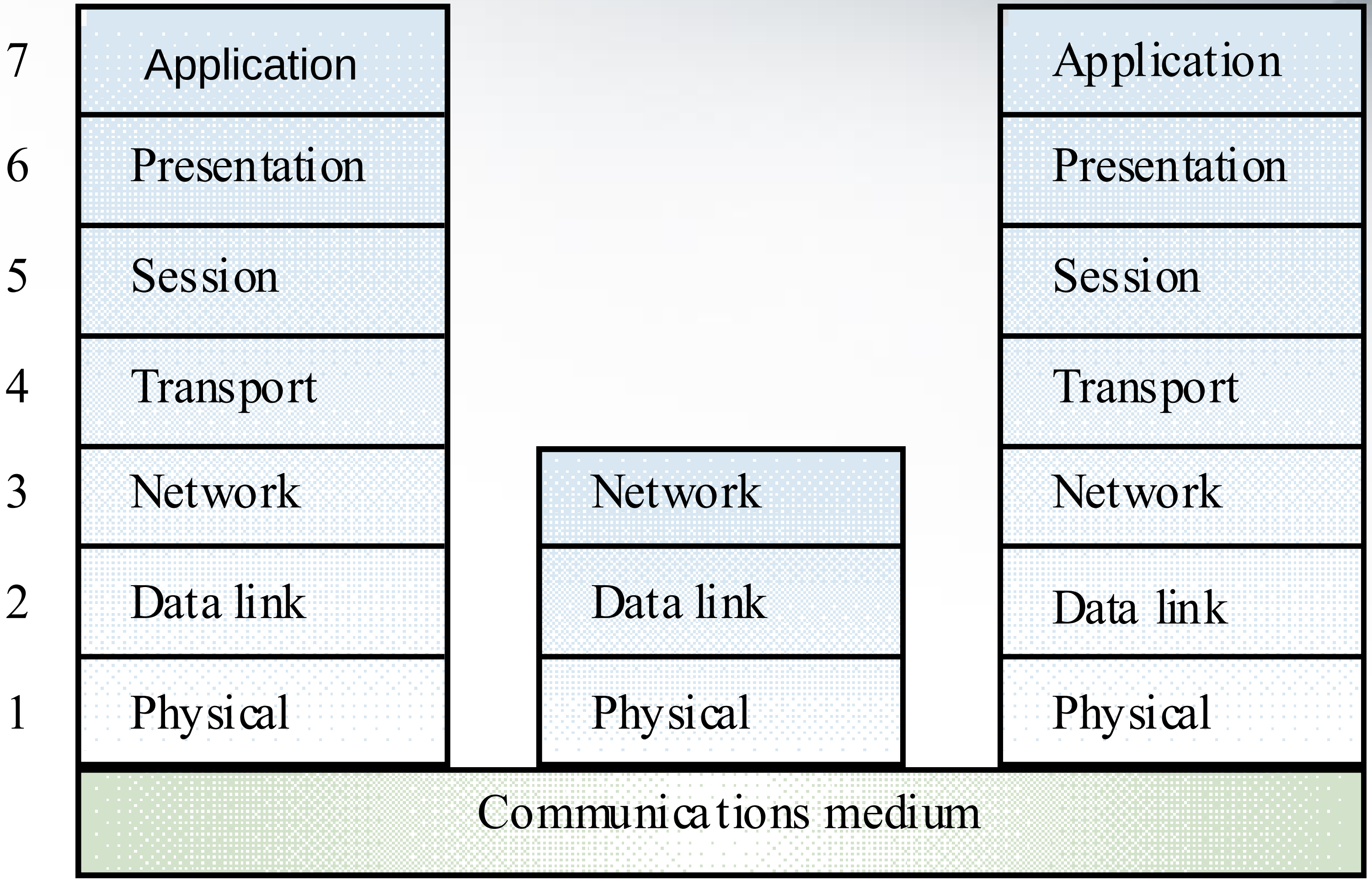
Interrupt-Driven Control Model



Compiler Model



OSI Reference Model



7	Application		Application
6	Presentation		Presentation
5	Session		Session
4	Transport		Transport
3	Network	Network	Network
2	Data link	Data link	Data link
1	Physical	Physical	Physical

Communications medium

Distributed Systems

Distributed Systems

- Most large computer systems are implemented as distributed systems
- Information is also distributed over several computers rather than being confined to a single machine
- Distributed software engineering has become very important

Distributed Systems Architectures

Distributed Systems Architectures

- Client/Server
  - offer distributed services which may be called by clients
  - servers providing services are treated differently than clients using the services
- Distributed Object
  - no distinctions made between clients and servers
  - any system object may provide and use services from any other system object

Middleware

Middleware

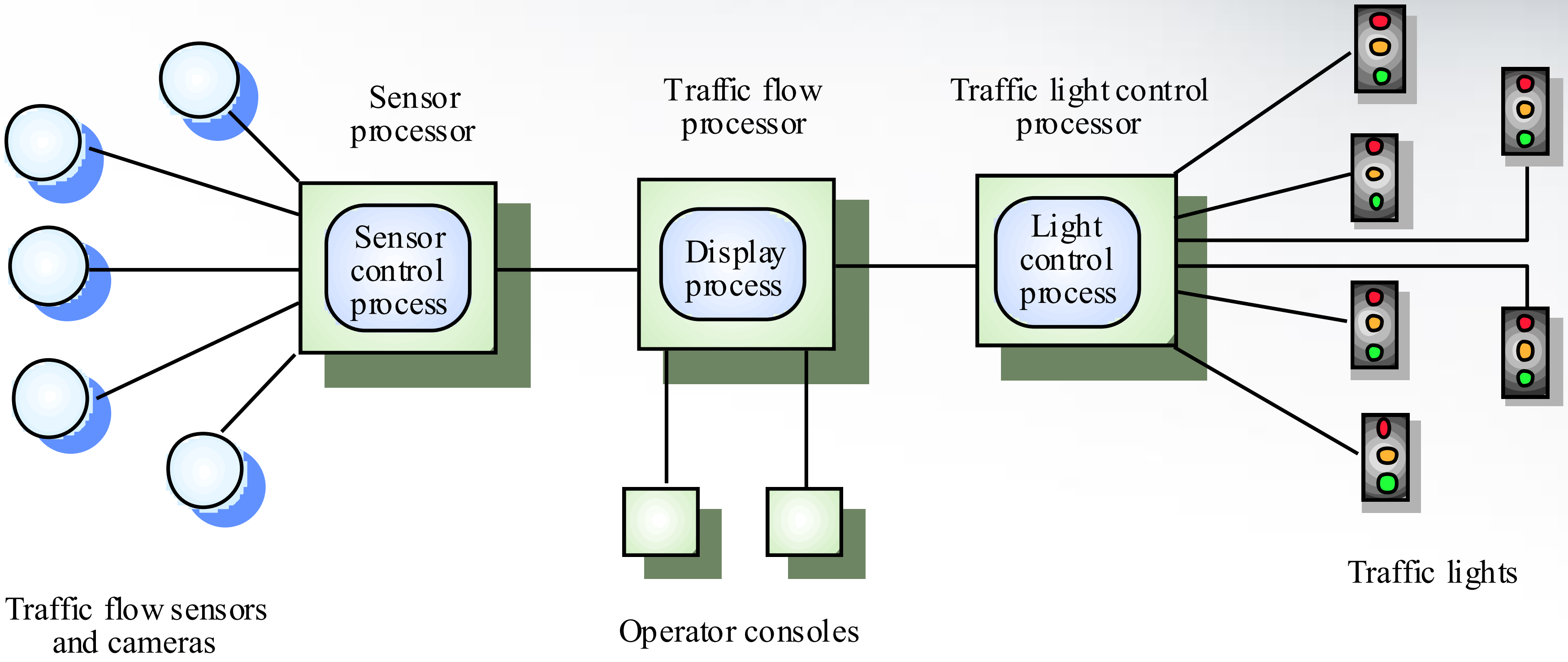
- Software that manages and supports the different components of a distributes system
- Sits in the middle of the system to broker service requests among components
- Usually off-the-shelf products rather than custom
- Representative architectures
  - CORBA (ORB)
  - COM (Microsoft)
  - JavaBeans (Sun)

Multiprocessor Architecture

Multiprocessor Architecture

- Simplest distributed system model
- System composed of multiple processes that may execute on different processors
- Model used in many large real-time systems
- Distribution of processes to processors may be preordered or may be under control of a dispatcher

Multiprocessor Traffic Control System

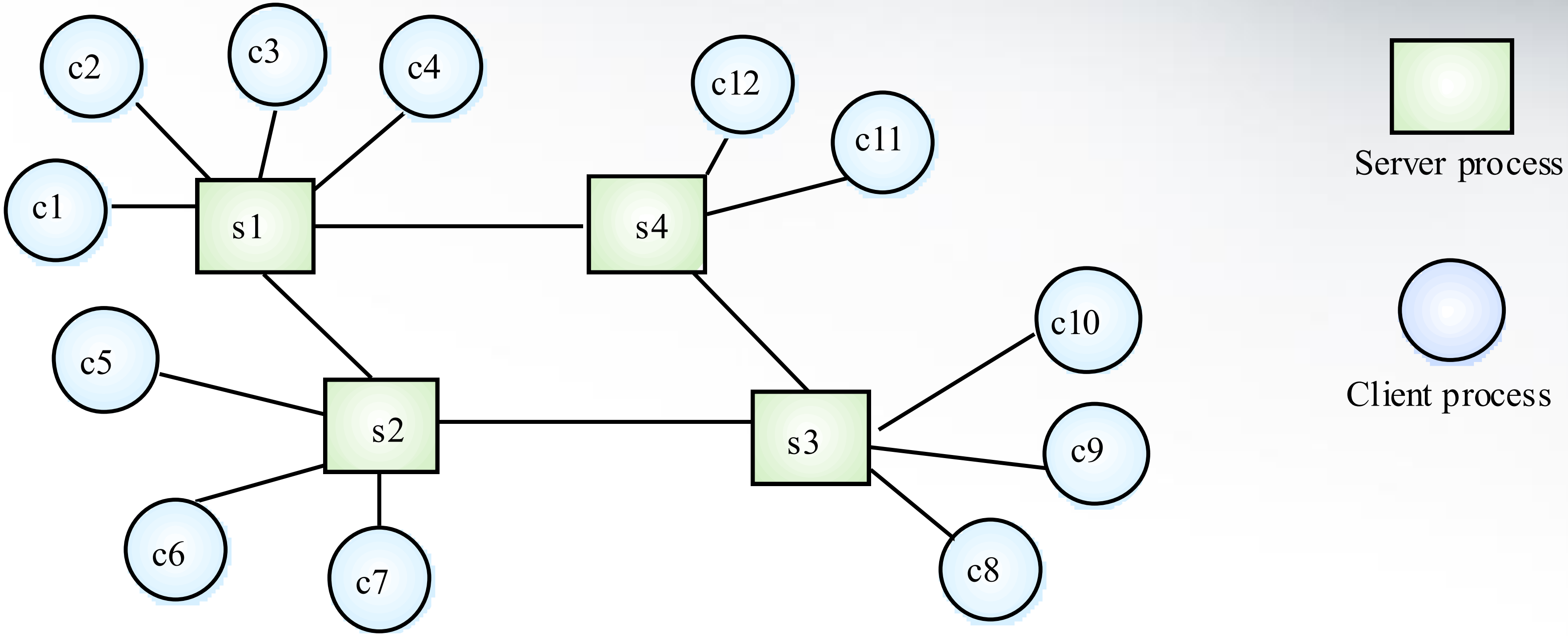


Client/Server Architectures

Client/Server Architectures

- Application is modeled as a set of services that are provided by servers and a set of clients that use these services
- Clients know the servers but the servers do not need to know all the clients
- Clients and servers are logical processes (not always physical machines)
- The mapping of processes to processors is not always 1:1

Client/Server System



Representative Client/Server Systems

- **File servers**
  - client requests selected records from a file
  - server transmits records to client over the network
- **Database servers**
  - client sends SQL requests to server
  - server processes the request
  - server returns the results to the client over the network

Representative Client/Server Systems

- **Transaction servers**
  - client sends requests that invokes remote procedures on the server side
  - server executes procedures invoked and returns the results to the client
- **Groupware servers**
  - server provides set of applications that enable communication among clients using text, images, bulletin boards, video, etc.

Client/Server Software Components

Client/Server Software Components

- **User interaction/presentation subsystem**
- **Application subsystem**
  - – implements requirements defined by the application within the context of the operating environment
  - – components may reside on either client or server side
- **Database management subsystem**
- **Middleware**
  - – all software components that exist on both the client and the server to allow exchange of information

Thin Client Model

Thin Client Model

- Used when legacy systems are migrated to client server architectures
  - the legacy system may act as a server in its own right
  - the GUI may be implemented on a client
- It chief disadvantage is that it places a heavy processing load on both the server and the network

Fat Client Model

Fat Client Model

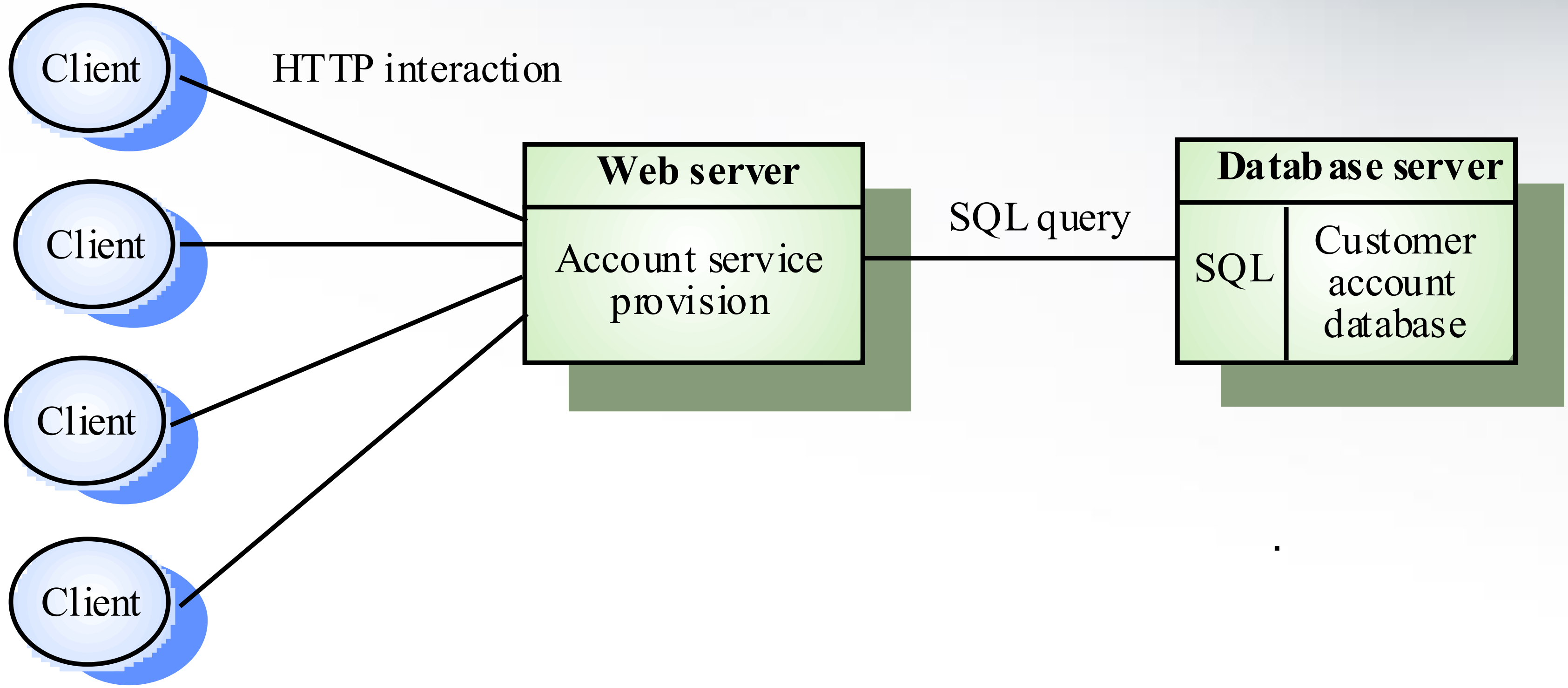
- More processing is delegated to the client as the application processing is locally extended
- Suitable for new client/server systems when the client system capabilities are known in advance
- More complex than thin client model with respect to management issues
- New versions of each application need to installed on every client

Three-tier Architecture

Three-tier Architecture

- Each application architecture layers (presentation, application, database) may run on separate processors
- Allows for better performance than a thin-client approach
- Simpler to manage than fat client approach
- Highly scalable (as demands increase add more servers)

Three-Tier Architecture



Design Issues for Client/Server Systems

Design Issues for Client/Server Systems

- **Data and architectural design**
  - dominates the design process to be able to effectively use the capabilities of RDBMS or OODMBS
- **Event-driven paradigm**
  - when used, behavioral modeling should be conducted
  - the control-oriented aspects of the behavioral model should translated into the design model

Design Issues for Client/Server Systems

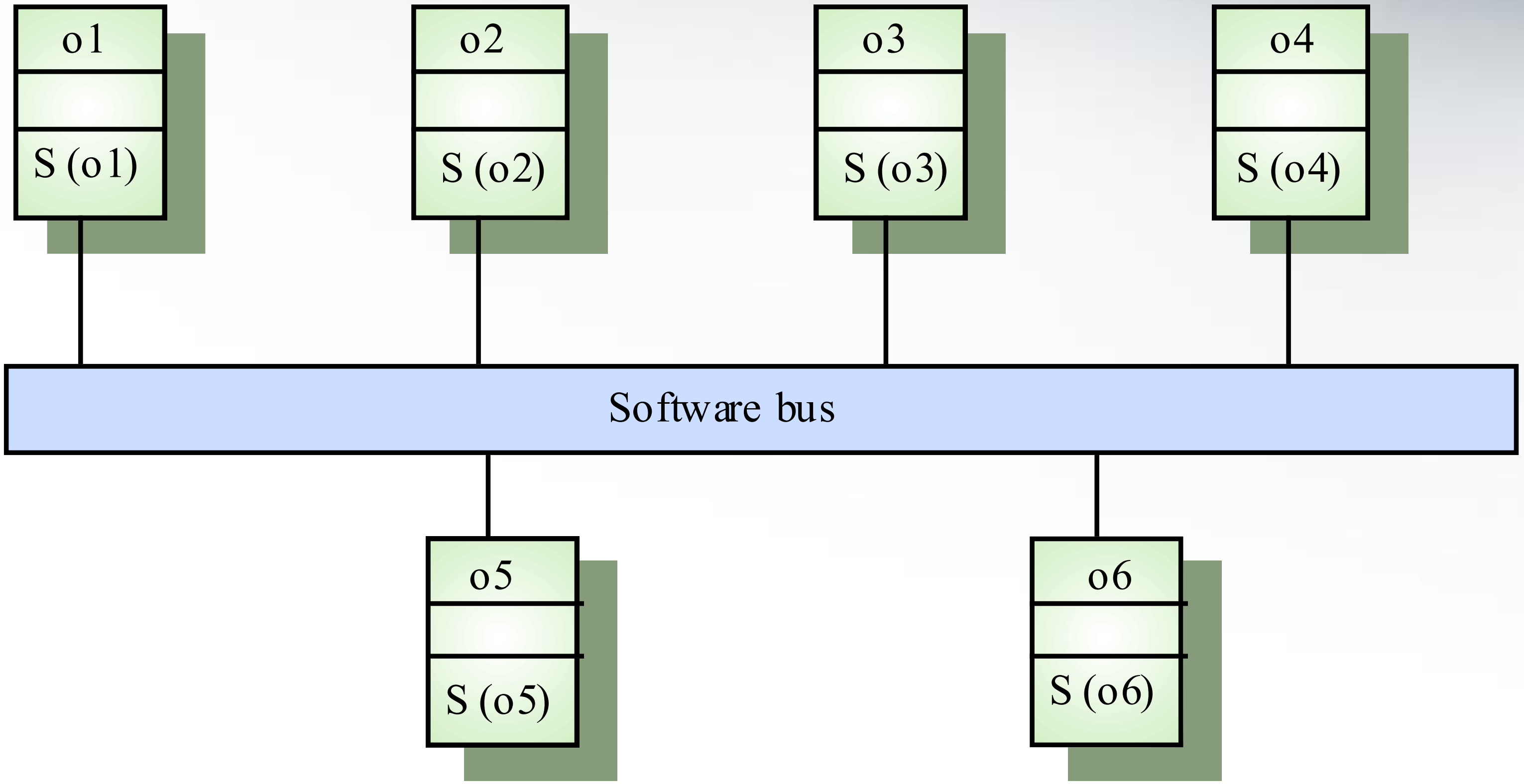
- **Interface design**
  - – elevated in importance
  - – user interaction/presentation component implements all functions associated with a GUI
- **Object-oriented point of view**
  - – often chosen, since object structure is provided by events initiated in the GUI and their event handlers within the client-based software

Distributed Object Architectures

Distributed Object Architectures

- No distinctions made between client objects and server objects
- Each distributable entity is an object that both provides and consumes services
- Object communication is though an object request broker (middleware or software bus)
- More complex to design than client/server systems

Distributed Object Architecture from Sommerville



Architectural Design Process

Architectural Design Process

- Basic Steps
  - Creation of the data design
  - Derivation of one or more representations of the architectural structure of the system
  - Analysis of alternative architectural styles to choose the one best suited to customer requirements and quality attributes
  - Elaboration of the architecture based on the selected architectural style
- A database designer creates the data architecture for a system to represent the data components
- A system architect selects an appropriate architectural style derived during system engineering and software requirements analysis

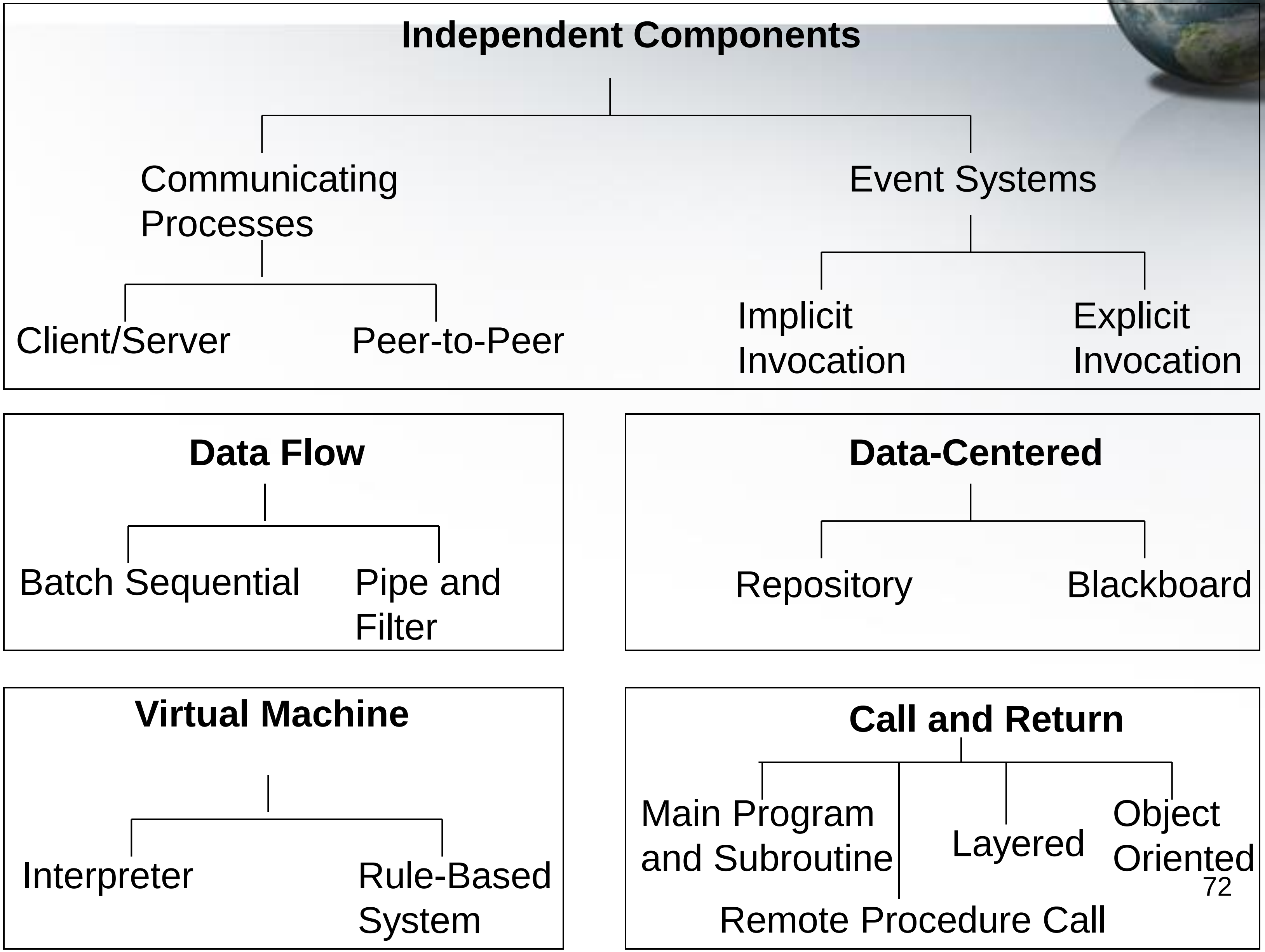
Emphasis on Software Components

- A software architecture enables a software engineer to
  - Analyze the effectiveness of the design in meeting its stated requirements
  - Consider architectural alternatives at a stage when making design changes is still relatively easy
  - Reduce the risks associated with the construction of the software
- Focus is placed on the software component
  - A program module
  - An object-oriented class
  - A database
  - Middleware

