OUTLINE

1 Object Technology: History and Applications
2 OOA, OOD, and OOP
3 Classes and Objects
4 Object Model Characteristics
5 Benefits of OO-Approach
OBJECT TECHNOLOGY

• A sound paradigm to economically produce enduring and resilient industrial-strength software systems

• Representative Applications
  – Missile Control Software
  – ERP packages for industrial plants
  – Traffic control systems: Airlines, railways
  – Telecommunications
  – Industrial Process Control
  – Simulation
  – Database-intensive applications such as banking, transportation, service sectors
  – Office automation and workflow management systems
  – Health-care systems
Analysis and Design

- **Analysis:**
  - A disciplined process for generating a set of faithful models for a problem, without consideration of a technical solution
  - Examine the **What** of a problem

- **Design:**
  - A disciplined process that will describe how an analysis model may be implemented within a technical environment
  - A design model addresses the **how** of a problem and provides enough detail so that the system can be implemented in a programming language
Analysis and Design Methods

• Method:
  – A disciplined process for generating a set of models to describe a software system using a well-defined notation

• Methodology:
  – A collection of methods applied across the software development life-cycle and unified by a distinctive approach

• Major Methods:
  – Top-Down Structured Design (based on algorithmic decomposition)
  – Data-Driven Design (based on input-output relationships)
  – OO-Design (based on objects)
OO-Technology: History

- 1960s - Abstraction ideas - Dijkstra
- 1967 - Simula 67
- 1970s - Parnas (information hiding); Liskov and Guttag (abstract data types); Hoare (monitors); Chen (ER approach); Minsky (theory of frames)
- 1972 - Smalltalk; later versions in 74, 76, 80
- 1980s - Object Pascal, Ada, Eiffel, Objective C, CLOS, C++
- Early 1990s - Booch, Rumbaugh, Jacobson developed their distinctive methodologies; CASE tools
- Mid 1990s - UML, design patterns, OO-frameworks, distributed objects, unified process, components, etc.
• **Object Oriented Programming**
  A method of implementation in which programs are organized as cooperative collections of objects (instances of classes), and whose classes are all members of a hierarchy of classes united via inheritance relationships.

• **Object Oriented Design**
  A method of design encompassing the process of object-oriented decomposition and a notation for depicting logical and physical as well as static and dynamic models of the system.

• **Object Oriented Analysis**
  A method of analysis that examines requirements from the perspective of classes and objects.
**OOA, OOD, and OOP**

- **OOA** provides a set of models from which one can carry out **OOD**
- **OOD** gives blueprints for implementing a system using **OOP** methods
- **OOP** leads to development of OO programs that implement the software system
CLASSES AND OBJECTS

• An object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain
  – An object has state, behavior, identity

• The structure and behavior of similar objects are defined in their class.
  – A class is the same as an Abstract Data Type
  – An object is an instance of a class.

• Examples of Objects:
  – Banking System: Bank, Customer, Account, SB Account, FD Account, Manager, Employee, Transaction, Cash
  – Simulation: Resource, computer, disk, channel, random number, scheduling policy, job, etc.
**ABSTRACT DATA TYPE**

- An ADT is a set of well-defined elements, together with a collection of well-defined operations
  - Operands and results are not only instances of the ADT but could be other types of operands or instances of other ADTs
  - At least one of the operands or the result is of the ADT type in question

- An ADT Implementation is a translation into statements of a programming language:
  - the declarations that define the ADT elements (attributes) and the ADT operations (method)
  - An ADT implementation chooses a *Data Structure* to represent the ADT
Example: List ADT

- A list is a collection of elements of a particular type with representative operations such as:
  1. insert \((x, p, L)\)
  2. locate \((x, L)\)
  3. retrieve \((p, L)\)
  4. delete \((p, L)\)
  5. next \((p, L)\)
  6. prev \((p, L)\)
  7. makenull \((L)\)
  8. first \((L)\)
  9. printlist \((L)\)
A Program Using List ADT