Software Architectural Style

- The software that is built for computer-based systems exhibit one of many architectural styles
- Each style describes a system category that encompasses
  - A set of component types that perform a function required by the system
  - A set of connectors (subroutine call, remote procedure call, data stream, socket) that enable communication, coordination, and cooperation among components
  - Semantic constraints that define how components can be integrated to form the system
  - A topological layout of the components indicating their runtime interrelationships

A Taxonomy of Architectural Styles



Independent Components

- Communicating Processes
  - Client/Server
  - Peer-to-Peer
- Event Systems
  - Implicit Invocation
  - Explicit Invocation

Data Flow

- Batch Sequential
- Pipe and Filter

Data-Centered

- Repository
- Blackboard

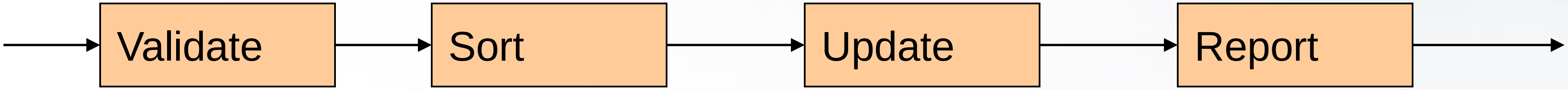
Virtual Machine

- Interpreter
- Rule-Based System

Call and Return

- Main Program and Subroutine
- Remote Procedure Call
- Layered
- Object Oriented

Data Flow Style



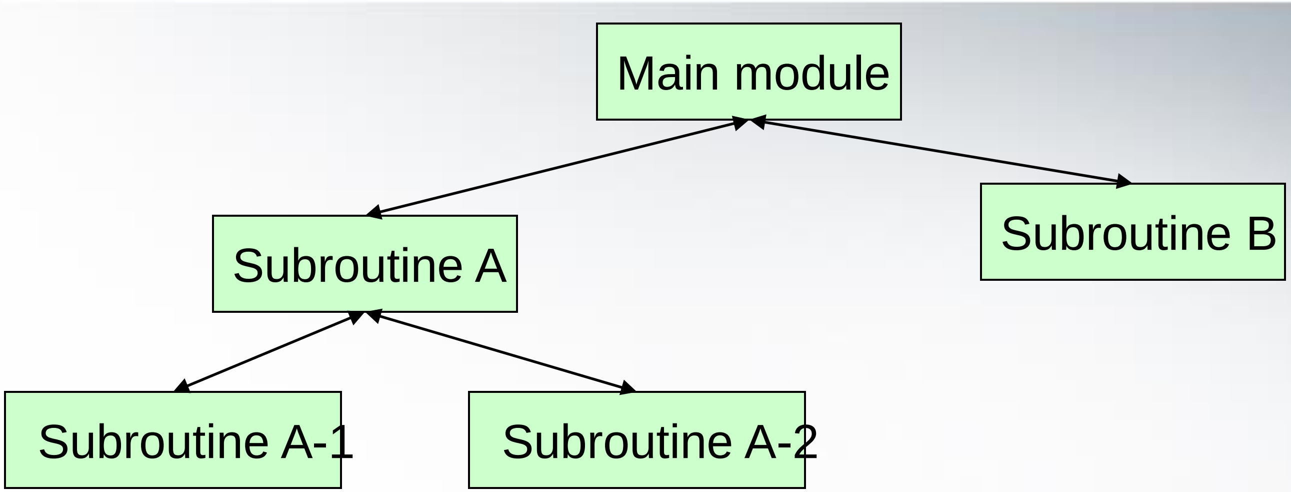
Data Flow Style

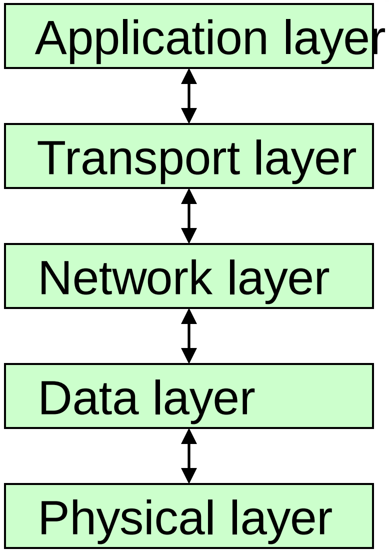
- **Batch sequential style**
  - The processing steps are independent components
  - Each step runs to completion before the next step begins
- **Pipe-and-filter style**
  - Emphasizes the incremental transformation of data by successive components
  - The filters incrementally transform the data (entering and exiting via streams)
  - The filters use little contextual information and retain no state between instantiations
  - The pipes are stateless and simply exist to move data between filters

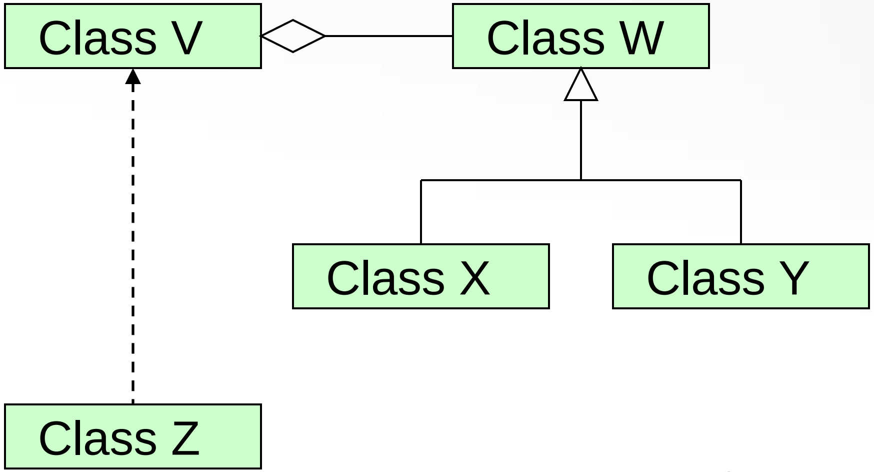
Data Flow Style (continued)

- Use this style when it makes sense to view your system as one that produces a well-defined easily identified output
  - The output should be a direct result of sequentially transforming a well-defined easily identified input in a time-independent fashion

Call-and-Return Style







Call-and-Return Style

- Has the goal of modifiability and scalability
- Has been the dominant architecture since the start of software development
- Main program and subroutine style
  - Decomposes a program hierarchically into small pieces (i.e., modules)
  - Typically has a single thread of control that travels through various components in the hierarchy

Remote procedure call style

- Consists of main program and subroutine style of system that is decomposed into parts that are resident on computers connected via a network
- Strives to increase performance by distributing the computations and taking advantage of multiple processors
- Incurs a finite communication time between subroutine call and response

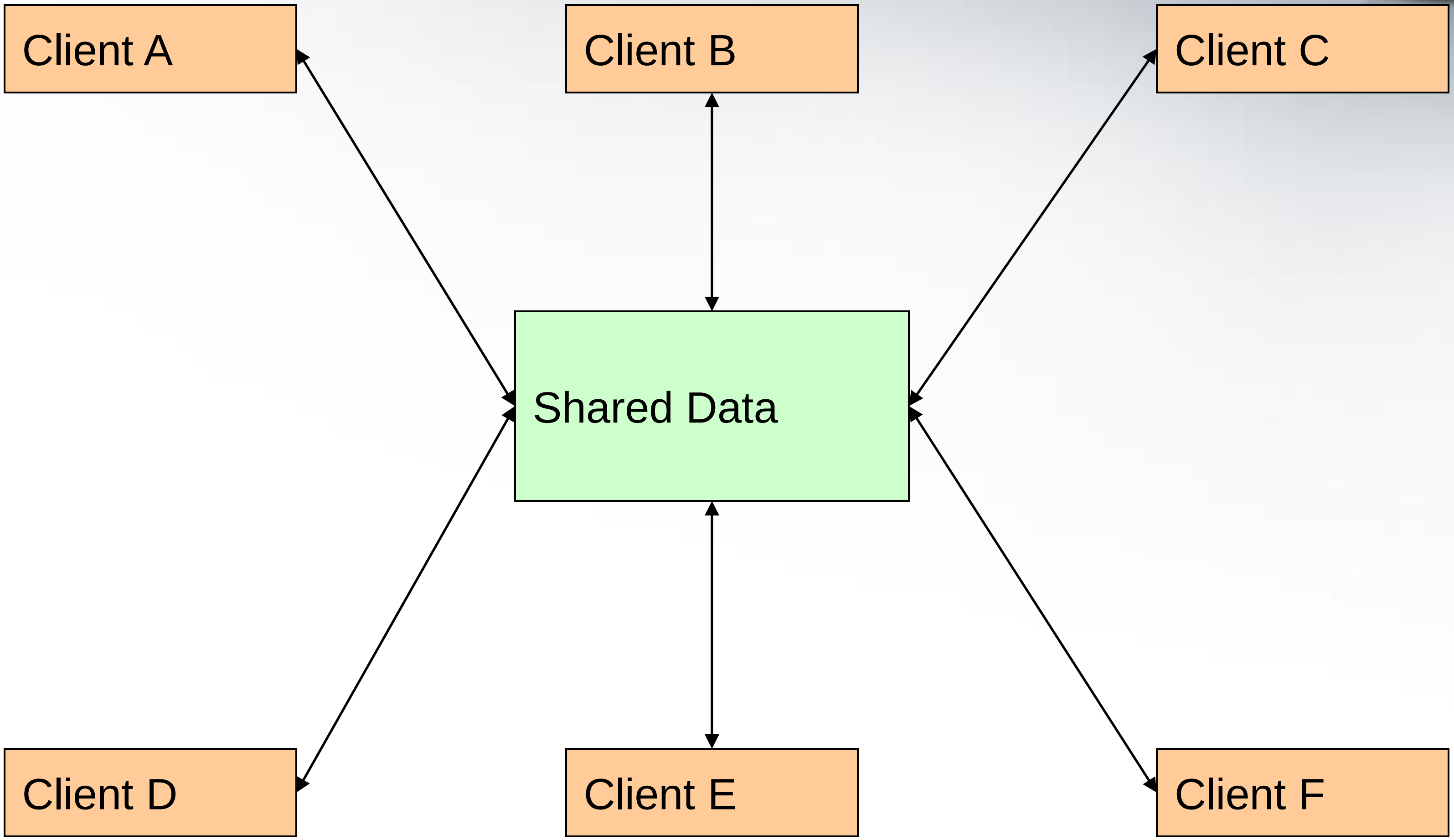
Call-and-Return Style

- Object-oriented or abstract data type system
  - Emphasizes the bundling of data and how to manipulate and access data
  - Keeps the internal data representation hidden and allows access to the object only through provided operations
  - Permits inheritance and polymorphism
- Use this style when the order of computation is fixed, when interfaces are specific, and when components can make no useful progress while awaiting the results of request to other components

Call-and-Return Style
Layered system

- – Assigns components to layers in order to control inter-component interaction
- – Only allows a layer to communicate with its immediate neighbor
- – Assigns core functionality such as hardware interfacing or system kernel operations to the lowest layer
- – Builds each successive layer on its predecessor, hiding the lower layer and providing services for the upper layer
- – Is compromised by layer bridging that skips one or more layers to improve runtime performance

Data-Centered Style



Client A

Client B

Client C

Shared Data

Client D

Client E

Client F

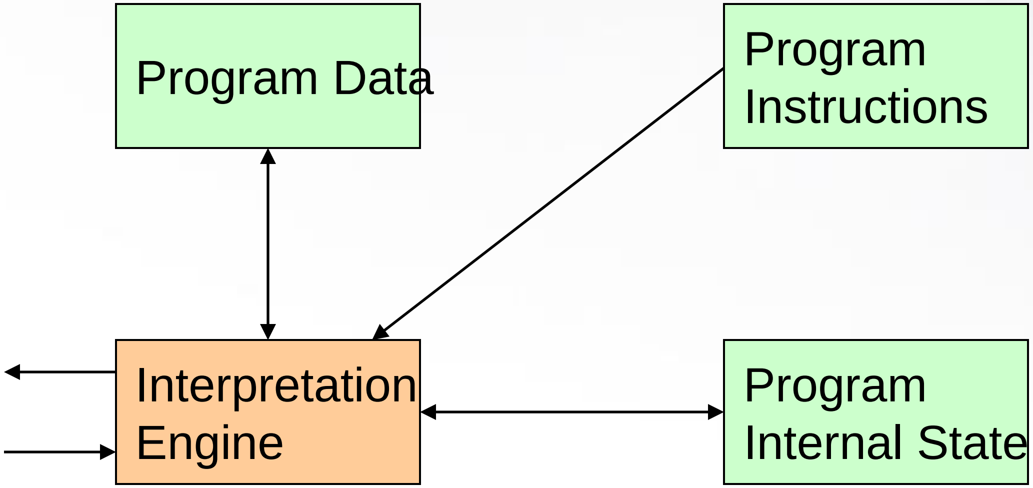
Data-Centered Style (continued)

- Has the goal of integrating the data
- Refers to systems in which the access and update of a widely accessed data store occur
- A client runs on an independent thread of control
- The shared data may be a passive repository or an active blackboard
  - A blackboard notifies subscriber clients when changes occur in data of interest
- At its heart is a centralized data store that communicates with a number of clients
- Clients are relatively independent of each other so they can be added, removed, or changed in functionality
- The data store is independent of the clients

Data-Centered Style

- Use this style when a central issue is the storage, representation, management, and retrieval of a large amount of related persistent data
- Note that this style becomes client/server if the clients are modeled as independent processes

Virtual Machine Style



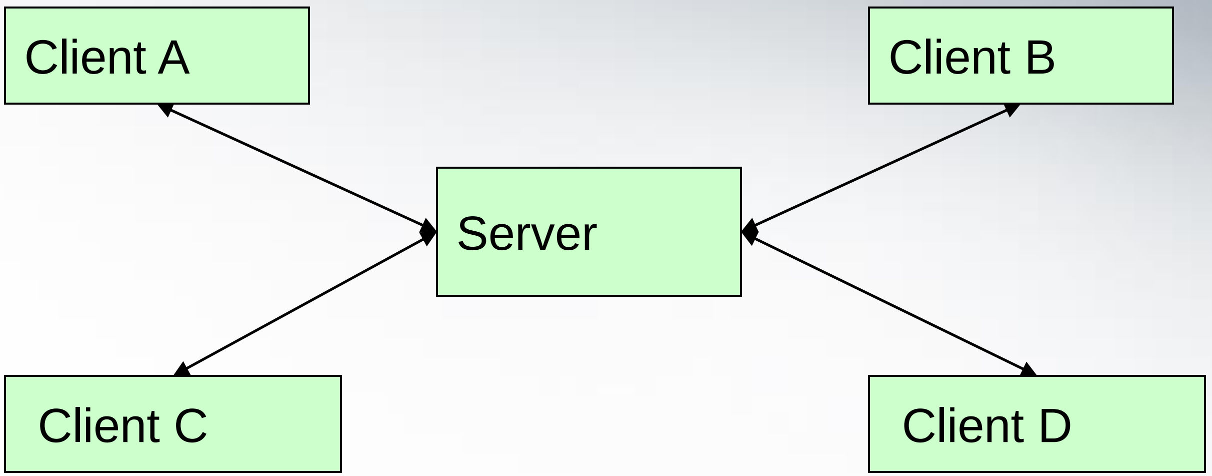
Virtual Machine Style

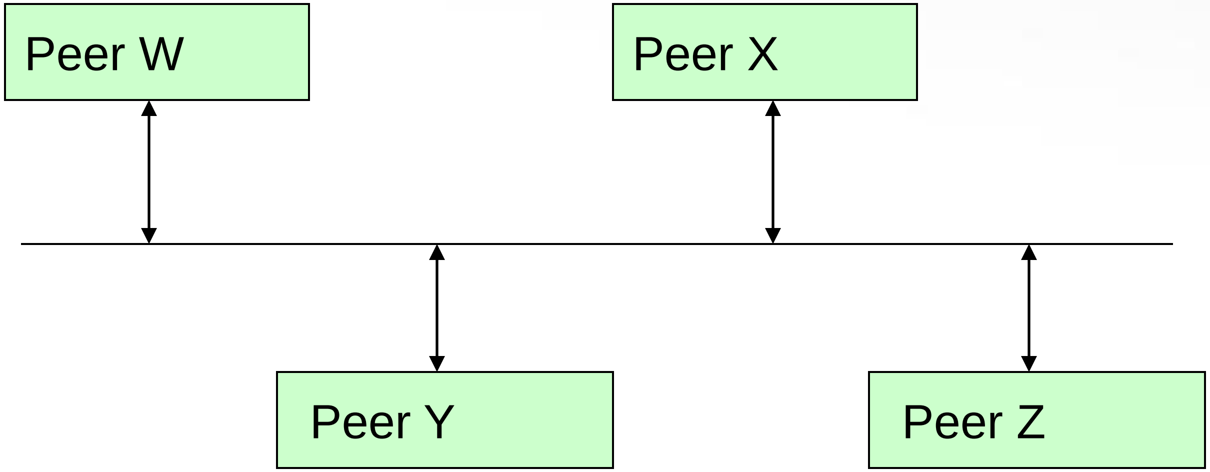
- Has the goal of portability
- Software systems in this style simulate some functionality that is not native to the hardware and/or software on which it is implemented
  - Can simulate and test hardware platforms that have not yet been built
  - Can simulate "disaster modes" as in flight simulators or safety-critical systems that would be too complex, costly, or dangerous to test with the real system

Virtual Machine Style

- Examples include interpreters, rule-based systems, and command language processors
- Interpreters
  - Add **flexibility** through the ability to interrupt and query the program and introduce modifications at runtime
  - Incur a **performance cost** because of the additional computation involved in execution
- Use this style when you have developed a program or some form of computation but have **no make of machine** to directly run it on

Independent Component Style





Independent Component Style

- Consists of a number of independent processes that communicate through messages
- Has the goal of modifiability by decoupling various portions of the computation
- Sends data between processes but the processes do not directly control each other

Independent Component Style

- Event systems style
  - Individual components announce data that they wish to share (publish) with their environment
  - The other components may register an interest in this class of data (subscribe)
  - Makes use of a message component that manages communication among the other components
  - Components publish information by sending it to the message manager
  - When the data appears, the subscriber is invoked and receives the data
  - Decouples component implementation from knowing the names and locations of other components

Independent Component Style

- Communicating processes style
  - These are classic multi-processing systems
  - Well-know subtypes are client/server and peer-to-peer
  - The goal is to achieve scalability
  - A server exists to provide data and/or services to one or more clients
  - The client originates a call to the server which services the request

Independent Component Style

- Use this style when
  - Your system has a graphical user interface
  - Your system runs on a multiprocessor platform
  - Your system can be structured as a set of loosely coupled components
  - Performance tuning by reallocating work among processes is important
  - Message passing is sufficient as an interaction mechanism among components

Heterogeneous Styles

- Systems are seldom built from a single architectural style
- Three kinds of heterogeneity
  - Locationally heterogeneous
    - The drawing of the architecture reveals different styles in different areas (e.g., a branch of a call-and-return system may have a shared repository)
  - Hierarchically heterogeneous
    - A component of one style, when decomposed, is structured according to the rules of a different style
  - Simultaneously heterogeneous
    - Two or more architectural styles may both be appropriate descriptions for the style used by a computer-based system

Types of Architecture

Types of Architecture

- Business Architecture
- Technical Architecture
- Solutions Architecture
- Enterprise Architecture
- Product Line Architecture

Business Architecture

- Concerned with the business model as it relates to an automated solution.
  - E-business is a good candidate
  - Structural part of requirements analysis.
  - Domain Specific

Technical Architecture

- Specific to technology and the use of this technology to structure the technical points (Technology Mapping) of an architecture
  - .NET
  - J2EE
  - Hardware architects

Solutions Architecture

- Specific to a particular business area (or project) but still reliant on being a technical focal point for communications between the domain architect, business interests and development.

Enterprise Architecture

- The organizing logic for a firm’s core business processes and IT capabilities captured in a **set of principles**, **policies** and **technical choices** to achieve the business standardization and integration requirements of the firm’s operating model.
- Concerned with cross project/solution architecture and communication between different practices in architecture.

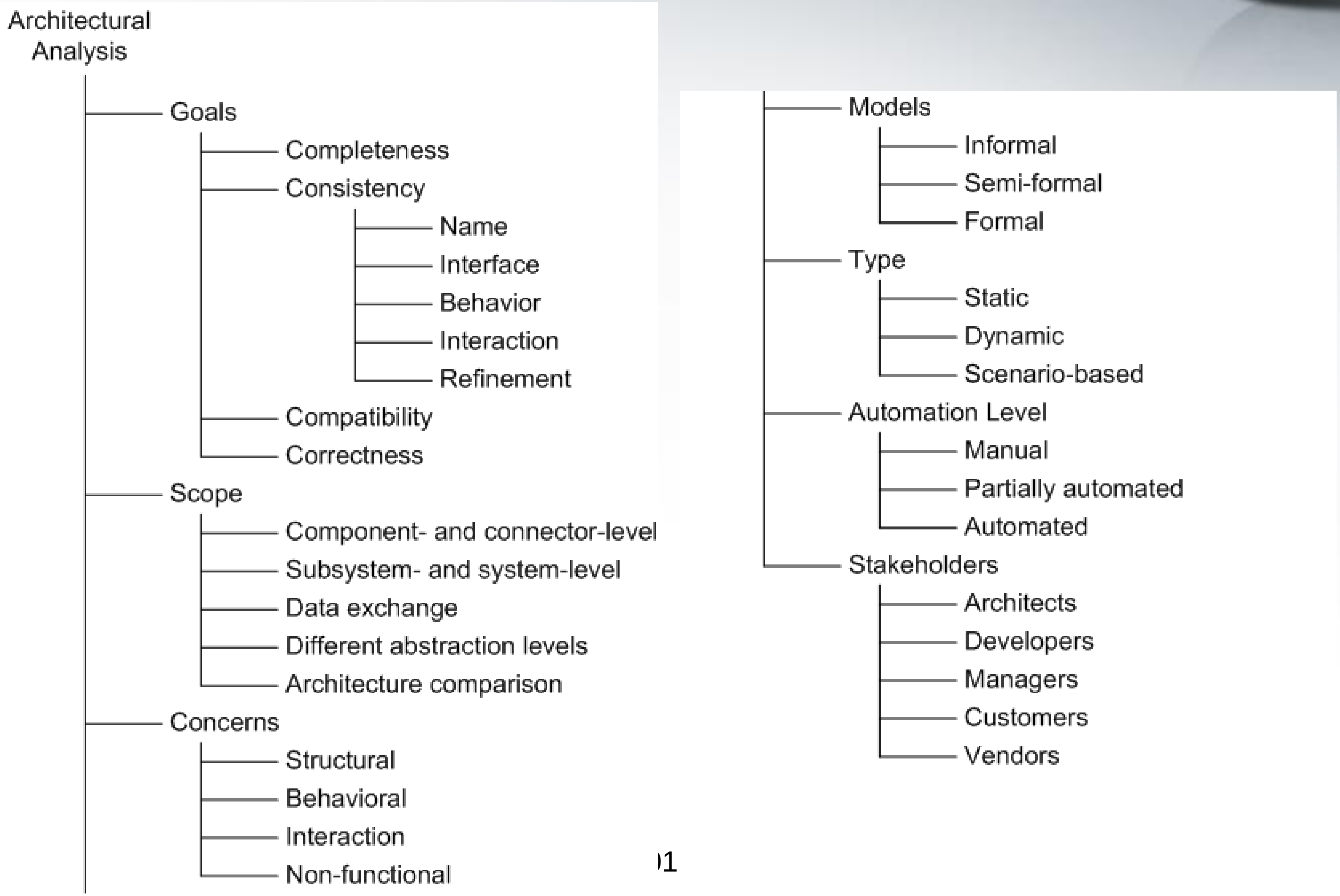
Product Line Architecture

- Common Architecture for a set of products or systems developed by an organization

Product Line - Initiation

- Evolutionary
  - – Product line architecture and components evolve with the requirements posed by new product line members.
- Revolutionary
  - Product line architecture and components developed to match requirements of all expected product-line members

Architectural Analysis in a Nutshell



Architectural Analysis

- Goals
  - Completeness
  - Consistency
    - Name
    - Interface
    - Behavior
    - Interaction
    - Refinement
  - Compatibility
  - Correctness
- Scope
  - Component- and connector-level
  - Subsystem- and system-level
  - Data exchange
  - Different abstraction levels
  - Architecture comparison
- Concerns
  - Structural
  - Behavioral
  - Interaction
  - Non-functional
- Models
  - Informal
  - Semi-formal
  - Formal
- Type
  - Static
  - Dynamic
  - Scenario-based
- Automation Level
  - Manual
  - Partially automated
  - Automated
- Stakeholders
  - Architects
  - Developers
  - Managers
  - Customers
  - Vendors

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Why do we need models?

To create a simplified representation of a real software system

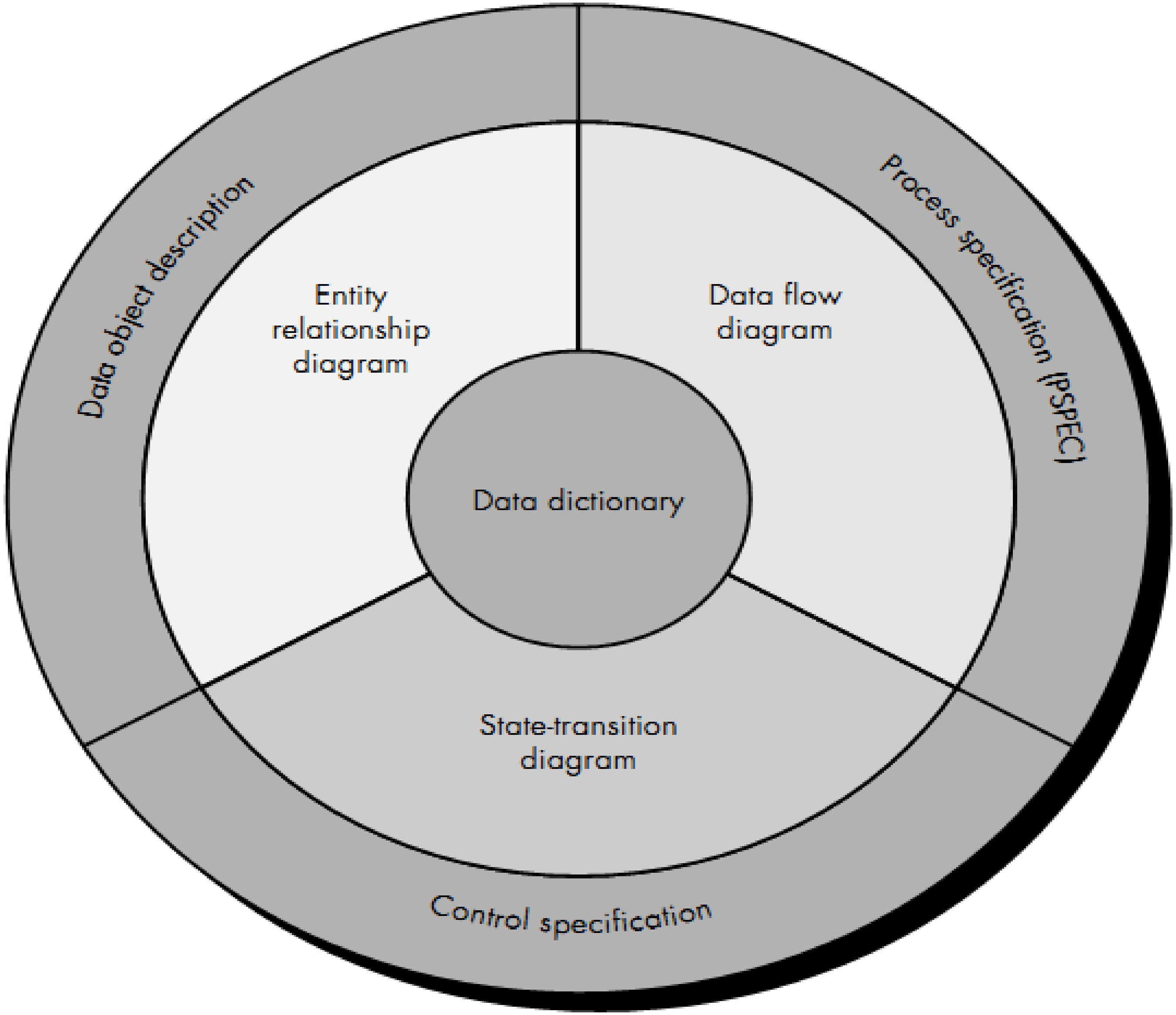
Modeling

- Modeling is a way of thinking about the problems and it organizes the real world ideas.

- A modeling method comprises **a language** and also **a procedure** for using the language to construct models.

- Modeling is the only way to **visualize your design** and check it against requirements before your crew starts to code.

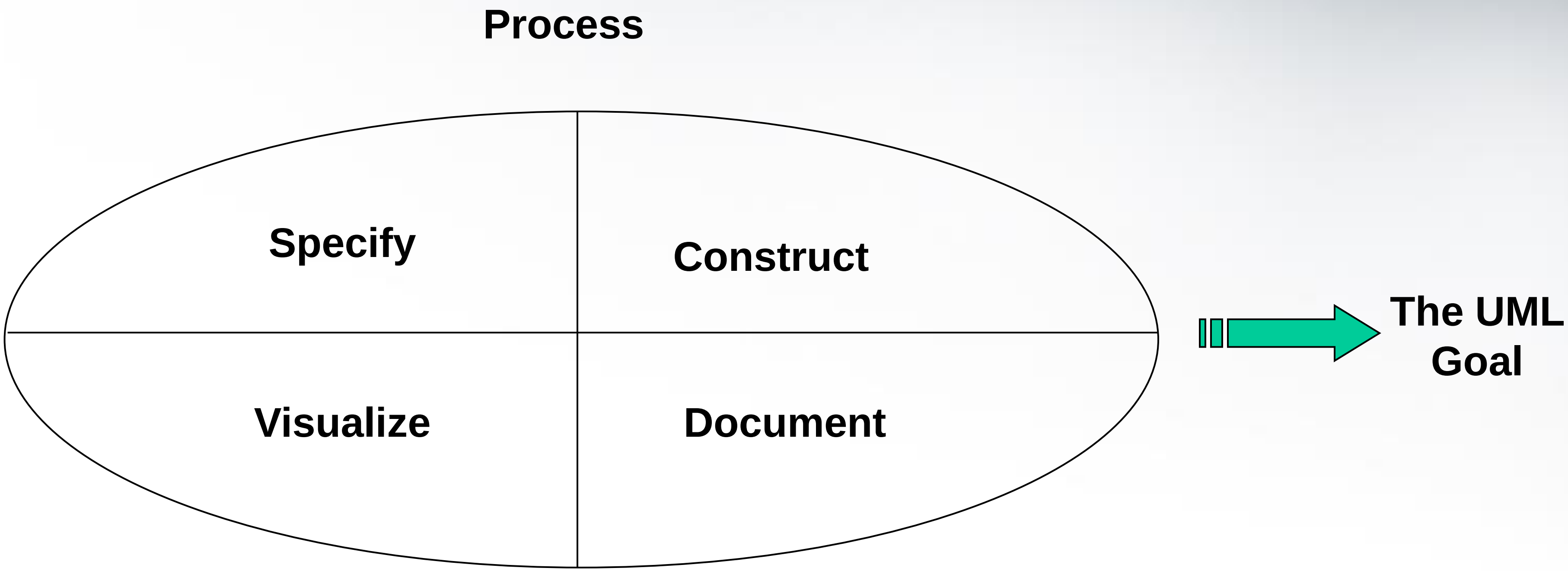
The structure of the analysis model



What is UML?

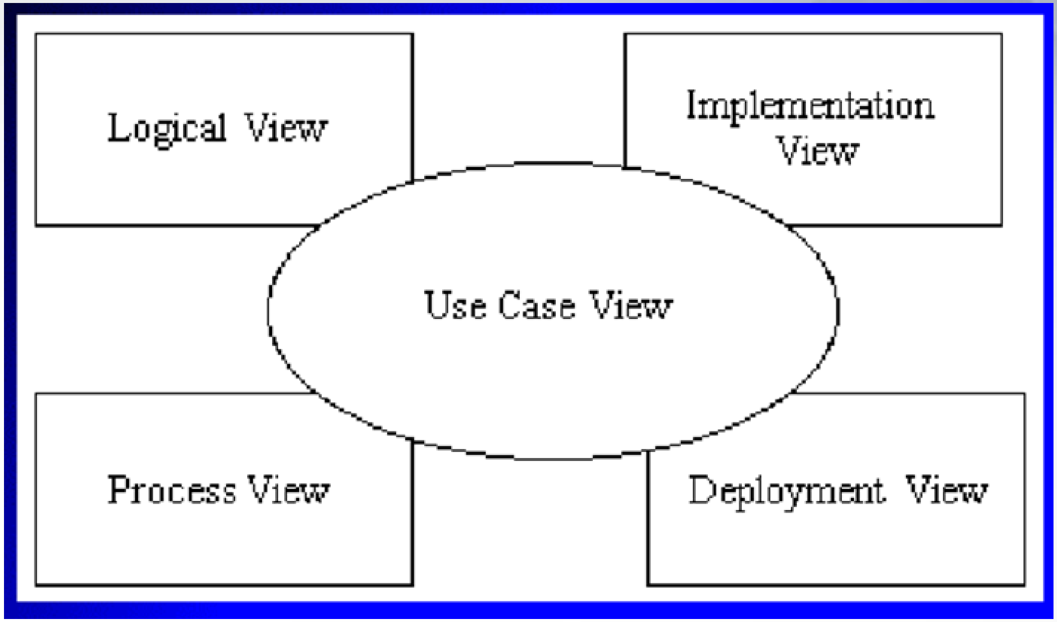
- A popular, standardized modeling language for object-oriented software.
- The Object Management Group (OMG) is the group that dictates the UML standard.
- Created by Grady Booch, James Rumbaugh, and Ivar Jacobson.
- It is used for both database and software modeling.

The Unified Modeling Language



UML Concepts-The 4+1 view

- Use Case view
  - Understandability
- Logical View
  - Functionality
- Process View
  - Performance
  - Scalable
  - Throughput
- Implementation View
  - Software management
- Deployment View
  - System topology
  - Delivery
  - Installation

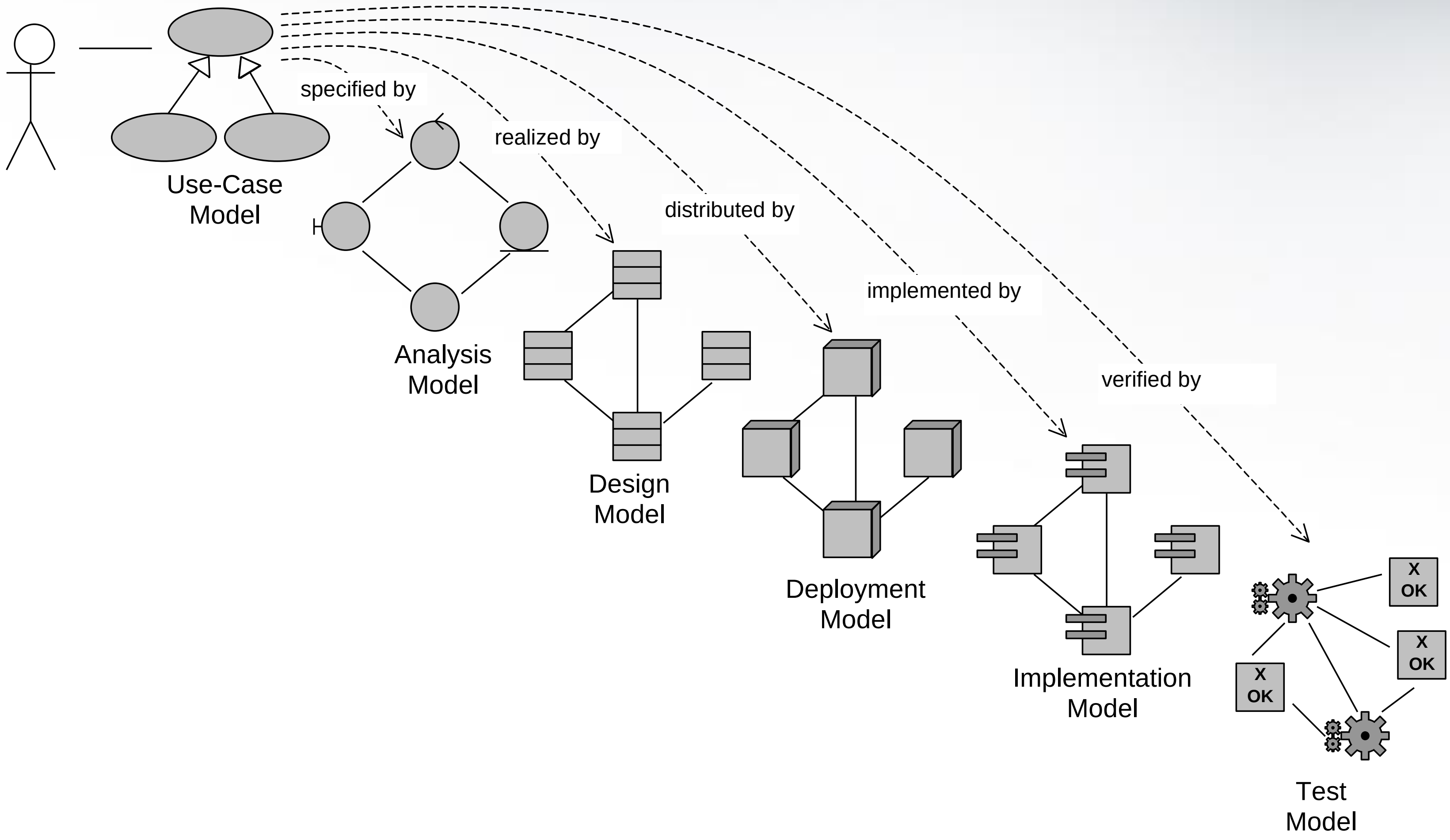


UML Models

- Functional Models
  - Use case diagrams
- Structural Models
  - Class diagrams
- Dynamic/Behavioral Models
  - Statechart diagrams
  - Sequence diagrams
  - Activity diagrams

Unified Process

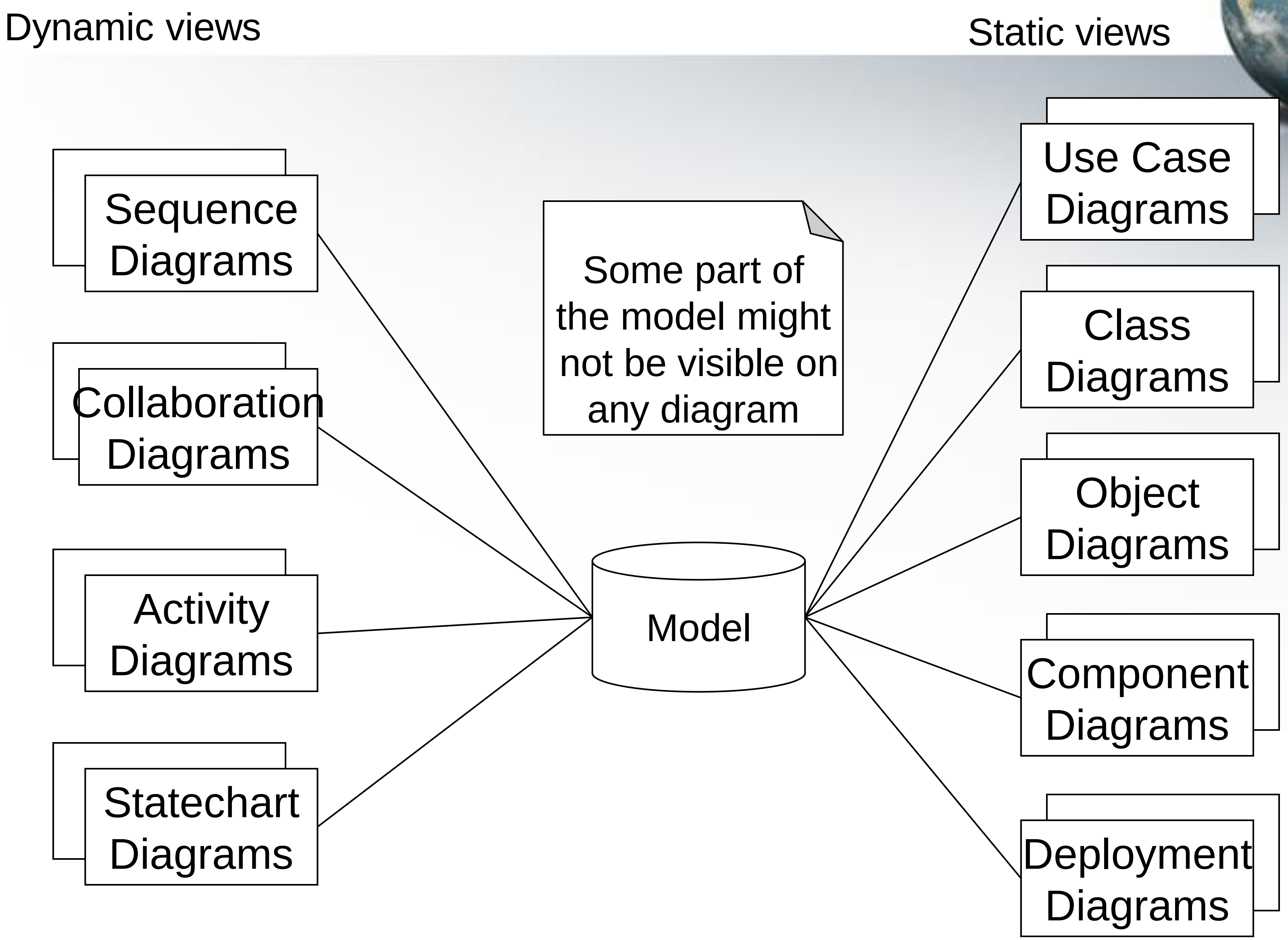
The ***use case model*** is supposed to define the requirements posed on the system and is intended to drive the whole development process.



Diagrams in UML

- A ***Diagram*** is the graphical presentation of a set of elements, most often rendered as a connected graph of things and relationships. UML includes 9 such diagrams.
- **1. Class Diagram.**
- **2. Object Diagram.**
- **3. Use Case Diagram.**
- **4. Sequence Diagram.**
- **5. Collaboration Diagram.**
- **6. State Chart Diagram.**
- **7. Activity Diagram.**
- **9. Deployment Diagram.**

There are 9 types of diagrams in UML



Dynamic views

Static views

Sequence Diagrams

Collaboration Diagrams

Activity Diagrams

Statechart Diagrams

Some part of the model might not be visible on any diagram

Model

Use Case Diagrams

Class Diagrams

Object Diagrams

Component Diagrams

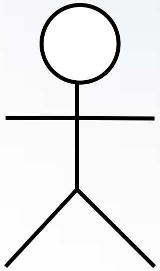
Deployment Diagrams

USE CASE DIAGRAMS

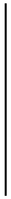
Use case diagrams

- Use Case Diagrams describe the functionality of a system and users of the system. These diagrams contain the following elements:
- **Actors**, which represent users of a system, including human users and other systems.
- **Use Cases**, which represent functionality or services provided by a system to users.

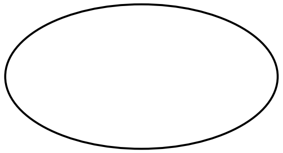
Use case diagram symbols



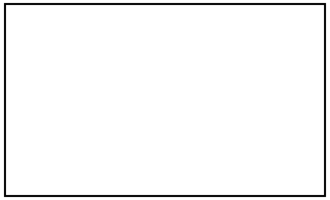
Actor



Relationships



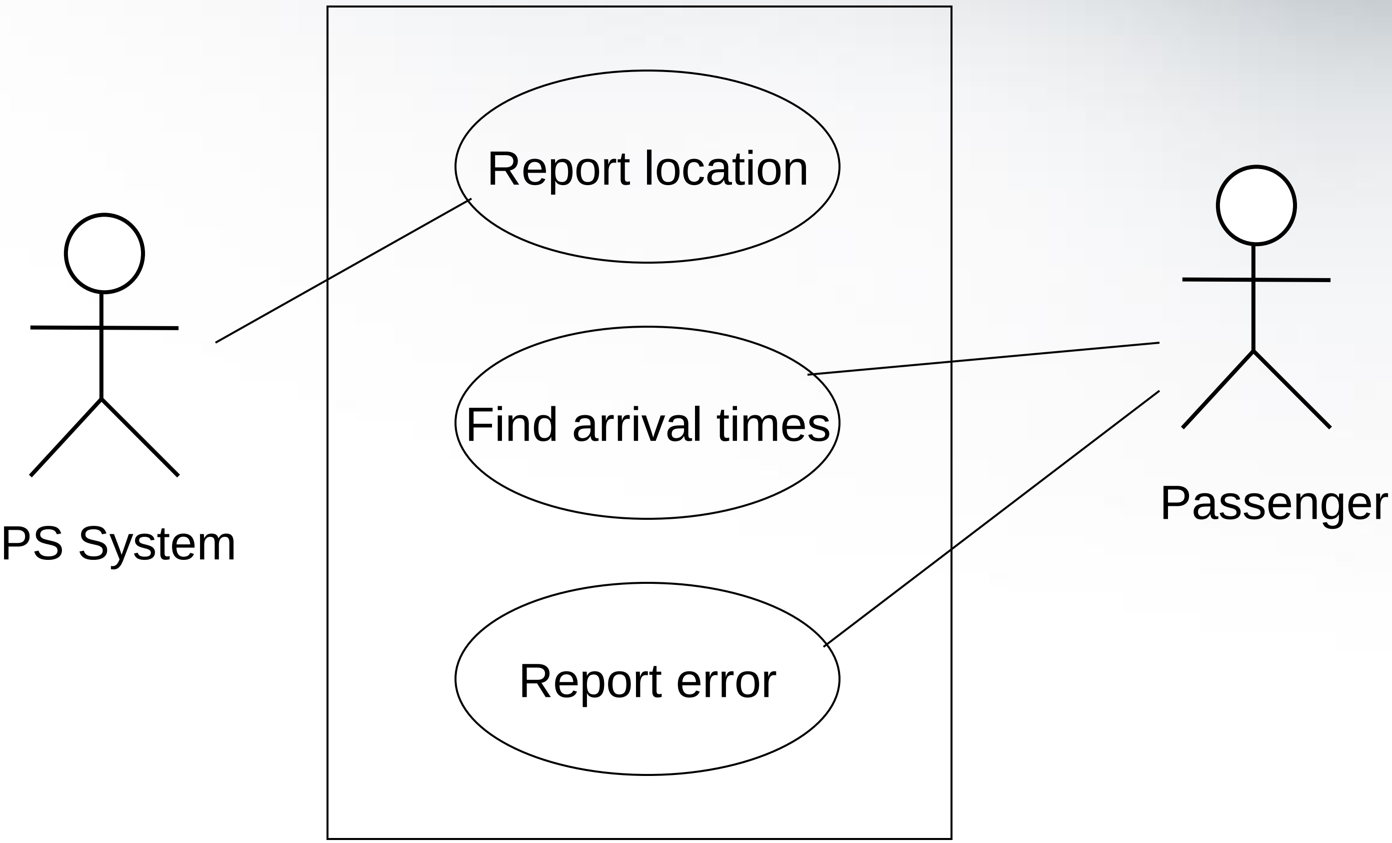
Use case



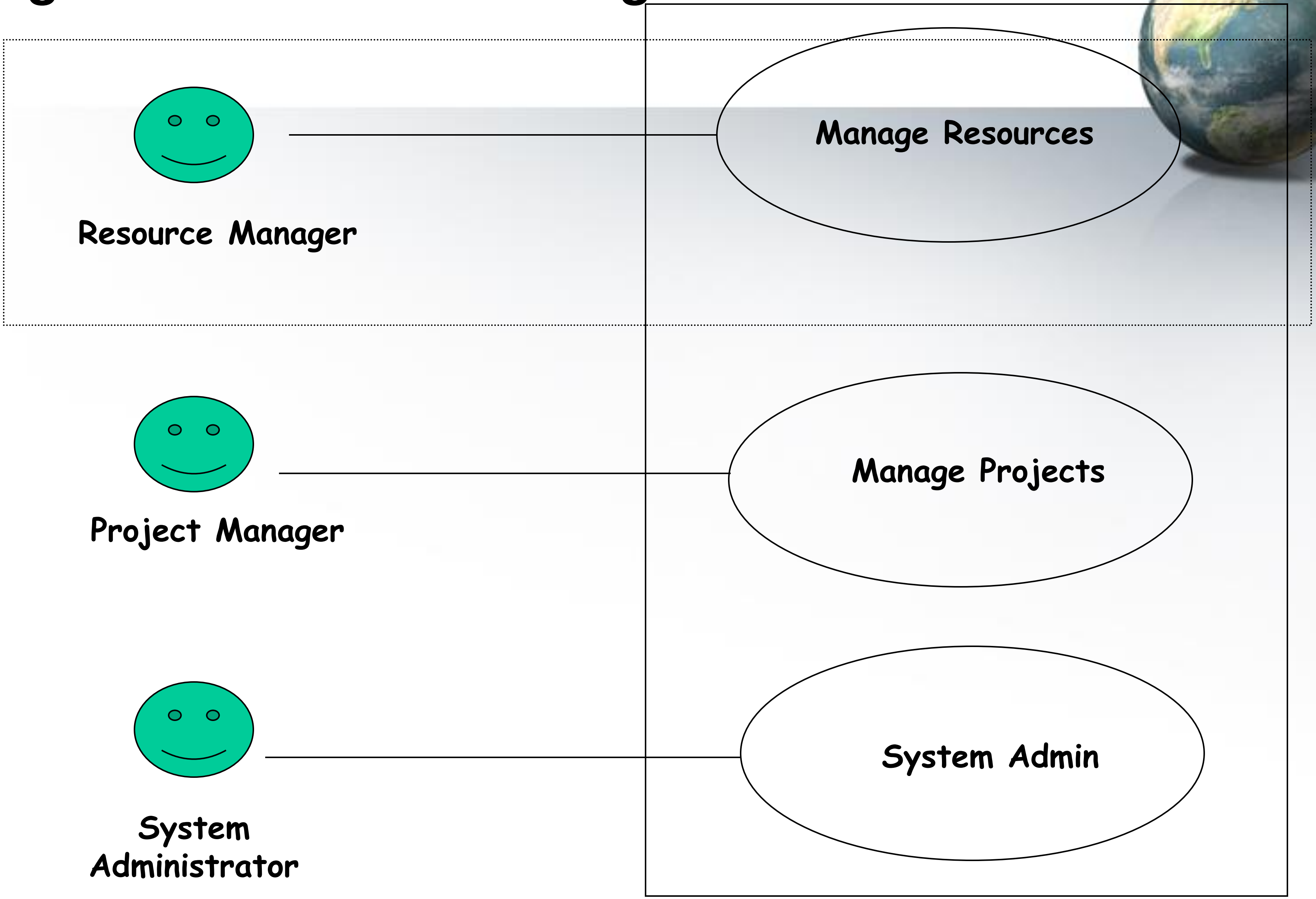
System

Use case example

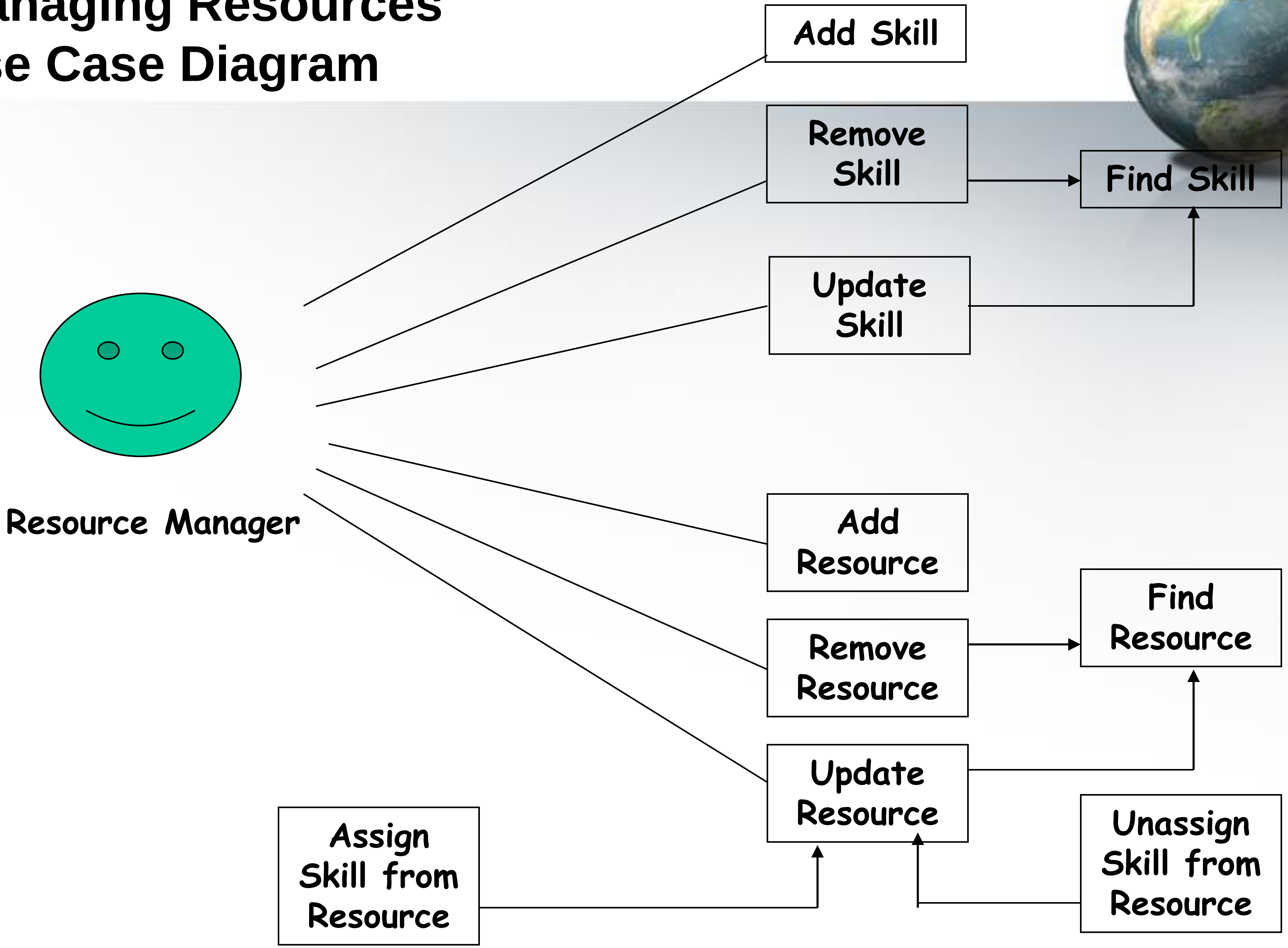
Actors don’t have to be people
Shows how actors interact with the system