Problem: To eliminate all duplicates from a list

```plaintext
{ 
    p = first (L);
    while (p != end(L)) {
        q = next (p, L);
        while (q != end(L)) {
            if (same (retrieve (p, L),
                     retrieve (q, L))))
                delete (q, L);
            else
                q = next (q, L);
        }
    p = next (p, L);
}
```

Data Structure Independent
Characteristics of an Object Model

- **Major Characteristics**
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy

- **Minor Characteristics**
  - Typing
  - Concurrency
  - Persistence
ABSTRACTION

Essential characteristics of an object that distinguish it from all other objects.

• provides crisply defined conceptual boundaries for an object

• focuses on the outside view of an object

• serves to separate object’s essential behavior from its implementation

• is a fundamental way to cope with complexity

Central problem: To decide upon the right set of abstractions for a given domain
ENCAPSULATION

Process of compartmentalizing the elements of an abstraction that constitute its structure and behavior.

- Complementary to abstraction:
  - abstraction focuses on observable behavior
  - encapsulation focuses on the implementation that gives rise to this behavior

- achieved through information hiding which is the process of hiding those aspects of the object behavior that are not relevant to the outside environment

- serves to separate interface of an abstraction and implementation of an abstraction
HIERARCHY

Represents a ranking or ordering of abstractions
Hierarchy can be:

1. **IS A** hierarchy (Inheritance, Generalization, Specialization)

2. **PART OF** hierarchy (Aggregation, composition)
   - Enables reuse
   - Leads to a more natural model of the problem
   - Inheritance hierarchy enables polymorphism (ability to hide many different implementations behind a single interface)
   - Class inheritance versus object composition is a very critical tradeoff in object oriented designs
Inheritance Relationships

Security
  presentValue()
  history()

CashAccount
  interestRate
  presentValue()

Stock
  presentValue()

Bond
  presentValue()

Property
  assessments
  presentValue()

SmallCapStock

LargeCapStock

Inheritance Relationships
Generalization

Shape

Origin

move()
resize()
display()

Rectangle
  corner : Point

Square

Circle
  radius : Float

Polygon
  points : List
display
**BENEFITS OF OO-APPROACH**

1. Leads to more comprehensive analysis and better decompositions

2. Modeling based on the real world
   - Easier to understand and maintain
   - More durable and flexible
   - More accurate description of real-world processes

3. More thorough and more scientific large-scale design
   - Modularity and componentization

4. Reusable and extensible software

5. Provides a more natural approach to iterative software development

**An ideal approach for large scale industrial strength software development**
Y. Narahari
Computer Science and Automation
Indian Institute of Science
Bangalore - 560 012
OUTLINE

1 Software Processes
2 OOAD Methodologies
3 OMT (Object Modeling Technique)
4 Rational Unified Process
Software Process

• Structured set of activities to transform user’s requirements into a robust and scalable software system

• Defines who is doing what, when, and how towards building a software product

• An effective process:
  – provides guidelines for efficient development of quality software
  – reduces risk
  – increases predictability
  – promotes a common vision and culture

• An effective process should be capable of evolving through:
  – technologies
  – tools
  – people
  – organizational patterns
**OOAD Methodologies**

- OOAD methodologies proliferated during the late 1980’s and early 1990’s
- Prominent Methodologies:
  - Booch methodology
  - Object Modeling Technique (OMT)
  - Objectory (Object Factory)
  - Fusion Methodology
- Each methodology recommended its own modeling notation
OMT

- Developed by James Rumbaugh and co-workers
- Uses a notation called OMT
- OOAD in OMT has three main phases:
  1. **Analysis**: Develop a precise, concise, accurate, understandable, and correct model of the system
     - Object Modeling
     - Dynamic Modeling
     - Functional Modeling
  2. **System Design**: Create a high-level design or a system architecture
  3. **Object Design**: Implement classes and relationships using data structures and algorithms
Rational Unified Process (RUP)