High Level Use Case Diagram



Managing Resources
Use Case Diagram



CLASS DIAGRAMS

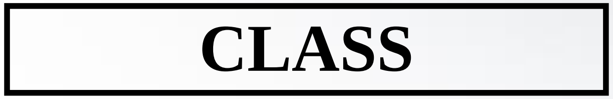
Class Diagrams

- Class Diagrams describe the static structure of a system, or how it is structured rather than how it behaves. These diagrams contain the following elements.

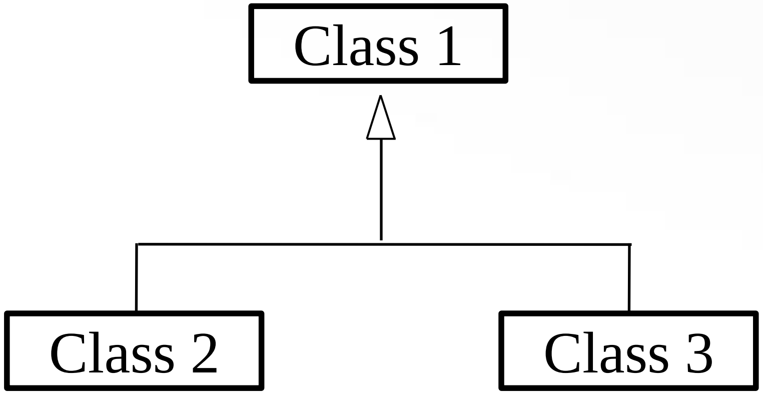
- **Classes**, which represent entities with common characteristics or features. These features include **attributes, operations** and **associations.**

- **Associations**, which represent relationships that relate two or more other classes where the relationships have common characteristics or features.

UML Class Diagram Symbols



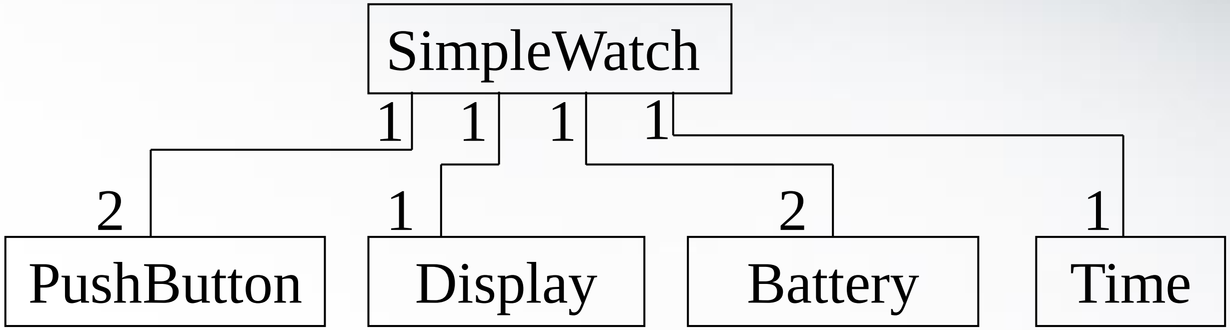
Class Name



Inheritance relationship

Class 2 and Class 3 derive from Class 1

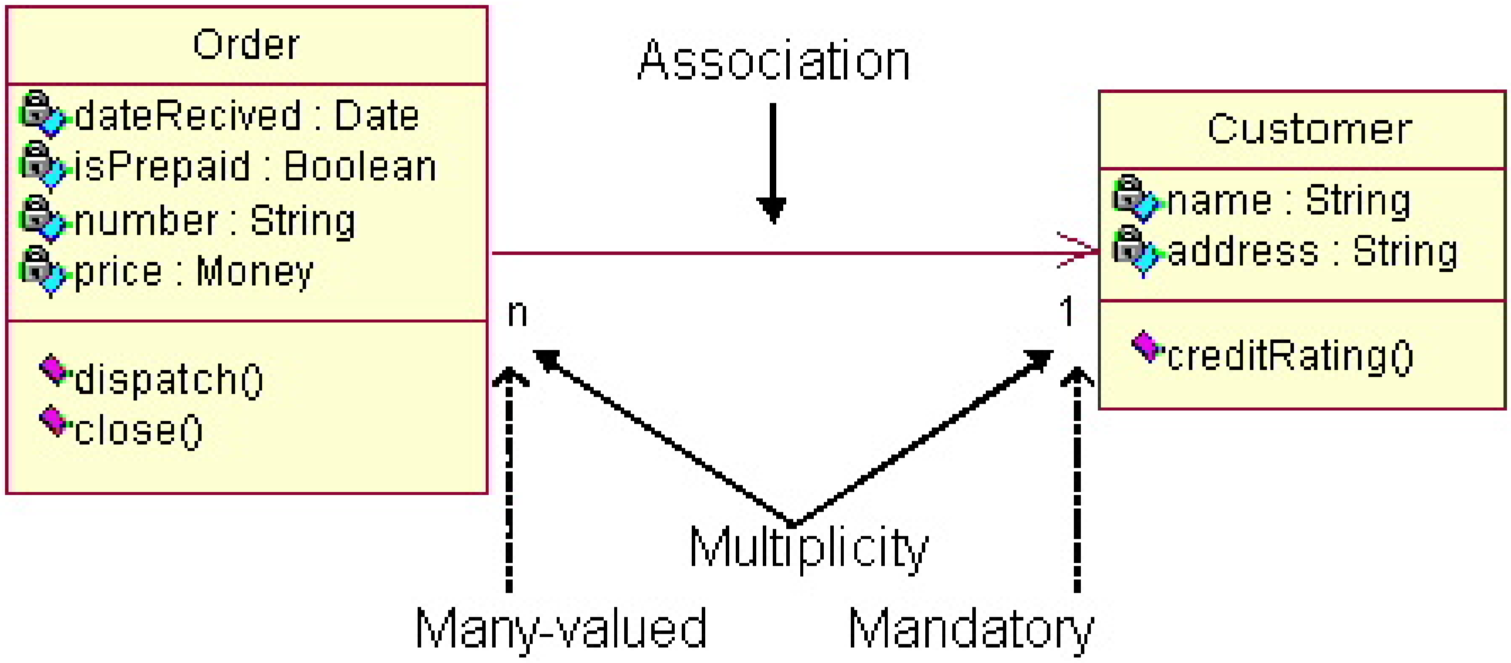
UML Class Diagram Example



SimpleWatch class contains

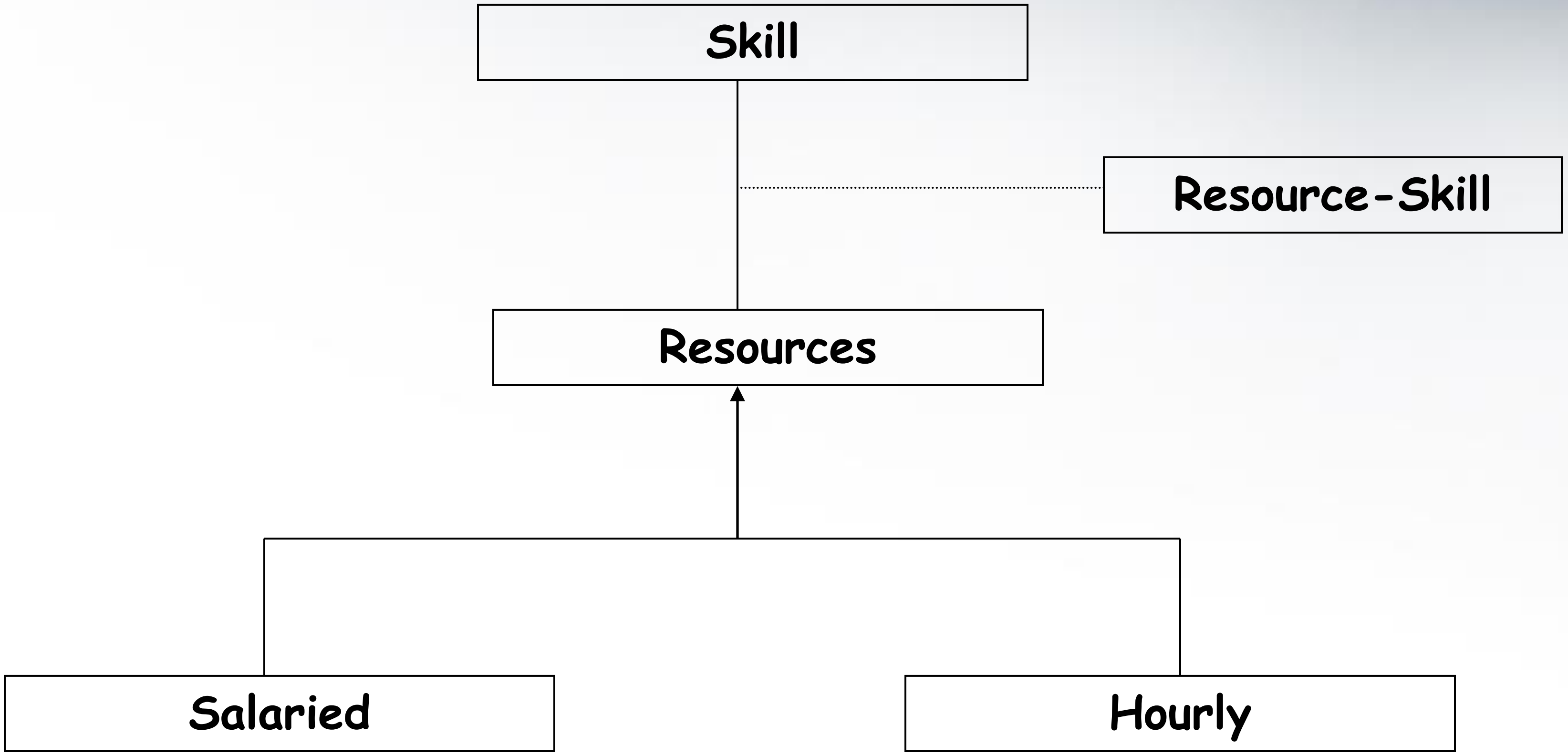
- 2 PushButtons
- 1 Display
- 2 Batteries
- 1 Time

UML Class Diagram Example

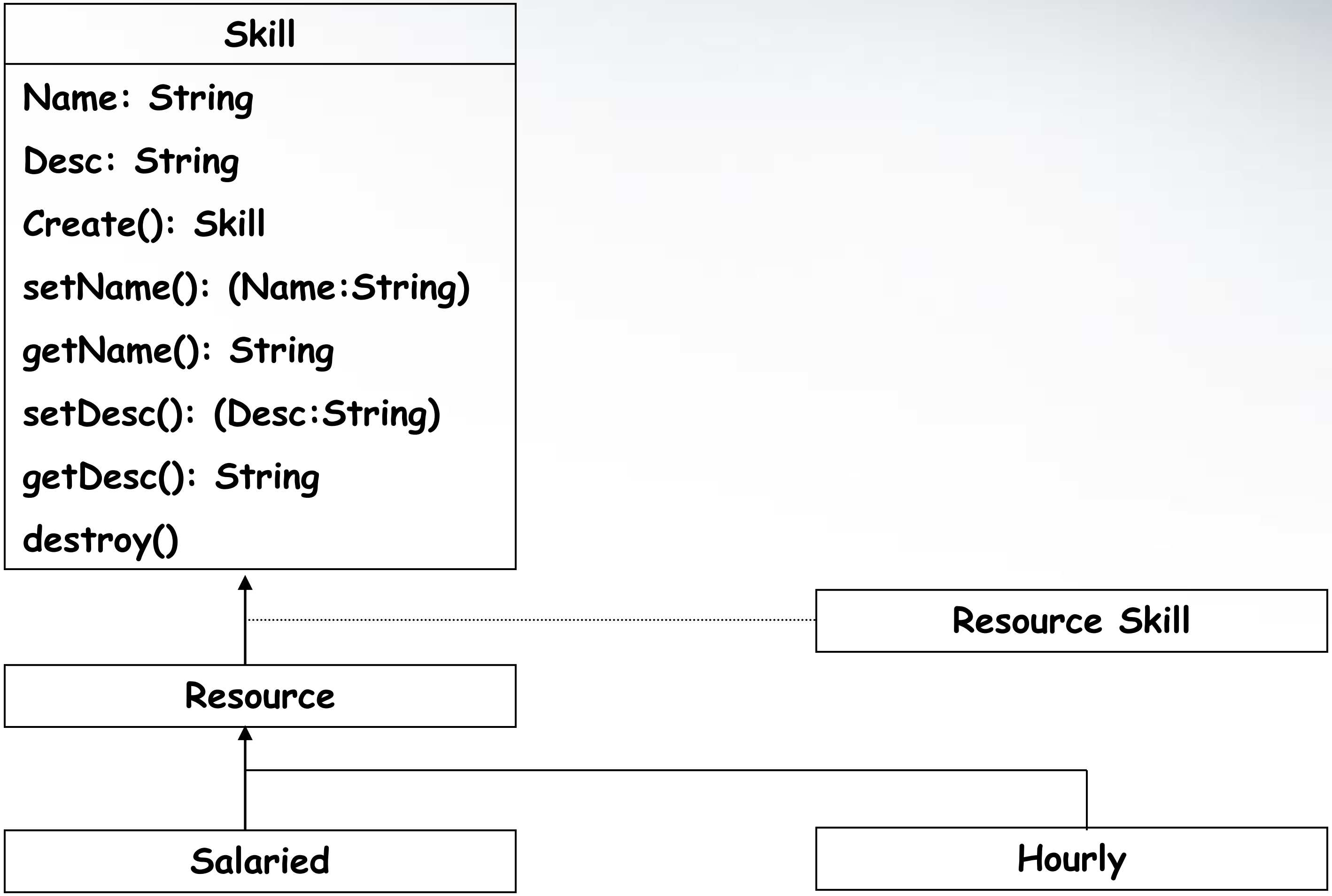


The class Order is associated with the class Customer
The multiplicity denotes the number of objects
For example, Order object can be associated only one customer but a customer can be associated to many orders

High-Level Resource Class Diagram



Detailed Resource Class Diagram



Skill

Name: String

Desc: String

Create(): Skill

setName(): (Name:String)

getName(): String

setDesc(): (Desc:String)

getDesc(): String

destroy()

Resource Skill

Resource

Salaried

Hourly

Relationships

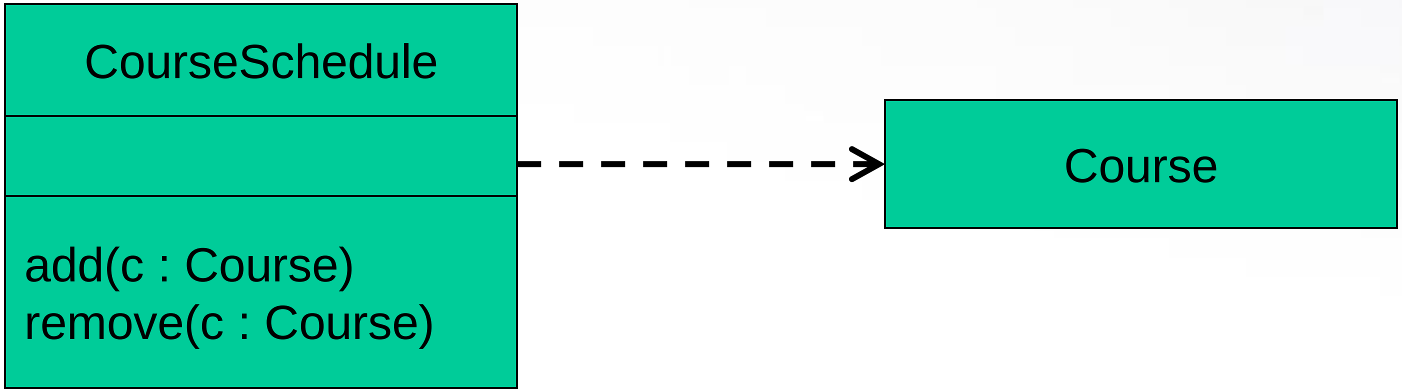
In UML, object interconnections (logical or physical), are modeled as relationships.

There are three kinds of relationships in UML:

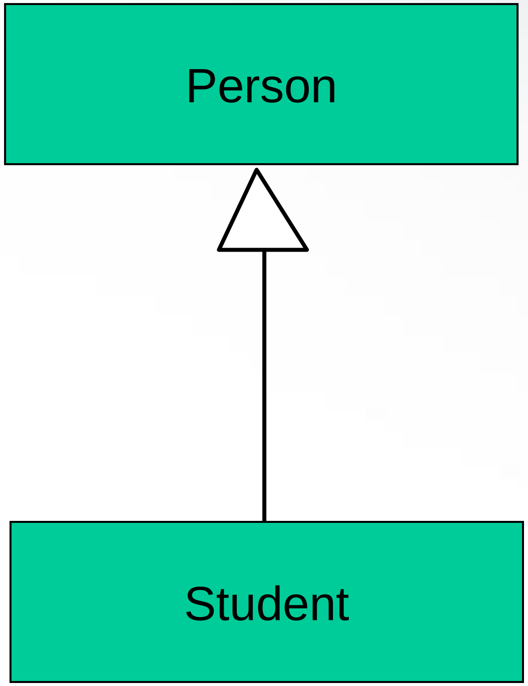
- dependencies
- generalizations
- associations

Dependency Relationships

A *dependency* indicates a semantic relationship between two or more elements. The dependency from *CourseSchedule* to *Course* exists because *Course* is used in both the **add** and **remove** operations of *CourseSchedule*.



Generalization Relationships

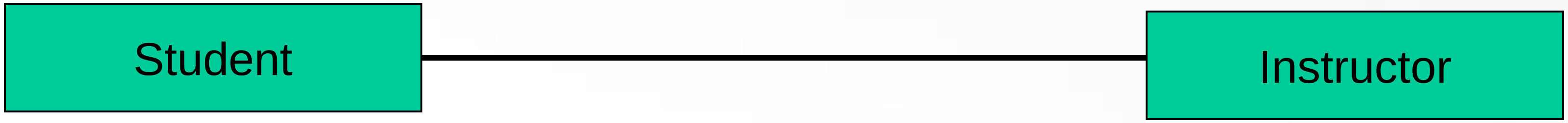


A *generalization* connects a subclass to its superclass. It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.

Association Relationships

If two classes in a model need to communicate with each other, there must be link between them.

An *association* denotes that link.



Student

Instructor

Object Diagrams

- Object Diagrams describe the static structure of a system at a particular time. Whereas a class model describes all possible situations, an object model describes a particular situation. Object diagrams contain the following elements:
- **Objects**, which represent particular entities. These are instances of classes.
- **Links,** which represent particular relationships between objects. These are instances of associations.

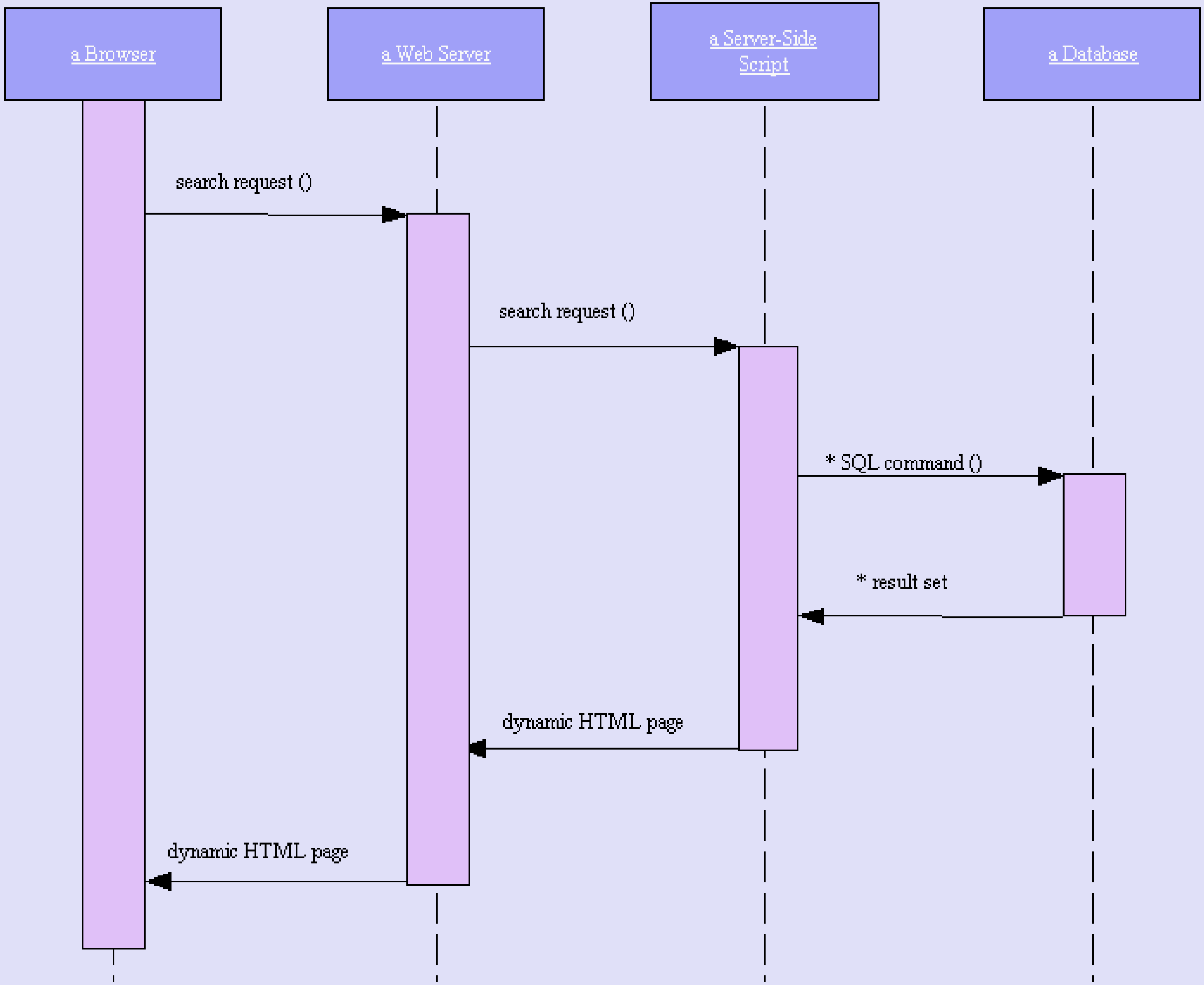
SEQUENCE DIAGRAMS

Sequence Diagrams

- Describe the dynamic behavior between actors and the system and between objects of the system
- Used during requirements analysis
  - To refine use case descriptions
  - To find additional objects
- Used during system design
  - To refine subsystem interfaces

Sequence diagram notation

- Classes are represented by columns
- Messages are represented by arrows
- Activations represented by narrow rectangles
- Lifelines are represented by dashed lines



a Browser

a Web Server

a Server-Side Script

a Database

search request ()

search request ()

* SQL command ()

* result set

dynamic HTML page

dynamic HTML page

Sequence Diagram: HTML Page

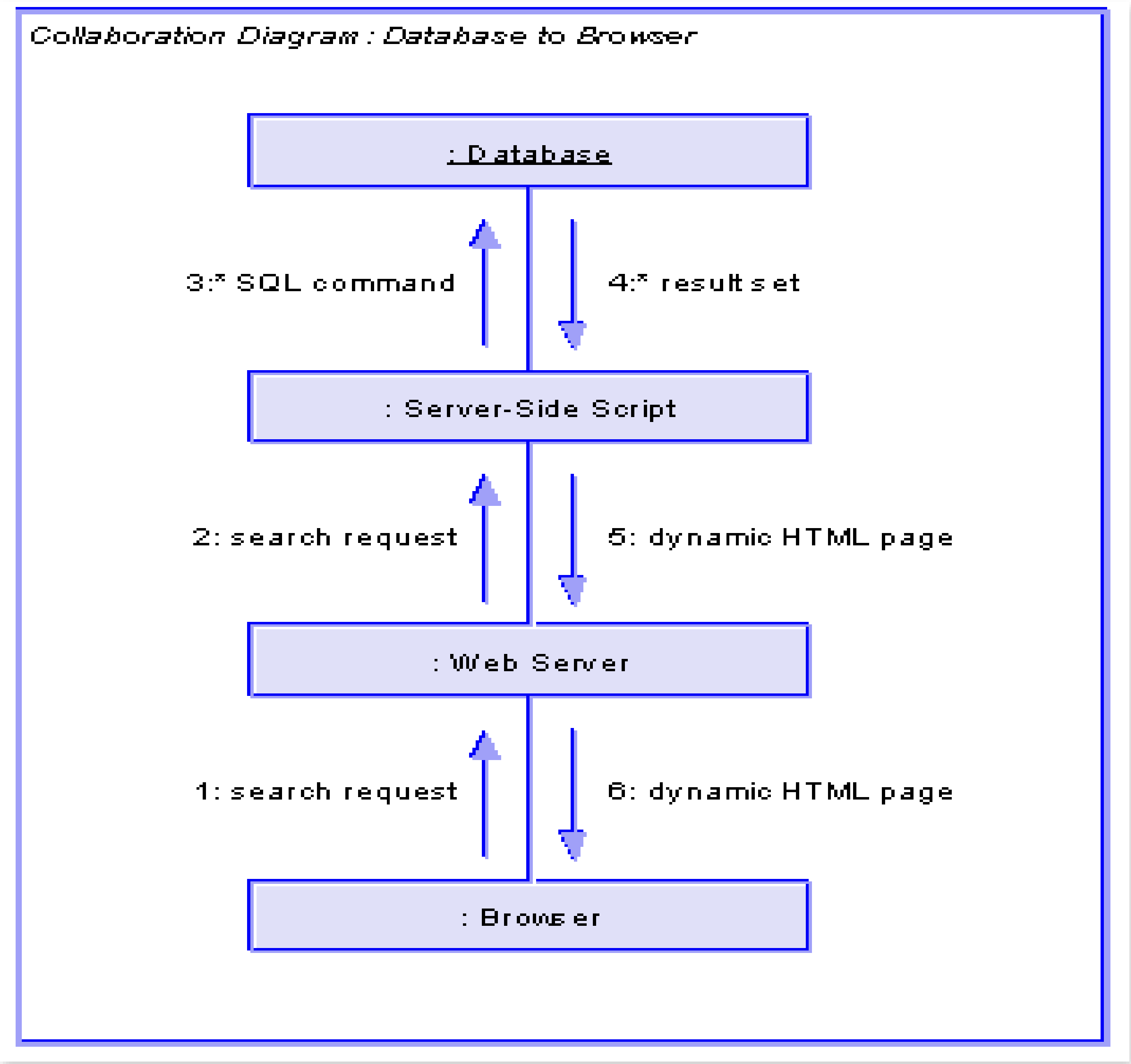
Collaboration Diagrams

- Collaboration Diagrams describe interactions among classes and associations. These interactions are modeled as exchanges of messages between classes through their associations. Collaboration diagrams are a type of interaction diagram. Collaboration diagrams contain the following elements.

- **Class roles**, which represent roles that objects may play within the interaction.

- **Association roles**, which represent roles that links may play within the interaction.

- **Message flows**, which represent messages sent between objects via links. Links transport or implement the delivery of the message.



Collaboration Diagram : Database to Browser
: Database
3:* SQL command
4:* result set
: Server-Side Script
2: search request
5: dynamic HTML page
: Web Server
1: search request
6: dynamic HTML page
: Browser

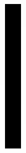
ACTIVITY DIAGRAMS

Activity Diagrams

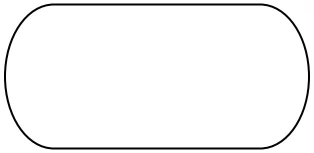
- Used to represent the behavior of a system in terms of activities and their precedence constraints
- Compared to flowchart diagrams because …
  - They can be used to represent control flow (order in which operations occur)
  - They can be used to represent data flow (objects exchanged among operations)

The completion of an activity triggers an outgoing transition which may initiate another activity

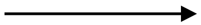
Symbols



- Synchronization of the control flow

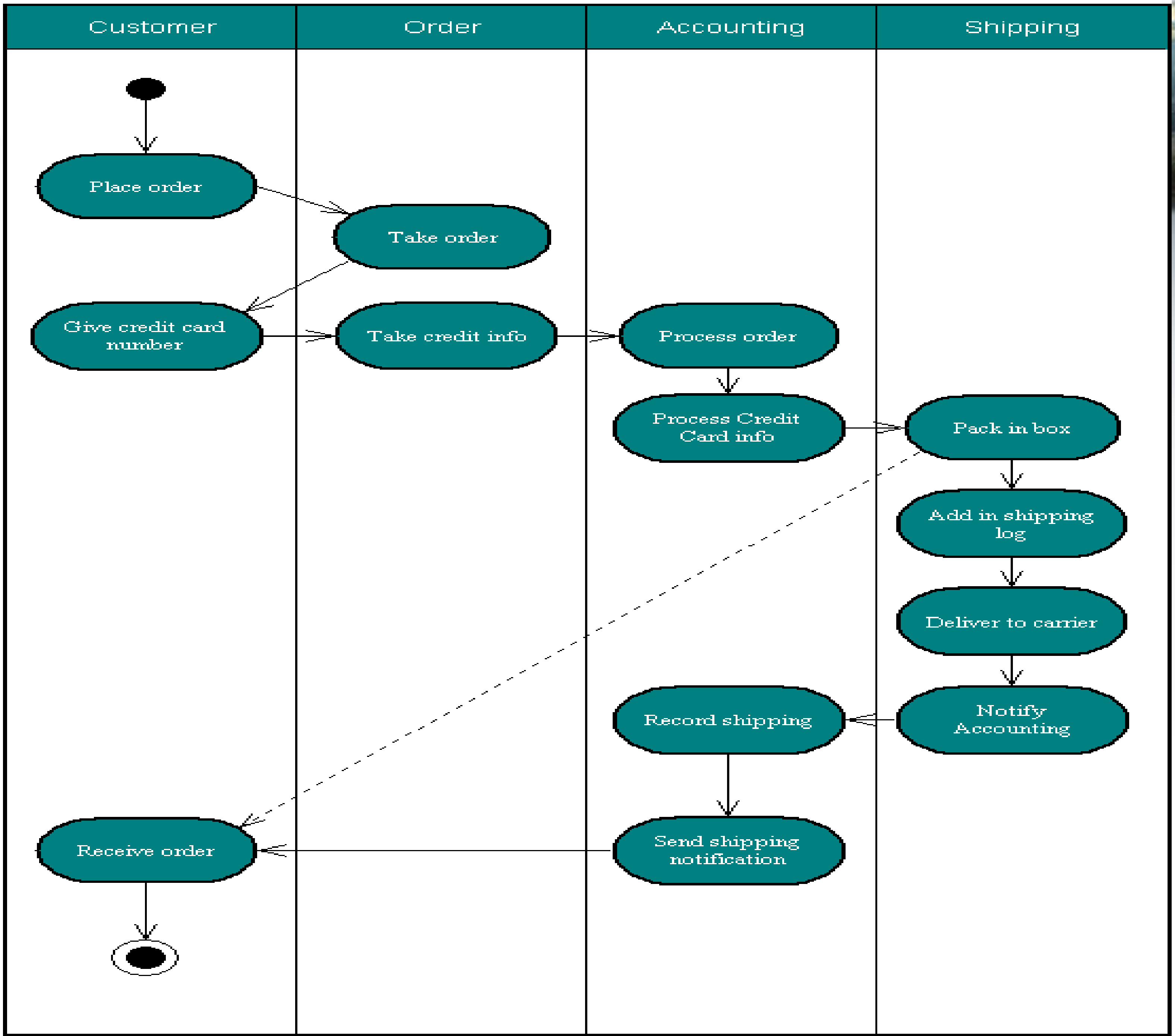


- Activities in the system



- Transitions between activities

UML Activity Diagram: Order Processing

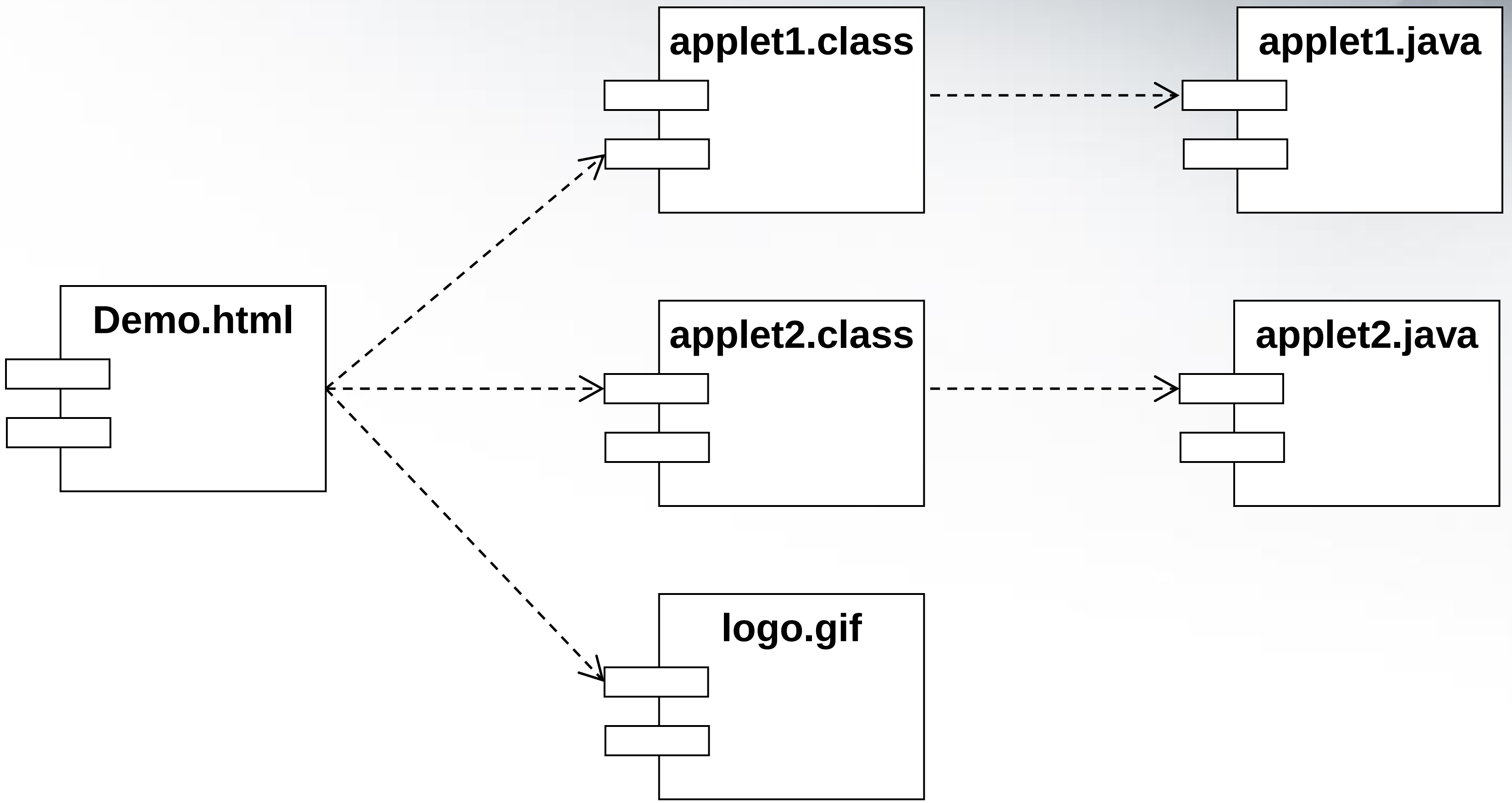


Component Diagrams

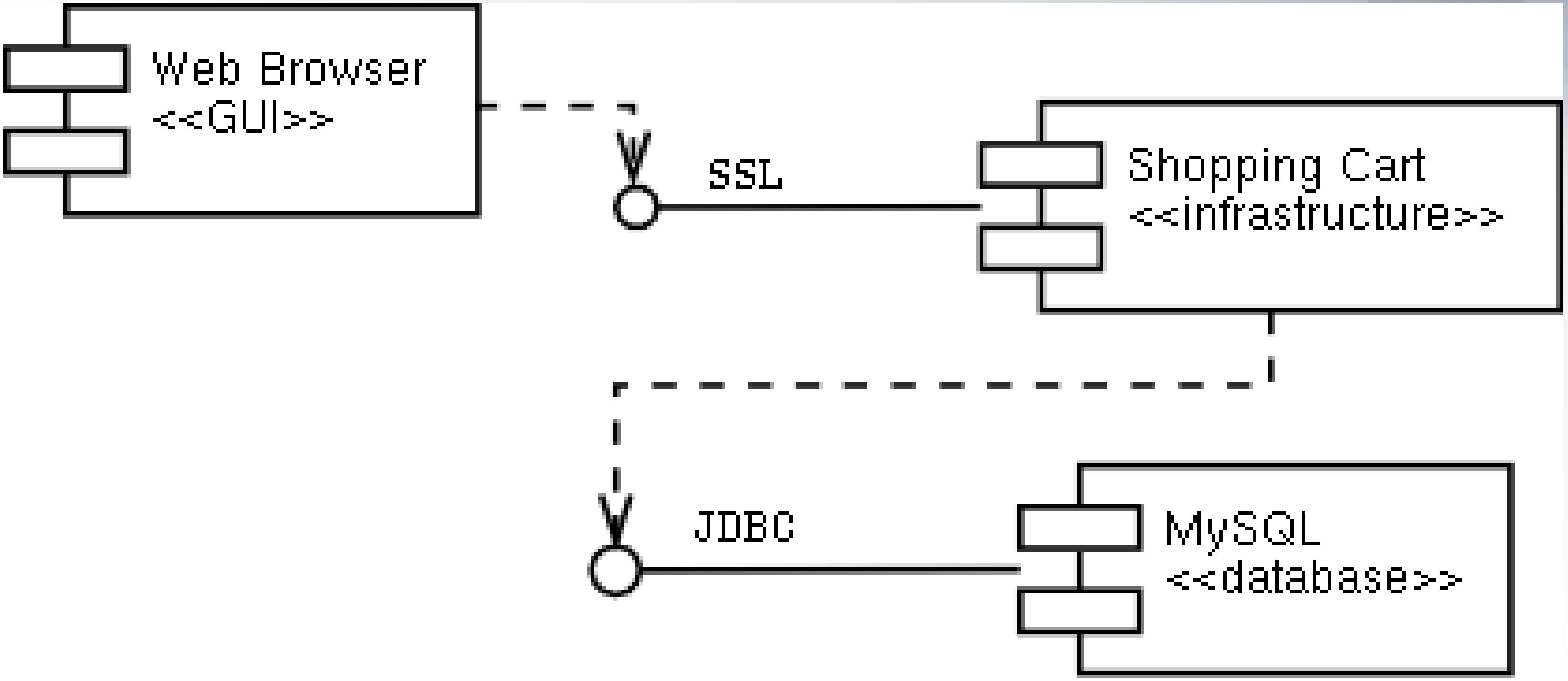
- Component diagrams describe the organization of and dependencies among software implementation components.
- These diagrams contain components, which represent distributable physical units, including source code, object code, and executable code.

Component diagram -

Captures the Physical Structure of the Implementation



Component Diagram Example

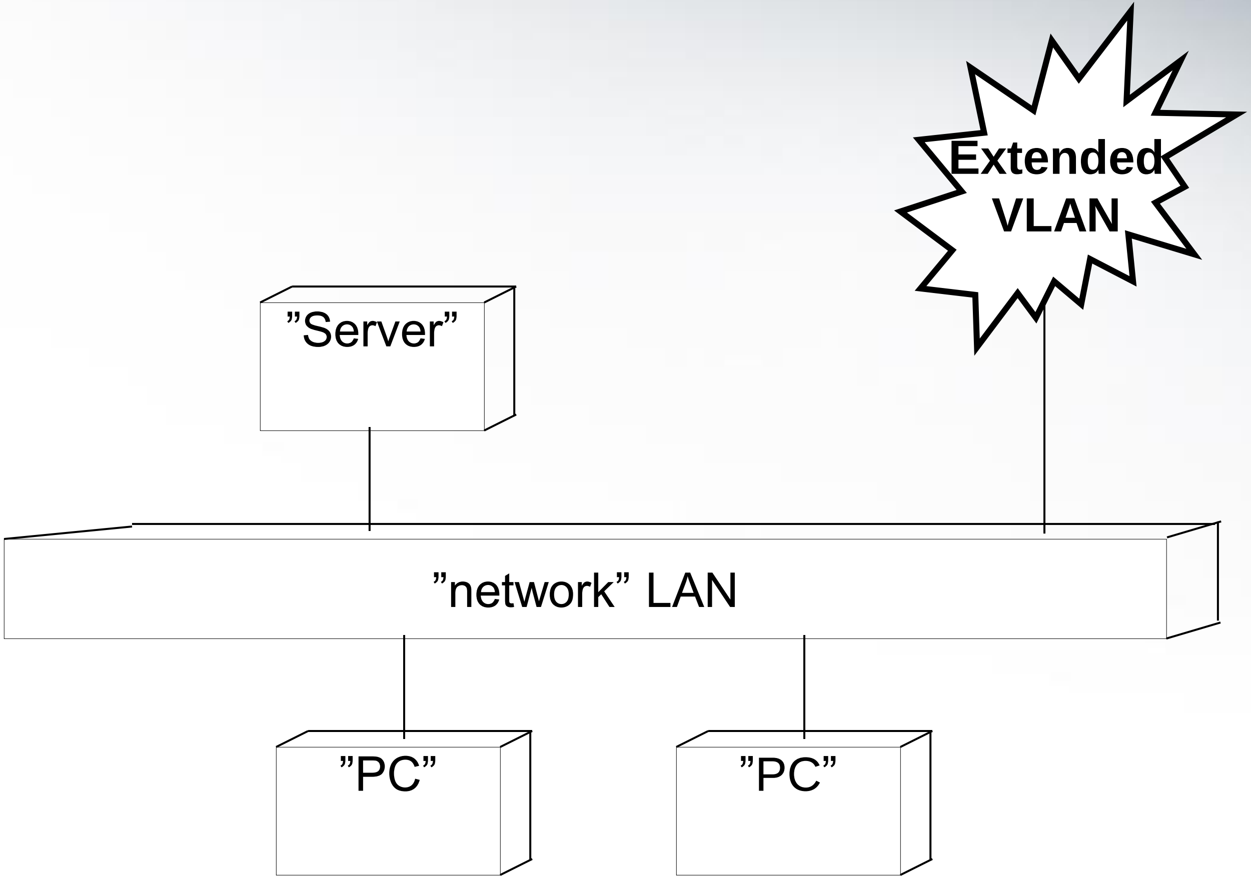


SSL = Secure Sockets Layer
JDBC = Java Database Connectivity

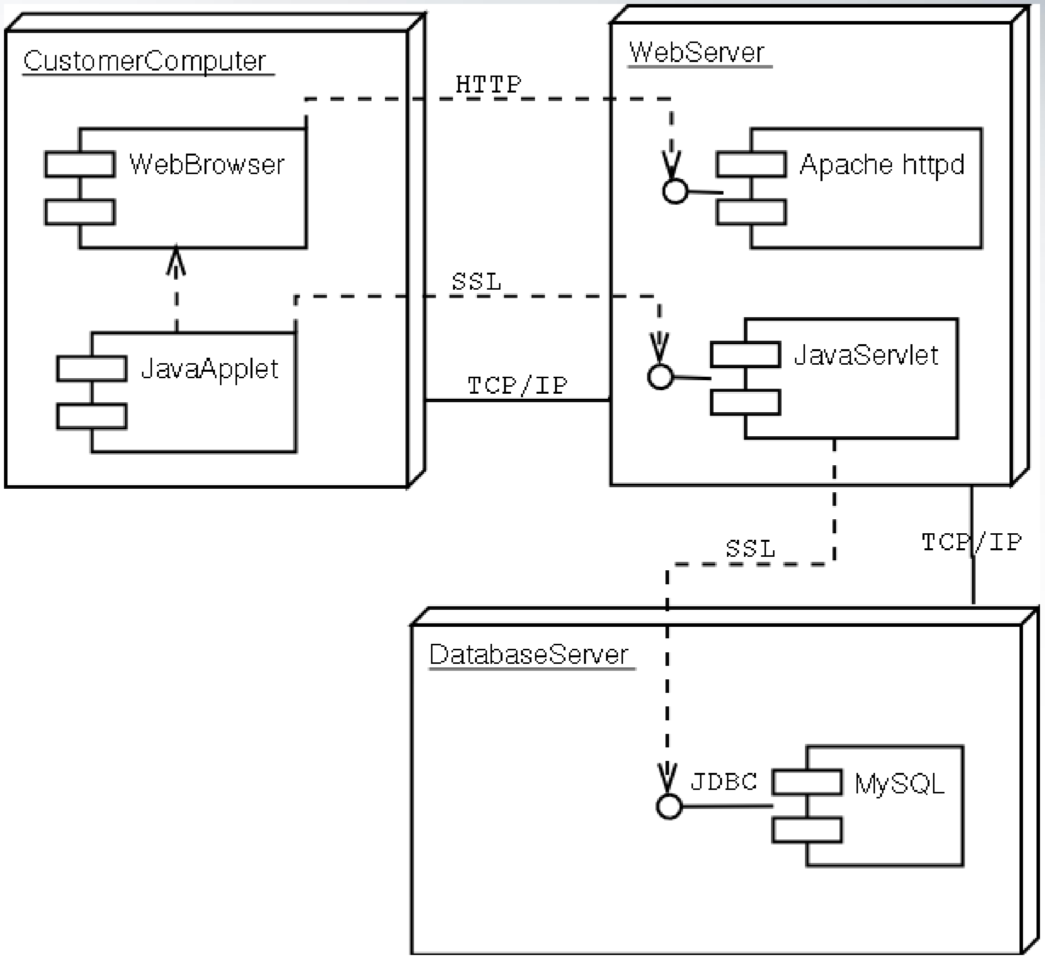
Deployment Diagrams

- Deployment diagrams describe the configuration of processing resource elements and the mapping of software implementation components onto them. These diagrams contain components and nodes, which represent processing or computational resources, including computers, printers, etc.