- UML gives a standard way to visualize, specify, construct, document, and communicate the artifacts of a software-intensive system
- UML has to be used in the context of an end-to-end software process
- RUP is the recommended software process when UML is used as the modeling language
- RUP has evolved over a period of thirty years
  - Ericsson Approach (Late 1960s)
Unified Process: A Cosmic View

- A generic process framework that can be customized for:
  - a variety of software systems
  - different application areas
  - different types of organizations
  - different project sizes

- Distinctive Aspects:
  - Use-Case Driven
  - Architecture-Centric
  - Iterative and Incremental
  - Component-based

- Four Phase Process:
  - Inception
  - Elaboration
  - Construction
  - Transition

UML is an Integral Part of the Process
Workflows and Phases

- Core Workflows
  - Requirements Capture
  - Analysis
  - Design
  - Implementation
  - Test

- Four Phase Process:
  - Inception
  - Elaboration
  - Construction
  - Transition

UML is an Integral Part of the Unified Process
The Software Development Life Cycle

The Software Development Life Cycle is divided into four main phases:

1. **Inception**
2. **Elaboration**
3. **Construction**
4. **Transition**

**Process Workflows**
- Business Modeling
- Requirements
- Analysis and Design
- Implementation
- Test
- Deployment

**Supporting Workflows**
- Configuration and Change Management
- Project Management
- Environment

The diagram illustrates the flow of activities across these phases with iteration markers such as iter #1, iter #2, iter #n, and additional markers like iter #n+1, iter #n+2, iter #m, and iter #m+1.
Use Case Driven

- Use Cases:
  - represent interactions between the user and system: *What does the system do for each user?*
  - capture functional requirements
- Developers create a series of analysis, design, and implementation models that realize the use cases
- Use cases drive the development process and bind the different phases of the process
Architecture Centric

- A software architecture is an intermediate level abstraction that connects the characteristics of systems users need to the characteristics of the systems software engineers can build.

- Software architecture embodies the most significant static and dynamic aspects of the systems.

- Architecture can be evolved from key use cases.

- Use cases represent the *function* and the architecture represents the *form* of a software product.
Iterative and Incremental

- A software development project is sliced into mini-projects; each mini-project is an iteration that results in an increment
- Iterations refers to steps in the work-flow
- Increments refer to growth in the product
- Each iteration deals with a group of use cases that together extend the usability of the product:
  - Developers identify and specify relevant use cases in the iteration
  - Create a design using chosen architecture
  - Implement the design in components
  - Verify that the components satisfy the use cases
Component Based

- An interface is a collection of operations that are used to specify the services of a class or component
- A physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces
- Define crisp abstractions with well-defined interfaces, enabling to easily replace older components with newer, compatible ones
- Three types of components:
  - Deployment components: components necessary and sufficient to form an executable system
  - Work Product Components: source code files and data files from which deployment components are created
  - Execution Components: created as a consequence of an executing system
Inception

- Develop a good idea and vision of the end-product
- Present a business case for the project:
  - What is the system primarily going to do for its users?
  - What could an architecture look like?
  - What is the plan and what will it cost?
- Most important risks are identified
- The Elaboration Phase is planned in detail
- Broad estimation of resources and costs is done
Elaboration

- Specify in detail (most of the) use cases
- Design the system architecture
- Do a detailed risk analysis
- Realize the most important use cases to provide an architecture baseline
- Draw up a detailed plan of activities for the Construction phase (Set up a series of iterations for construction and assign use cases to iterations)
- Do an exhaustive estimation of resources and costs
Construction

- Build the system in a series of iterations
- Each iteration involves analysis, design, coding, testing, integration for the assigned use cases
- Incremental in function; builds on the use cases developed in the previous iteration
- Iterative: Rewrite existing code to make it more flexible

Transition

- Optimization and fine-tuning of the product
- Beta-testing
- Does not involve any development to add major functionality
UML Overview

Y. Narahari

Computer Science and Automation
Indian Institute of Science
Bangalore - 560 012
1 Modeling Principles
2 UML Evolution
3 Bird’s Eye View
4 UML Things
5 UML Relationships
6 UML Diagrams
Why Model?