Deployment diagram



Example


CustomerComputer
WebBrowser
JavaApplet
HTTP
SSL
TCP/IP
WebServer
Apache httpd
JavaServlet
SSL
TCP/IP
DatabaseServer
JDBC
MySQL

STATECHART DIAGRAMS

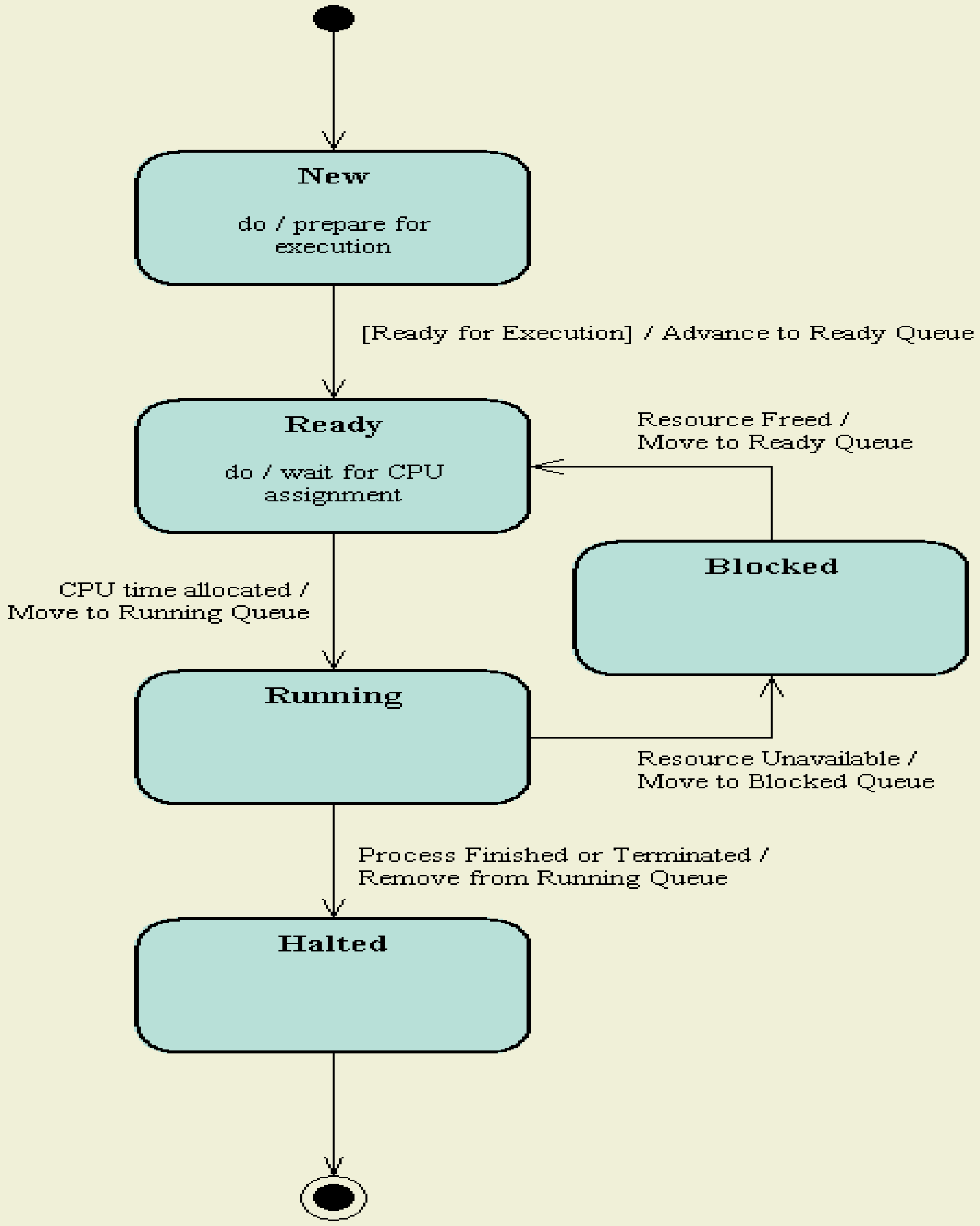
UML Statechart Diagram

- A statechart diagram shows the behavior of classes in response to external stimuli
- This diagram models the dynamic flow of control from state to state within a system.

State chart Diagrams

- State Chart (or state) diagrams describe the states and responses of a class. State Chart diagrams describe the behavior of a class in response to external stimuli. These diagrams contain the following elements:
- **States**, which represent the situations during the life of an object in which it satisfies some condition, performs some activity, or waits for some occurrence.
- **Transitions**, which represent relationships between the different states of an object.

STATE DIAGRAM : CPU EXECUTION



New

do / prepare for execution

[Ready for Execution] / Advance to Ready Queue

Ready

do / wait for CPU assignment

Resource Freed / Move to Ready Queue

Blocked

CPU time allocated / Move to Running Queue

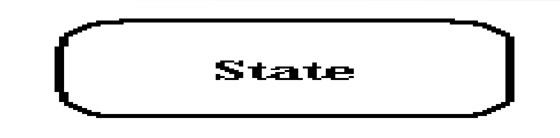
Running

Resource Unavailable / Move to Blocked Queue

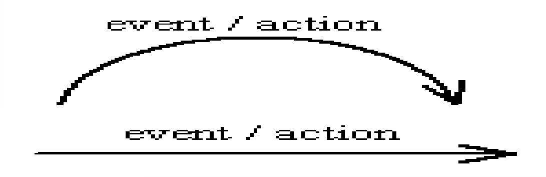
Process Finished or Terminated / Remove from Running Queue

Halted

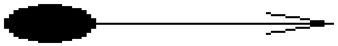
Basic Statechart Diagram Symbols and Notations



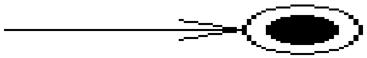
- States represent situations during the life of an object



- A solid arrow represents the path between different states of an object



- A filled circle followed by an arrow represents the object's initial state.



- An arrow pointing to a filled circle nested inside another circle represents the object's final state.

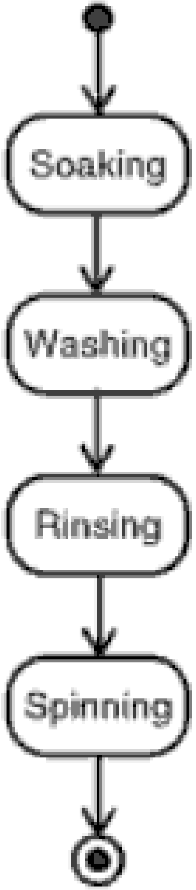
States, Transitions, Events

A **state** is a mode or condition of being.

A **transition** is a change from one state to another.

An **event** is a noteworthy occurrence at a particular time.

States of an object



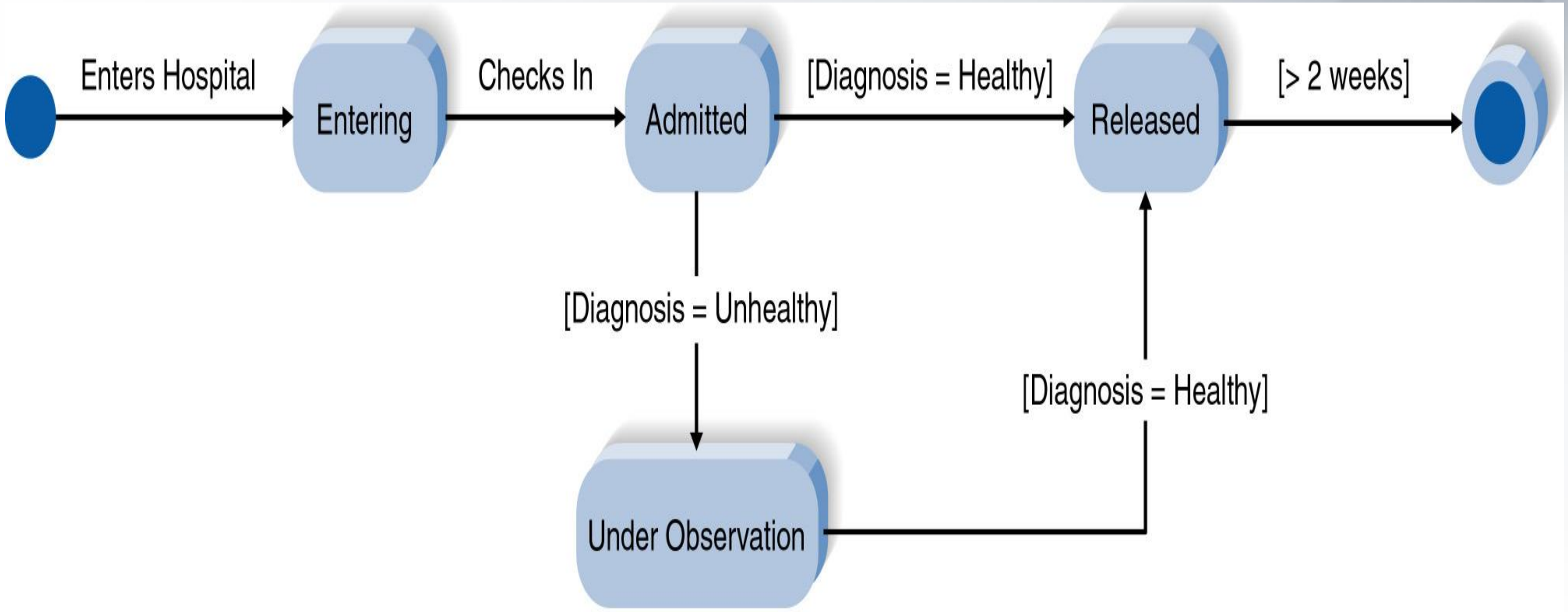
Soaking

Washing

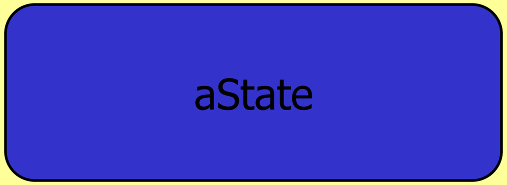
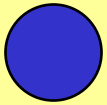
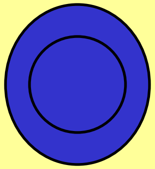
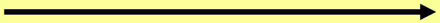
Rinsing

Spinning

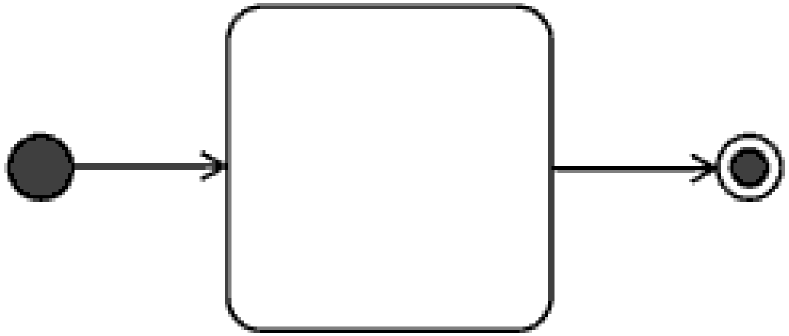
Example State Diagram

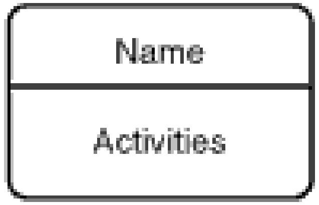


State Diagram Syntax

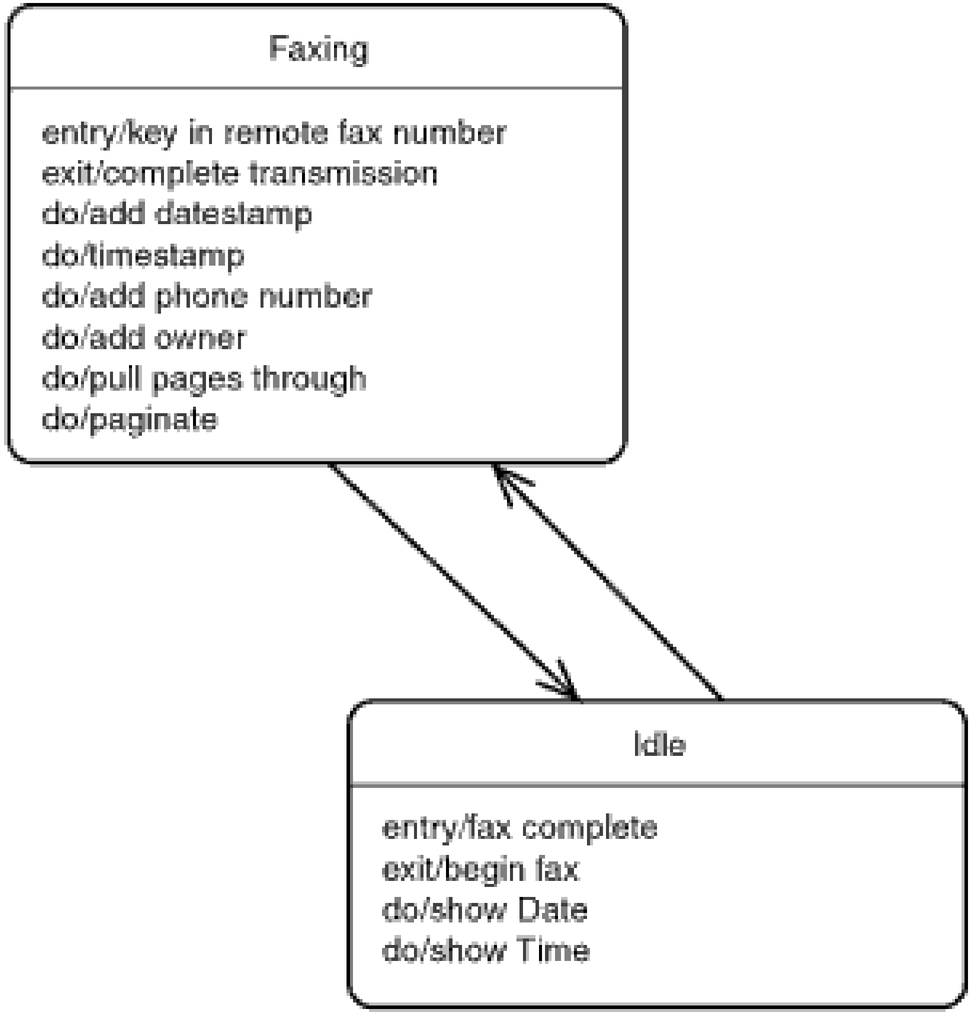
A STATE	
AN INITIAL STATE	
A FINAL STATE	
AN EVENT	anEvent
A TRANSITION	

State Diagram Syntax





State Diagram Syntax



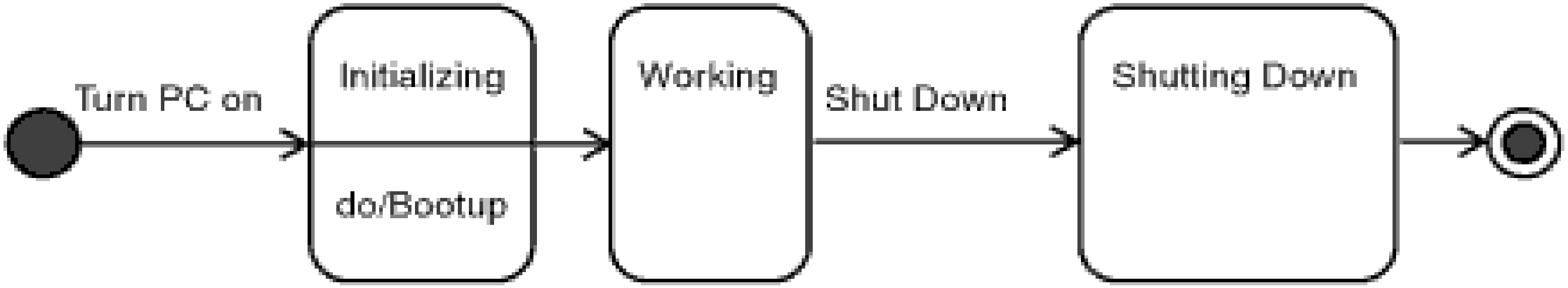
Faxing

entry/key in remote fax number
exit/complete transmission
do/add datestamp
do/timestamp
do/add phone number
do/add owner
do/pull pages through
do/paginate

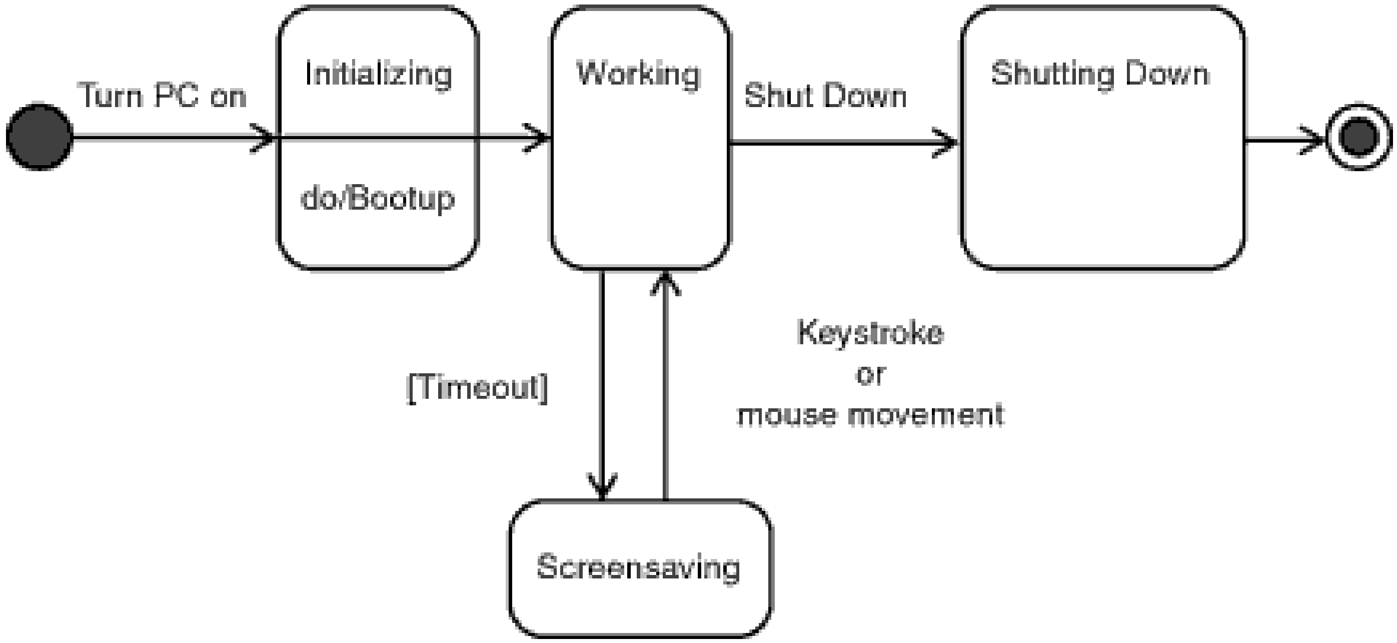
Idle

entry/fax complete
exit/begin fax
do/show Date
do/show Time

Example State Diagram



Example State Diagram



Worked example

1. Customer creates order on the Web
2. Customer submits order once finished
3. Credit authorisation needs approval before order is accepted
4. If denied, order is returned to customer for changes or deletion
5. If accepted, order is placed
6. Order is shipped to customer
7. Customer receives order
8. Order is closed

Nested State Diagrams

- State diagrams for an object may be nested, allowing the control mechanism to be viewed at different levels of abstraction.

- A nested state diagram is a form of generalisation on states.

Another Example - Digital Watch

A simple digital watch has a display and two buttons to set it, the A button and the B button. The watch has two modes of operation, display time and set time.

In the display time mode, hours and minutes are displayed.

The set time mode has two sub modes: set hours and set minutes.

The A button is used to select modes. Each time it is pressed, it advances through the modes: display, set hours, set minutes, display...etc. Within the sub modes, pressing B will advance the hours or minutes each time it is pressed. Buttons must be released before they generate another event.

States, Transitions, Events

A **state** is a mode or condition of being.

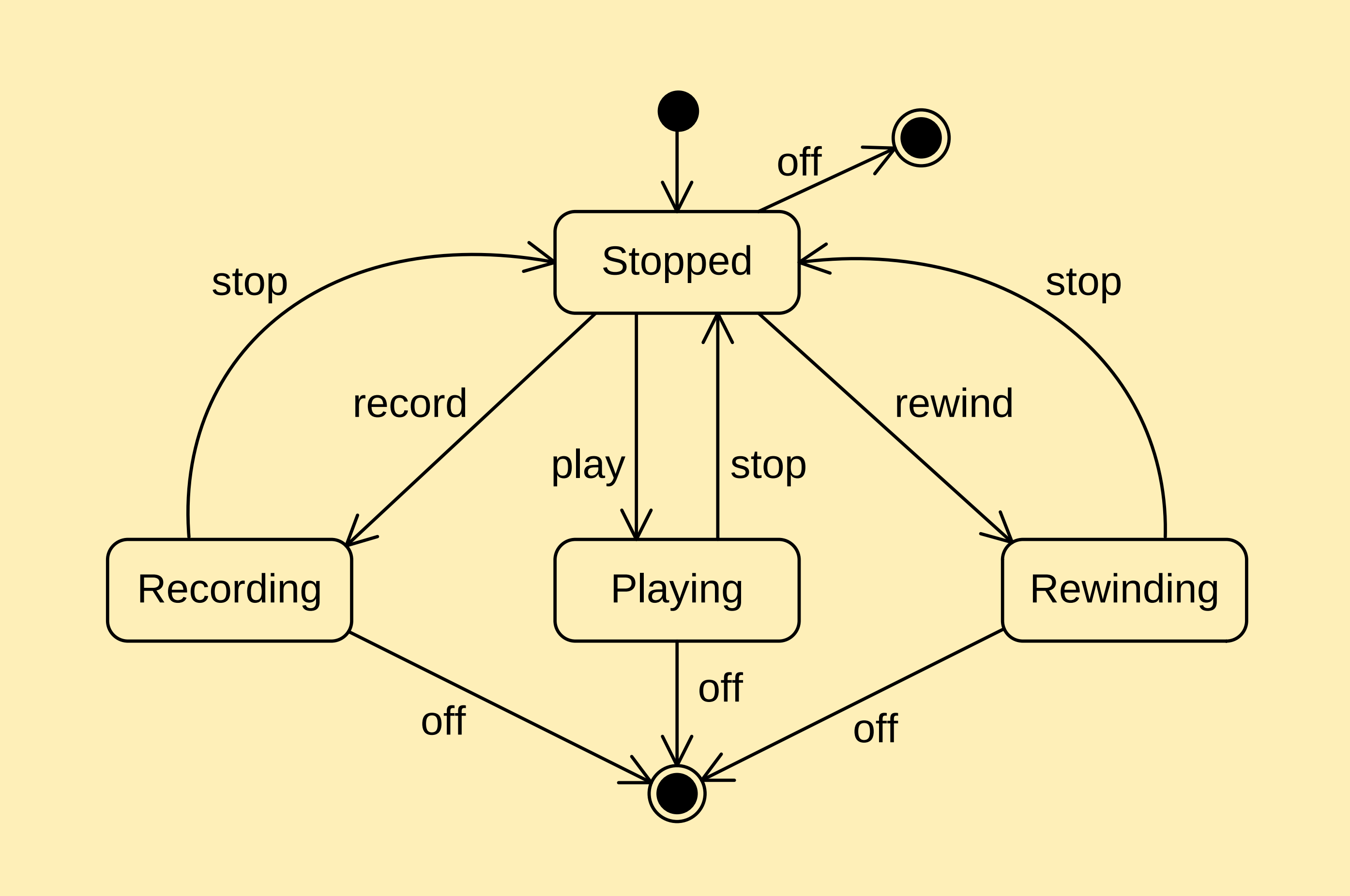
A **transition** is a change from
one state to another.

An **event** is a noteworthy
occurrence at a particular time.

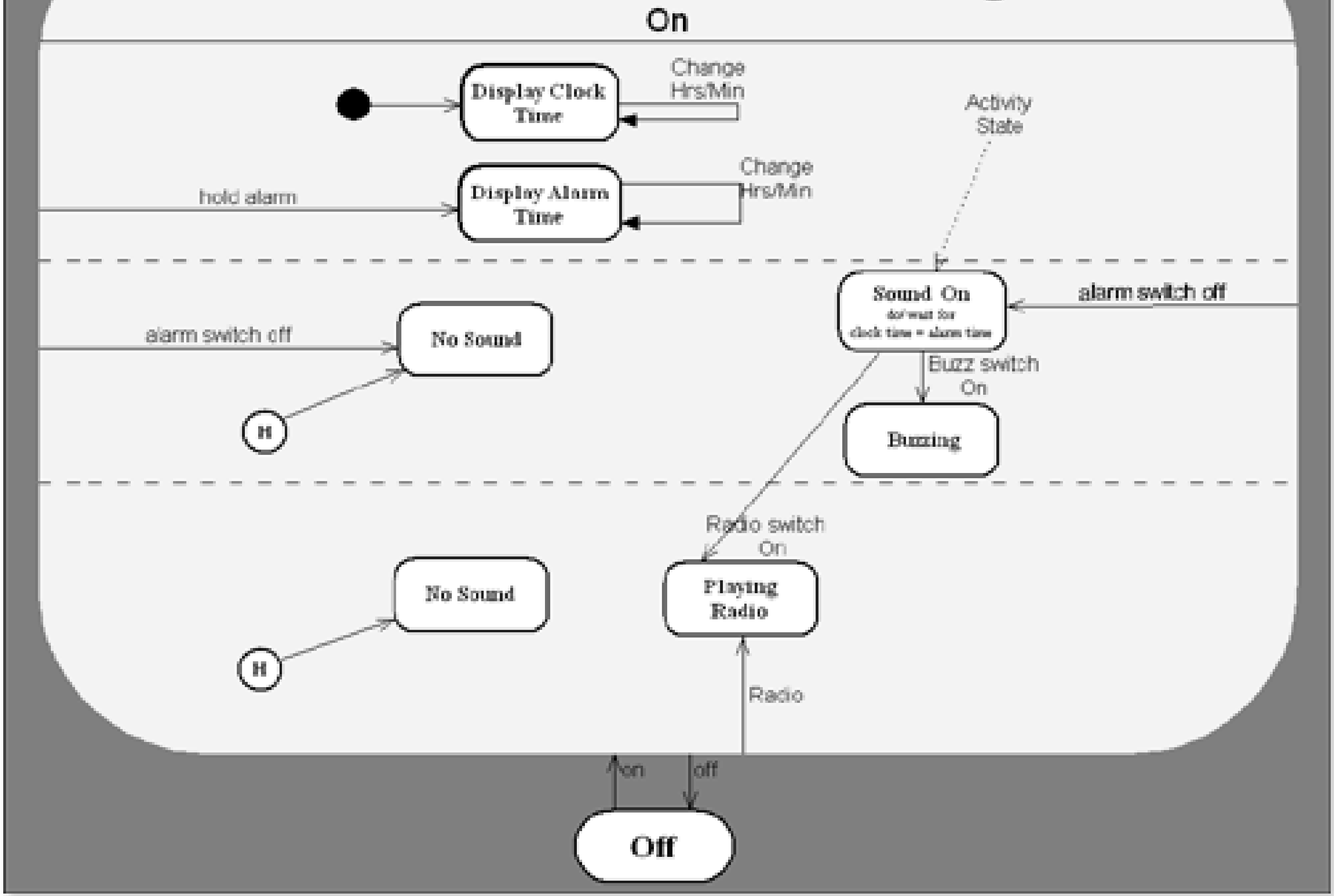
UML State Diagrams

- Represent states by rounded rectangles containing the state name
- Represent transitions by solid arrows labeled with one or more transition strings
- Transition strings
  - Describe triggering circumstances
  - Actions that result
- Initial pseudo-state designates the initial state
- Optional final state represents halting

State Diagram Example



Concurrent State Alarm Clock Diagram



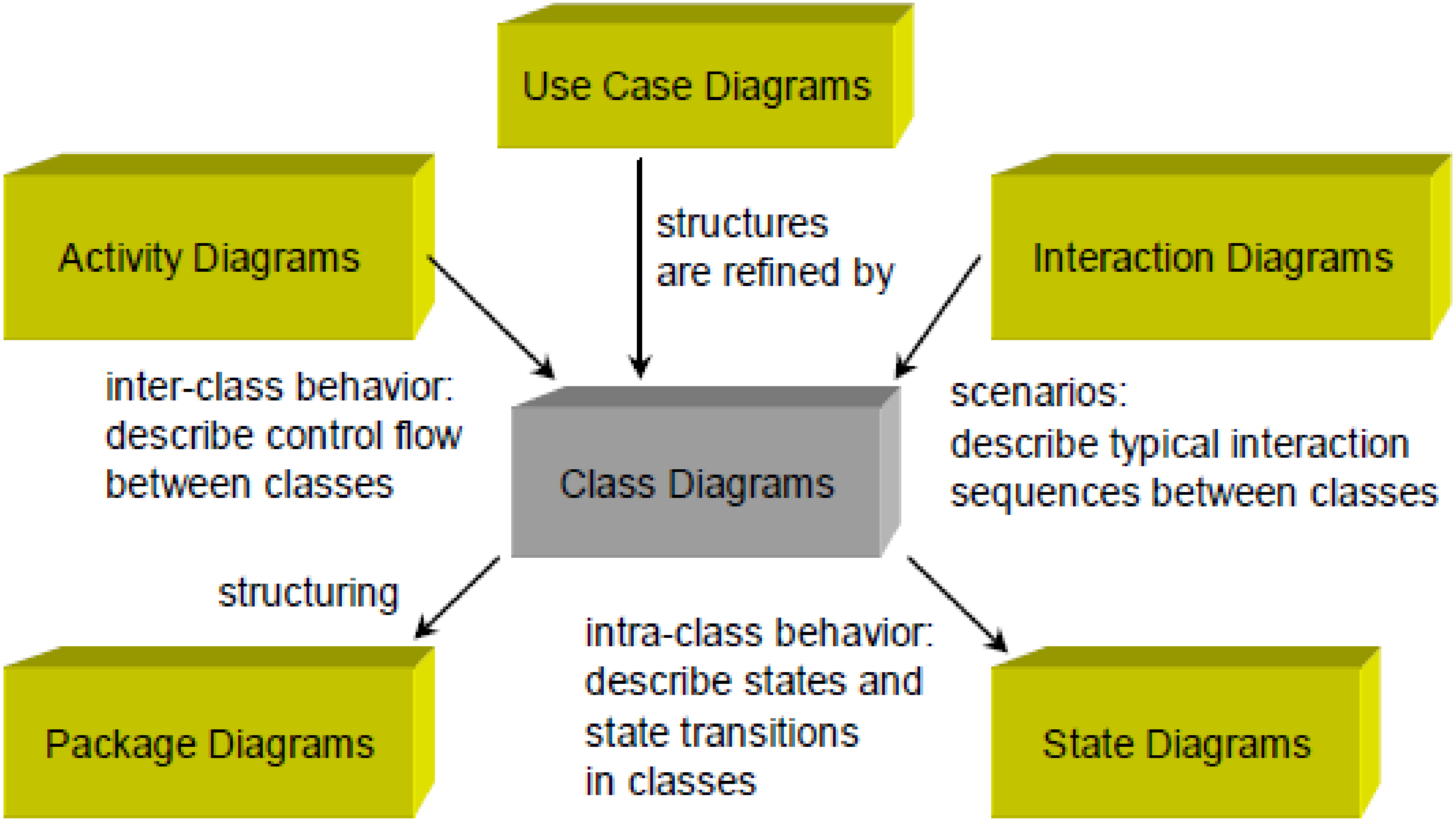
Description of state chart diagram

- **State:**
  - Display alarm time
  - Display clock time
  - No sound
  - Sound on
  - Buzzing
  - Playing radio

- **Transition:**
  - On
  - Off
  - Hold alarm
  - Alarm switch off
  - Change Hrs/Min
  - Radio

- **Activity:**
  - Do/wait for clock time to alarm time

Role of Class Diagrams



Use Case Diagrams

structures
are refined by

Activity Diagrams

Interaction Diagrams

inter-class behavior:
describe control flow
between classes

Class Diagrams

scenarios:
describe typical interaction
sequences between classes

structuring

intra-class behavior:
describe states and
state transitions
in classes

Package Diagrams

State Diagrams

Class diagrams are **central** for analysis, design and implementation.
Class diagrams are the **richest** notation in UML.

Case Study on Online Shopping Portal

In this case, we will be designing a simple online shopping portal .The site will provide a soothing shopping experience for customers. The system will allow more than one categories and different brands under the segment. The case will trace the following sequence.

Scenario

- A customer visits the online shopping portal. A customer may buy item or just visit the page and logout. The customer can select a segment, then a category, and brand to get the different products in the desired brand. The customer can select the product for purchasing. The process can be repeated for more items. Once the customer finishes selecting the product/s the cart can be viewed, If the customer wants to edit the final cart it can be done here. For final payment the customer has to login the portal, if the customer is visiting for the 1st time he must register with the site, else the customer must use the login page to proceed. Final cart is submitted for payment and card details and address (where shipment has to be made) are be confirmed by the customer .Customer is confirmed with a shipment Id and delivery of goods within 15 days.

Assumptions

- The currency followed is Rs as the site provides for only Indian customer base.
- There are different segments, categories, brands where a brand can fall under more than one category.
- The shipment of the goods is not covered under the scope of the case.
- Complains by the customers are not handled by the case study.
- Customers have to be validated before the payment can be confirmed.
- Draw the Use Case Diagram, Class Diagram, Activity Diagram..
- Sequence Diagram for the Following Activities:

1) For Register A New User.

2) For Adding A New Product to System.

Actors

Two actors

- Customer
- Admin

Use-cases

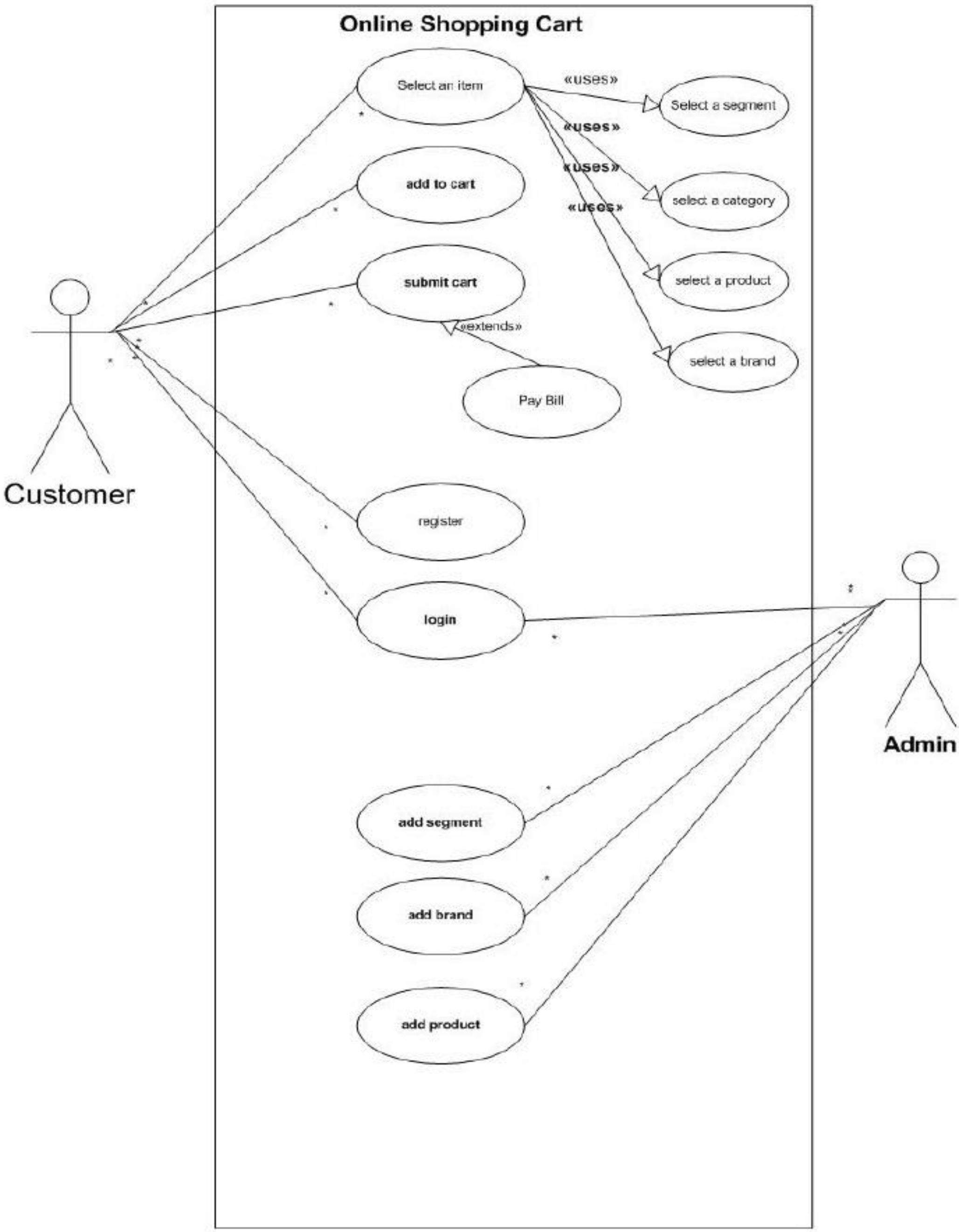
- Select
- Add to cart
- Submit Cart
  - Pay Bill

- If the customer is visiting for the 1st time he must register with the site, else the customer must use the login page to proceed.
  - Register
  - Login

- Add Segment
- Add Brand
- Add Product

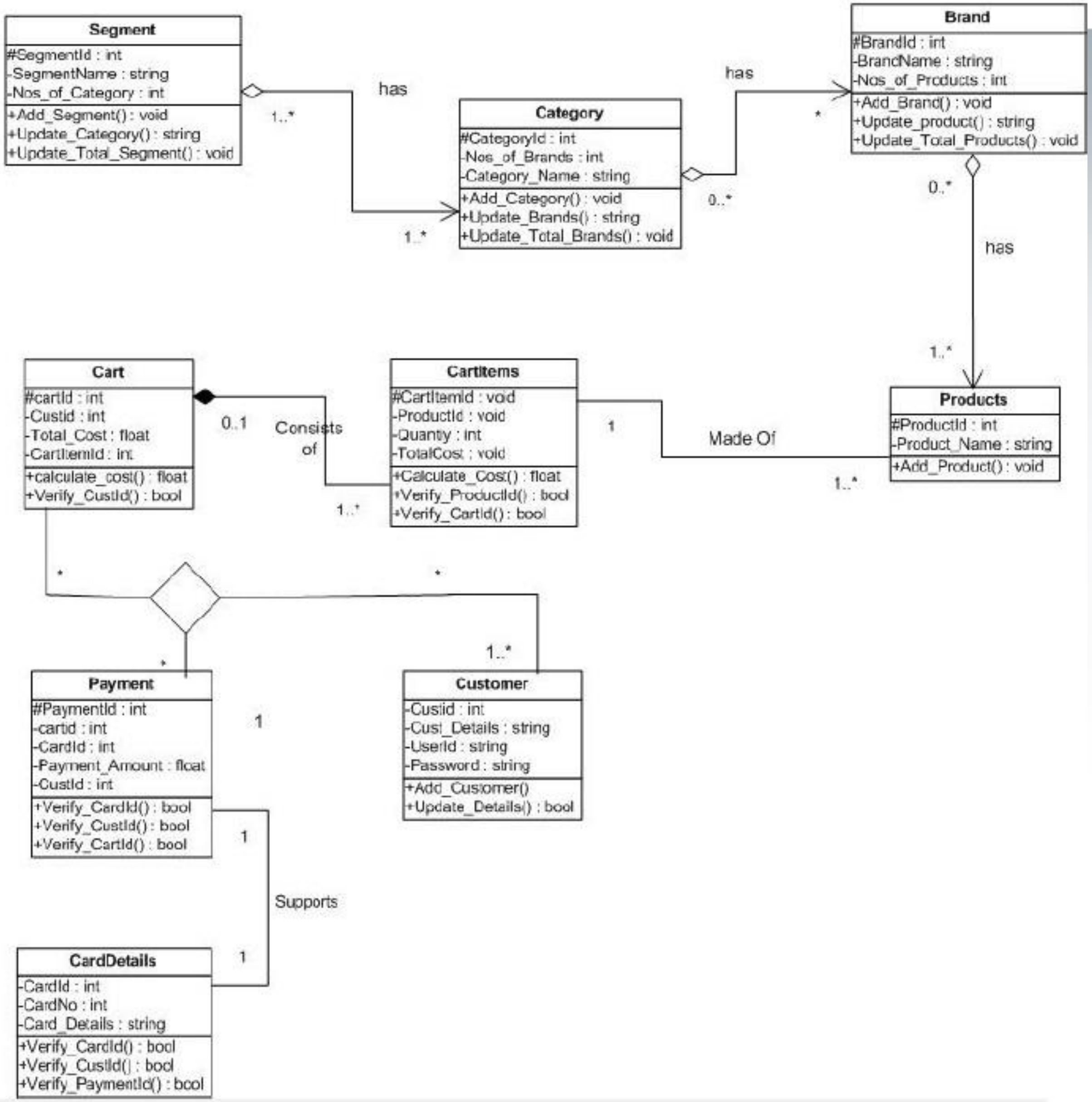
The customer can select a segment, then a category, and brand to get the different products in the desired brand.

- Select a Segment
- Select a Category
- Select a Product
- Select a Brand

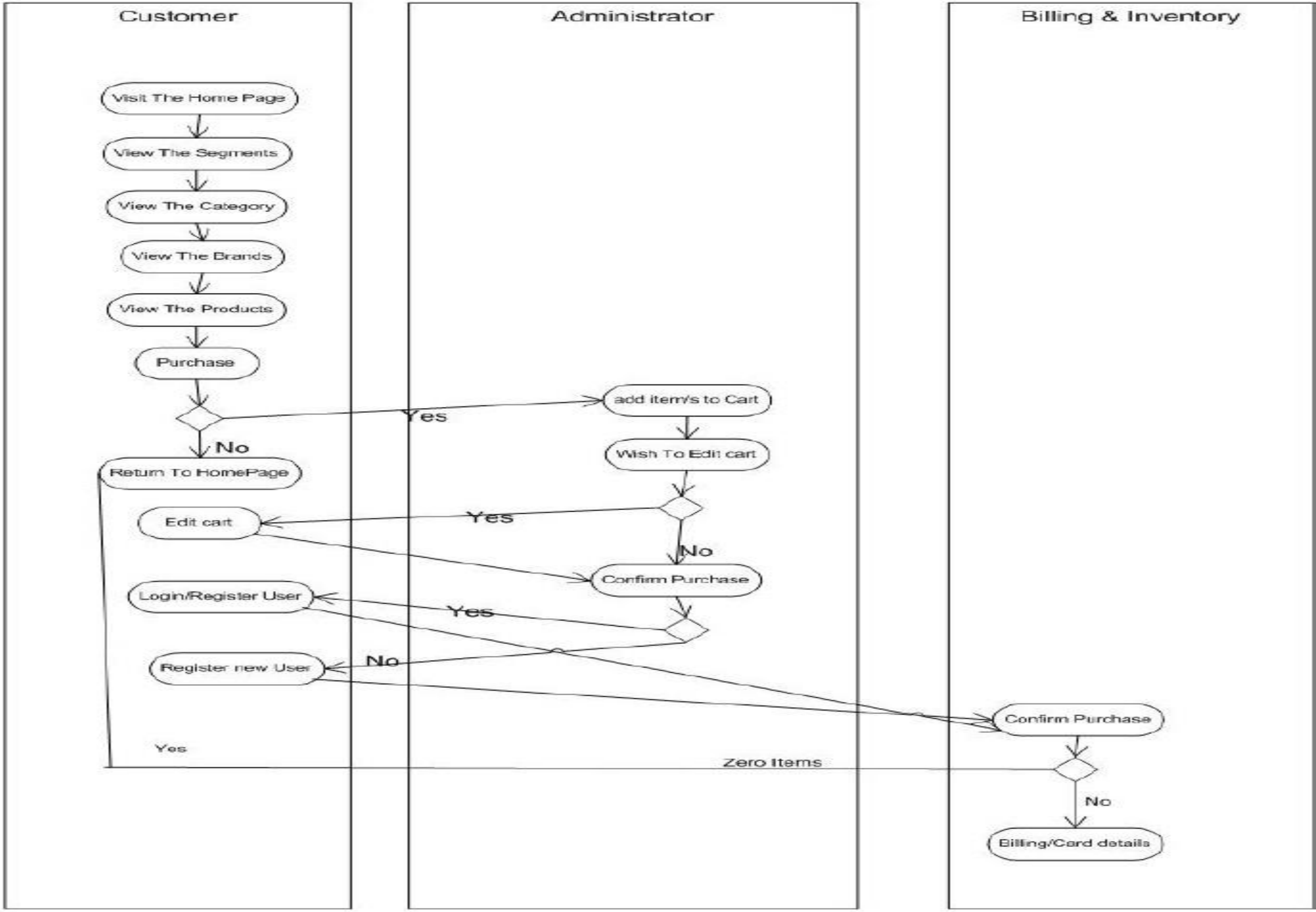

Online Shopping Cart
Select an item
«uses»
Select a segment
«uses»
«uses»
select a category
«uses»
select a product
select a brand
add to cart
submit cart
«extends»
Pay Bill
register
login
add segment
add brand
add product
Customer
Admin

- The customer can select a segment, then a category, and brand to get the different products in the desired brand.

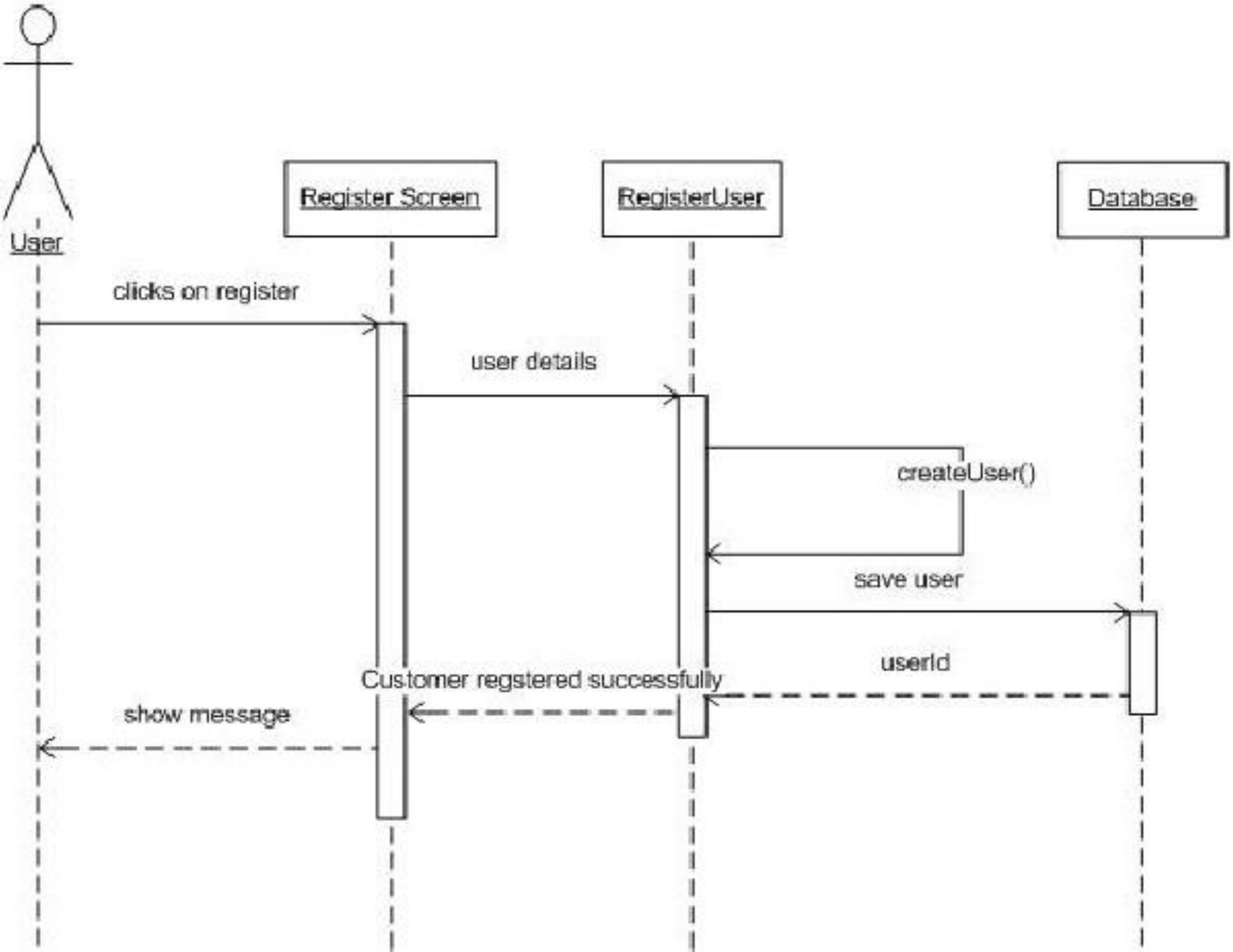
CLASS DIAGRAM



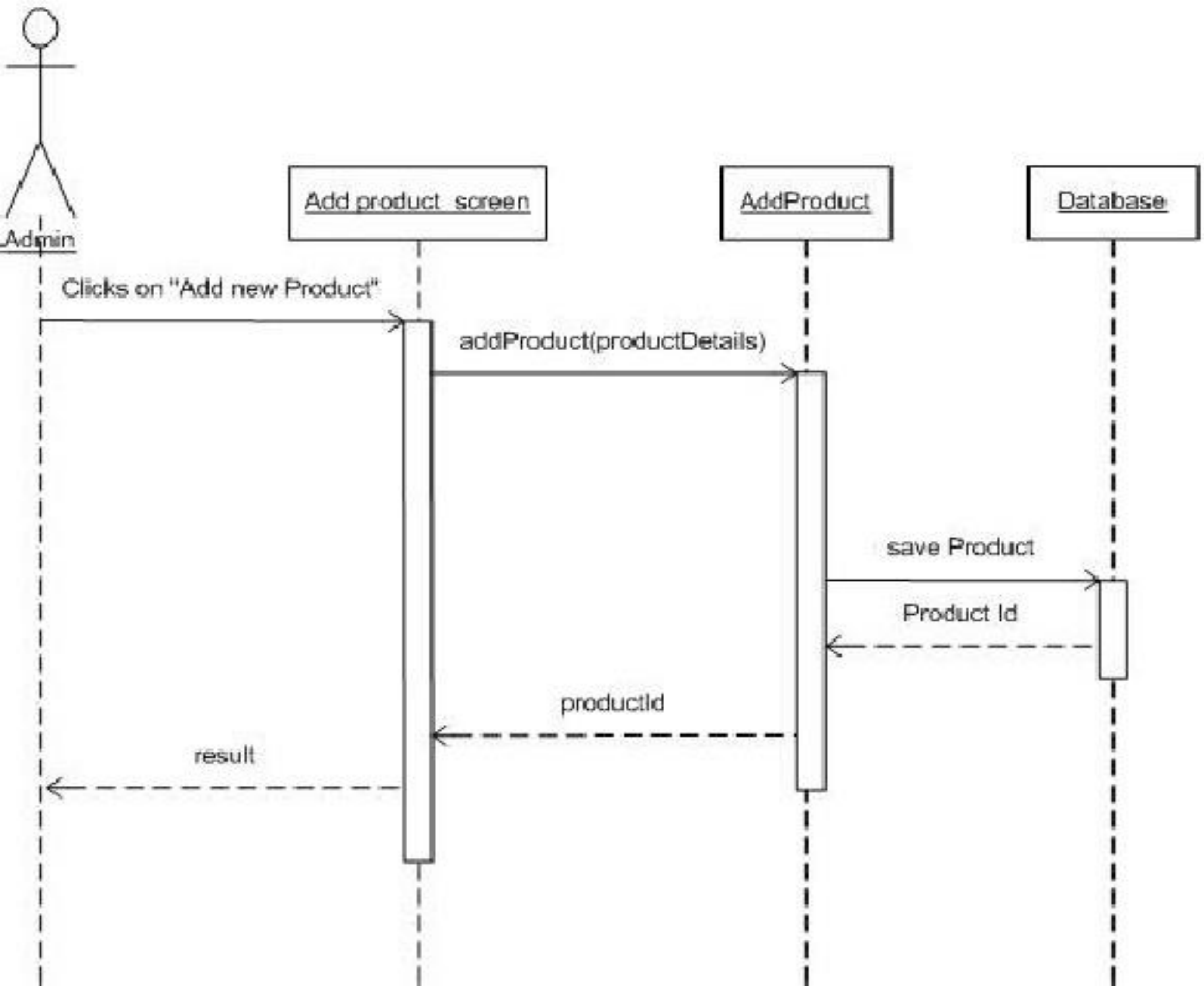
ACTIVITY DIAGRAM



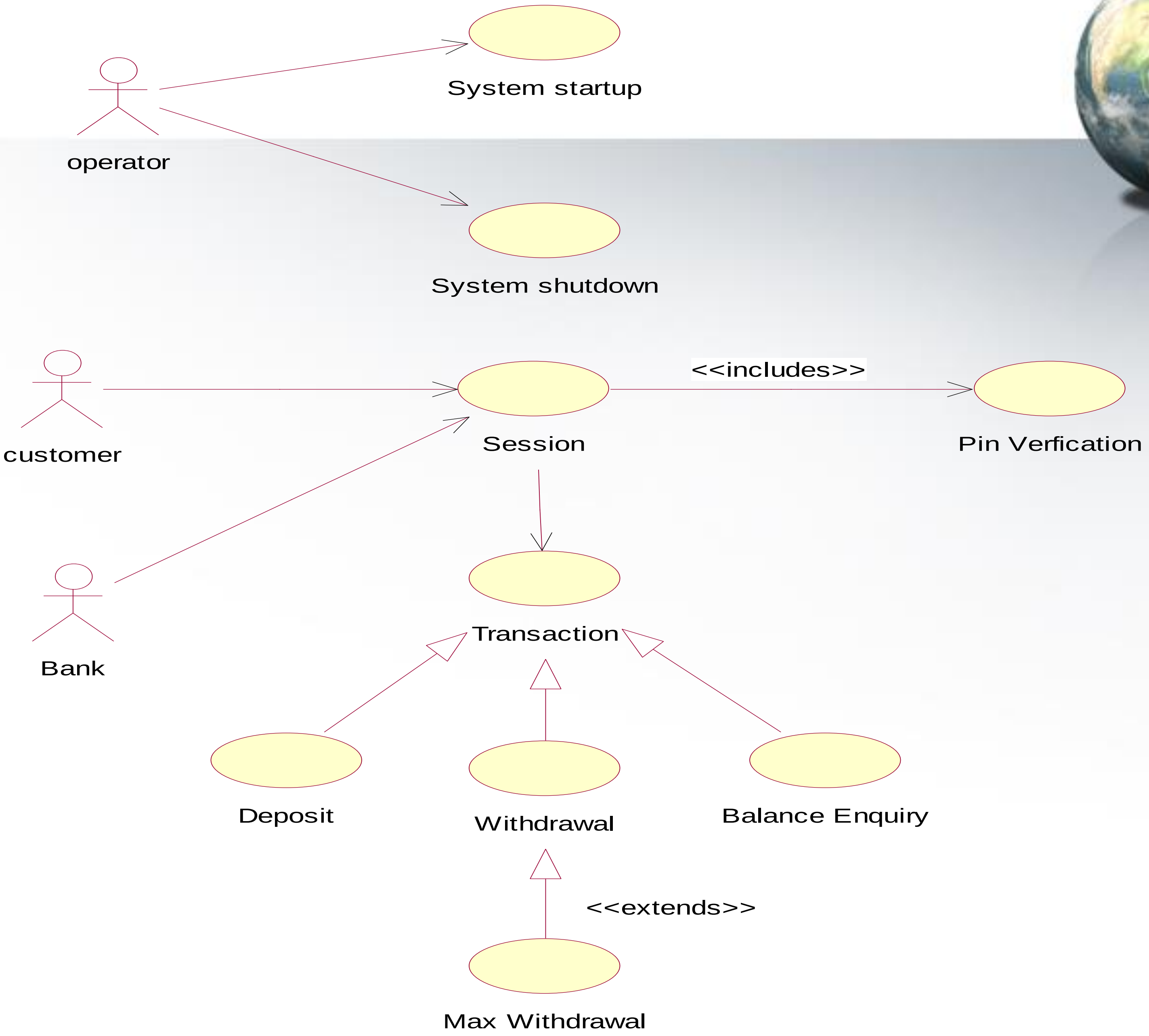
Use Case : Register a user



Use Case : Add a new product



- Draw Use Case diagram, Class Diagram and Sequence diagram for Automated Teller Machine (ATM). The ATMs will allow customers to access details of their account, such as the balance enquiry, withdraw cash provided they have sufficient funds remaining, fund transfer etc. Customers will use the ATM by inserting their account card and entering a 4-digit PIN. Customers should be able to change this PIN via the ATM as well.



operator

System startup

System shutdown

customer

Bank

Session

<<includes>>

Pin Verfication

Transaction

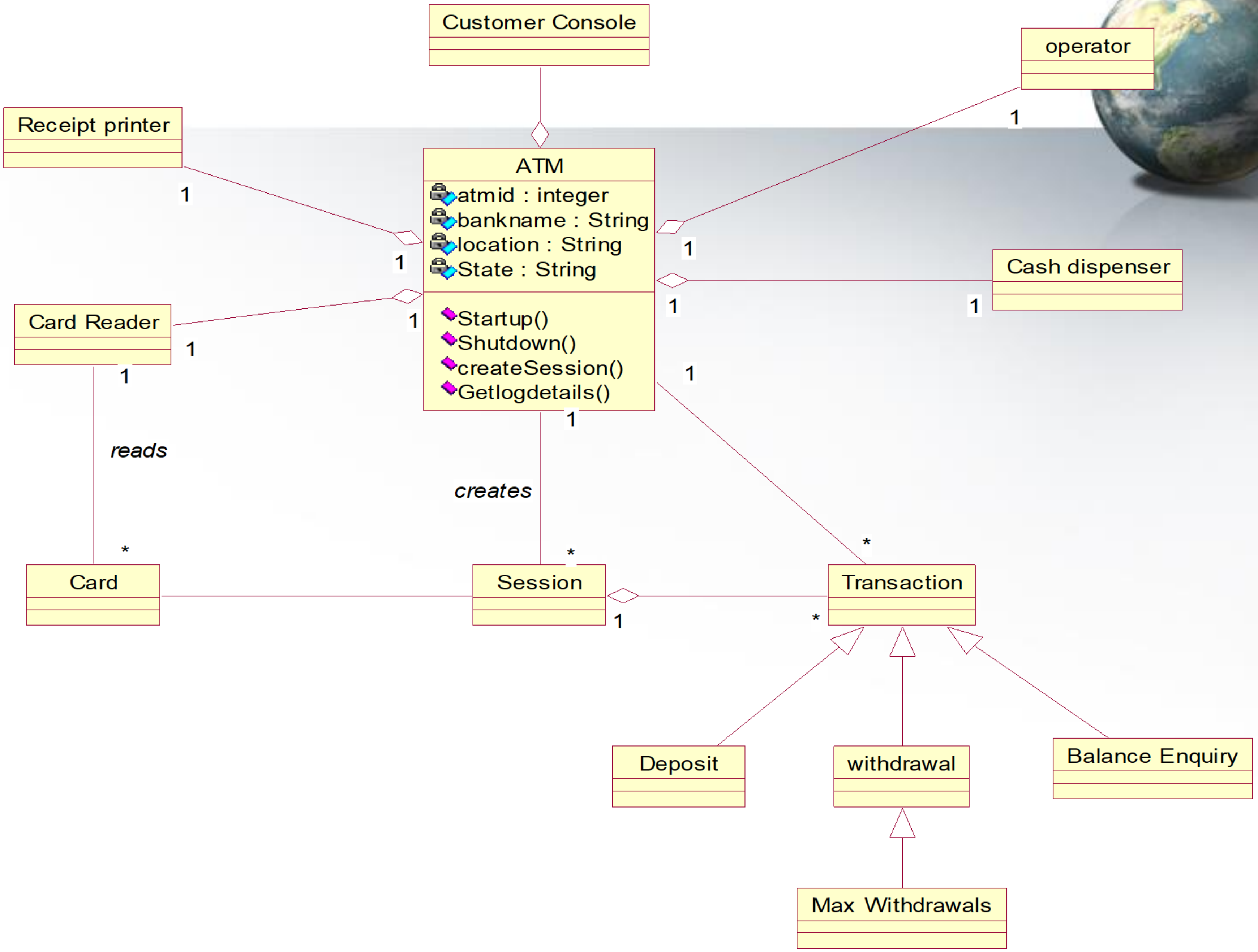
Deposit

Withdrawal

Balance Enquiry

<<extends>>

Max Withdrawal



Customer Console

Receipt printer

operator

ATM

atmid : integer

bankname : String

location : String

State : String

Startup()

Shutdown()

createSession()

Getlogdetails()

Cash dispenser

Card Reader

Card

Session

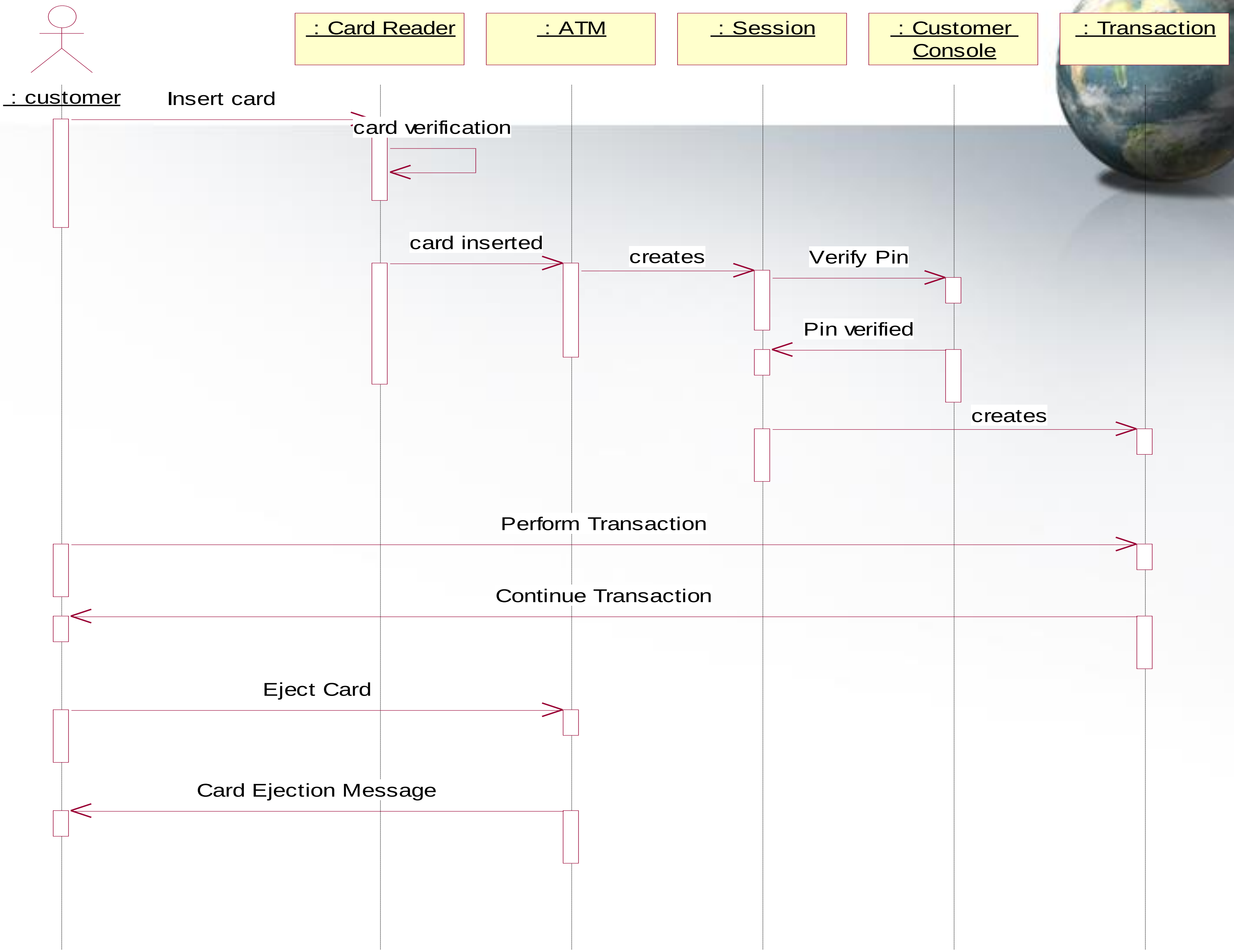
Transaction

Deposit

withdrawal

Balance Enquiry

Max Withdrawals



: customer

: Card Reader

: ATM

: Session

: Customer Console

: Transaction

Insert card

card verification

card inserted

creates

Verify Pin

Pin verified

creates

Perform Transaction

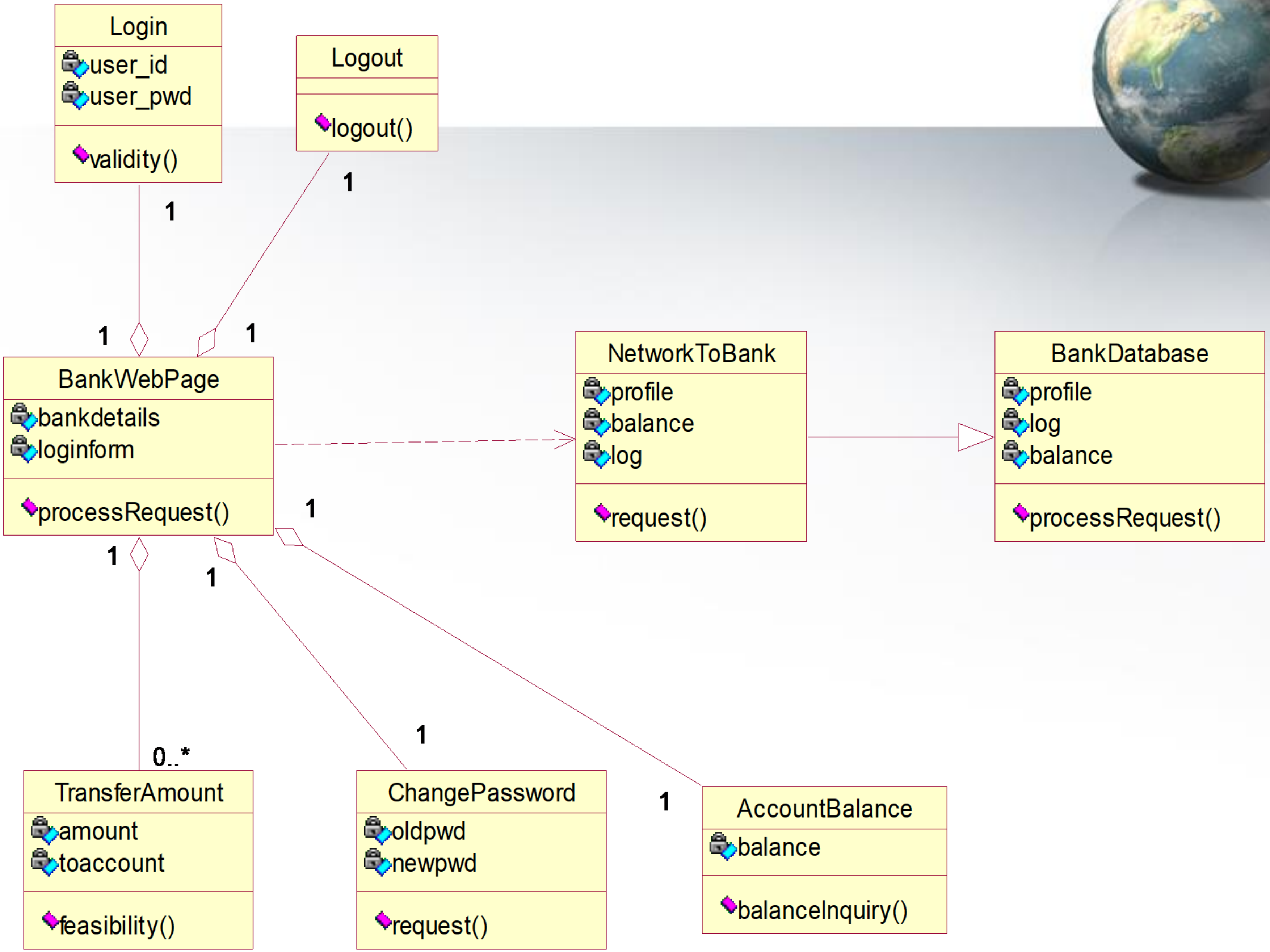
Continue Transaction

Eject Card

Card Ejection Message

Online Transaction System

- Account transaction begins when customer is successfully logged in to the site. Several menus were displayed related to profile of customer and the recent transactions and the current account balance. The main purpose of using online account transactions is to transfer cash from one account to another for this purpose the customer is provided fields to specify the accounts to which he is transferring amount. After every transaction a confirmation is displayed to customer. The customer is also provided the possibility to change the account login password, but not the user id, every transaction is added to the bank database.

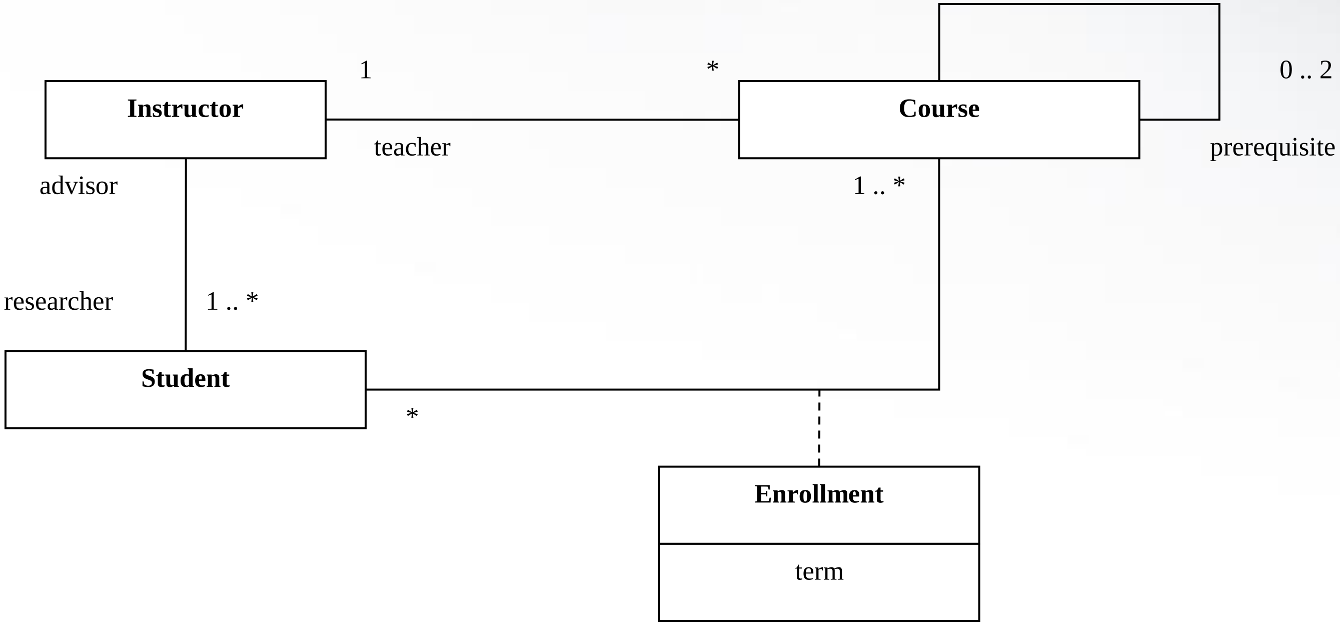

Login
user_id
user_pwd
validity()
Logout
logout()
1
1
1
1
BankWebPage
bankdetails
loginform
processRequest()
NetworkToBank
profile
balance
log
request()
BankDatabase
profile
log
balance
processRequest()
1
1
1
0..*
1
1
TransferAmount
amount
toaccount
feasibility()
ChangePassword
oldpwd
newpwd
request()
AccountBalance
balance
balanceInquiry()

- Given the following assertions for a relational database that represents the current term enrollment at a large university, draw an UML diagram for this schema that takes into account all the assertions given. There are 2000 instructors, 4000 courses, and 30,000 students.

Use as many UML constructs as you can to represent the true semantics of the problem.

Assertions:

- An instructor may teach none, one, or more courses (average is 2.0 courses)
- An instructor must advise the research of at least one student (average = 2.5 students)
- A course may have none, one, or two prerequisites (average = 1.5 prerequisites)
- A course may exist even if no students are currently enrolled
- All courses are taught by only one instructor.
- The average enrollment in a course is 30 students.
- A student must select at least one course per term (average = 4.0 course selections).



Instructor

1

*

teacher

Course

0 .. 2

prerequisite

advisor

1 .. *

researcher

1 .. *

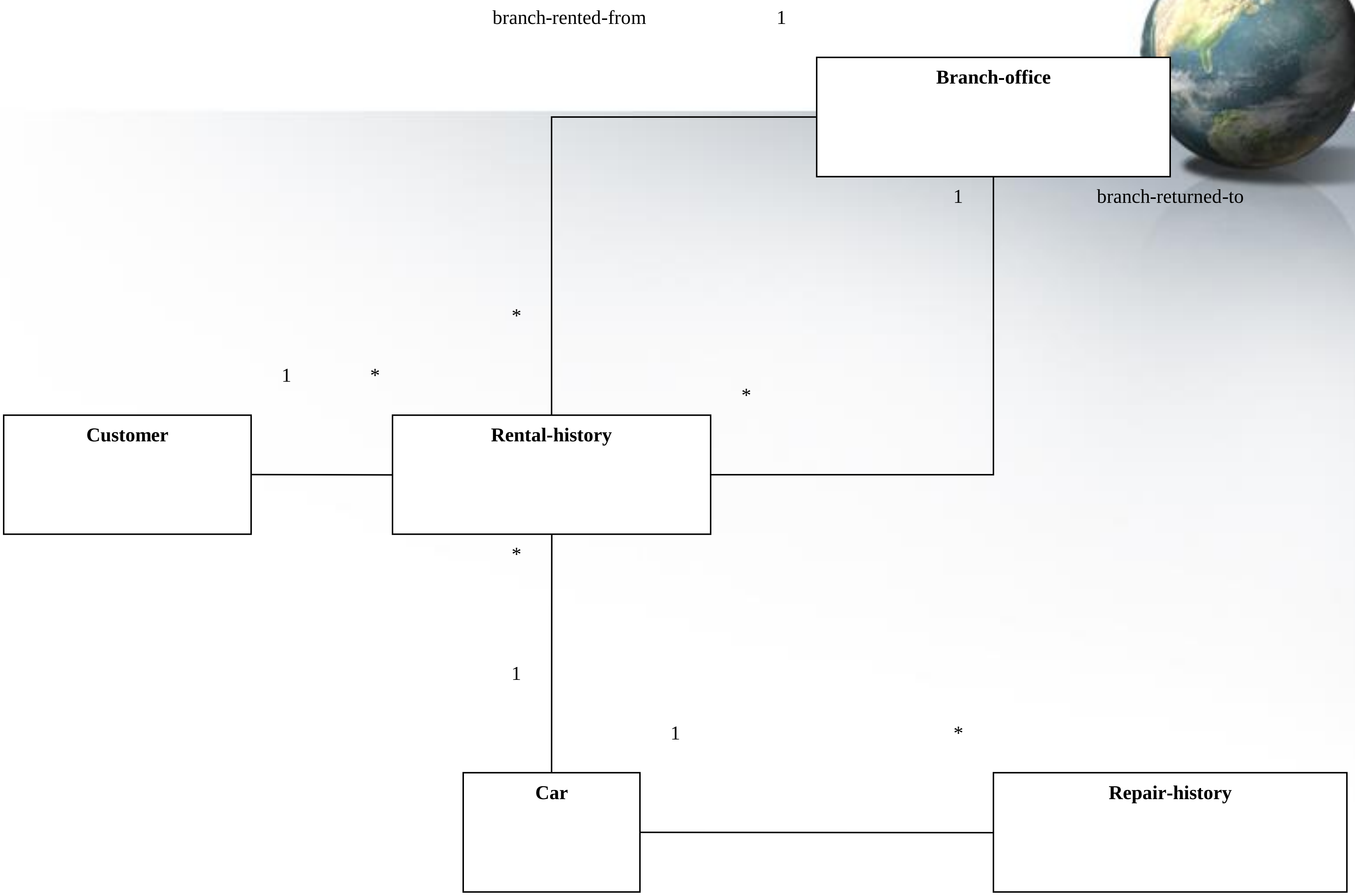
Student

*

Enrollment

term

- Draw UML class diagrams for a car rental agency database (e.g. Hertz), keeping track of current rental location of each car, its current condition and history of repairs, and customer information for a local office, expected return date, return location, car status (ready, being-repaired, currently rented, being-cleaned). Select attributes from your intuition about the situation. Draw one diagram showing the relationships of the classes without the attributes listed. Then show each class individually with the attributes listed.



branch-rented-from

1

Branch-office

1

branch-returned-to

*

1

*

*

Customer

Rental-history

*

1

1

*

Car

Repair-history

Customer

customer-id
last-name
first-name
middle-initial
street-address
city
state
zip
drivers-license
state-of-license
credit-card-type
credit-card-number

Branch-office

branch-id
street-address
city
state
zip
manager-id
manager-last-name
manager-first-name
manager-middle-initial
manager-phone
manager-email
manager-fax

Rental-history

car-id
date-rented
time-rented
date-returned
time-returned
mileage-out
mileage-returned
customer-id
branch-id-rented-from
branch-id-returned-to
date-expected-return

Car

car-id
make
model
year
color
license-plate
license-state
status

Repair-history

car-id
date-to-shop
date-returned
problem-type
shop-employee-id

References and Bibliography

1. Software Engineering, Ian Sommerville, Pearson Education, Inc., publishing as Addison-Wesley.
2. Software Engineering-A practitioner’s approach, Roger S Pressman, Mcgraw-Hill
3. Software Engineering: Principles And Practice, Waman S Jawadekar, Tata McGraw-Hill Education Pvt. Ltd.
4. NPTEL Course Website

Acknowledgement

- The content of this document is taken from different papers, articles, books, blogs, forums, industry reports and web sources.We would like to thank all contributors for providing free access of all these learning materials.