- A model is a simplification of reality
- We build models so that we can better understand the system we are developing
- Models have four objectives:
  1. Help us to visualize a system as it is or as we want it to be
  2. Permit us to specify the structure and behavior of a system
  3. Provide us a template that guides us in constructing a system
  4. Document the decisions we have made
Principles of Modeling

1. The choice of what models to create has a profound influence on how a problem is approached and how a solution is evolved.

2. Every model may be expressed at different levels of precision.

3. The best models are connected to reality.

4. No single model is sufficient. Every non-trivial system is best approached through a small set of nearly independent models.
Unified Modeling Language

- OMG standard notation for object oriented modeling
- UML is a modeling language
- UML is a notation, not a methodology
- Developed jointly by: Booch, Rumbaugh, and Jacobson
- Version 0.8 in 1995; Version 1.0 in 1997; Version 1.2 was adopted as an OMG standard in September 1998
- Details can be downloaded from www.rational.com
- UML can be used in:
  - conceptual modeling
  - visualization
  - specification
  - constructing
  - documenting
**UML: A Bird’s Eye View**

- UML has three building blocks:
  - Things
  - Relationships
  - Diagrams

- **Things**
  - Structural
  - Behavioral
  - Grouping
  - Annotational

- **Relationships**
  - Dependency
  - Association
  - Generalization
  - Realization
UML: A Bird’s Eye View

- **Diagrams**
  - Class Diagram
  - Object Diagram
  - Use Case Diagram
  - Component Diagram
  - Deployment Diagram
  - Sequence Diagram
  - Collaboration Diagram
  - Statechart Diagram
  - Activity Diagram

- **Extensibility Mechanisms**
  - Stereotypes
  - Tagged Values
  - Constraints
UML: Things

- **Structural Things**: Static parts of a model, either conceptual or physical.
  - Classes
  - Interfaces
  - Collaborations
  - Use Cases
  - Active classes
  - Components
  - Nodes

- **Behavioral Things**: Dynamic parts of UML models
  - Interaction
  - State machine

- **Grouping Things**: Organizational parts of UML models, for example, packages

- **Annotational Things**: Explanatory parts of UML models, for example, notes.
**UML: Relationships**

- **Dependency**: Semantic relationship between two things in which a change to one thing may affect the semantics of the other.

- **Association**: Structural relationship that describes a connection among objects.
  - Can be binary, ternary, etc.
  - Aggregation and composition.

- **Generalization**: Relationship in which specialized objects (children) share the structure and behavior of the parent objects.

- **Realization**: Semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out.
  - interfaces and classes
  - interfaces and components
  - use cases and collaborations
Structural Relationships

School

1..*

has

1

1..*

Department

0..1

assignedTo

1..*

1..*

Student

attends

1..*

Course

teaches

1..*

Instructor

0..1 chairperson

structural relationships
Architectural Views

- **Use Case View**: use case diagrams; activity diagrams
- **Design View**: class diagrams, interaction diagrams, state chart diagrams
- **Process View**: class diagrams, interaction diagrams, activity diagrams
- **Implementation View**: component diagrams
- **Deployment View**: deployment diagrams
Forward and Reverse Engineering

- **Forward Engineering** is the process of transforming a model to an implementation language

- **Reverse Engineering** is the process of transforming code into a model through a mapping from a specific implementation language

- (UML) models are semantically richer than code

- Forward engineering results in a loss of information; thus reverse engineering cannot completely recreate a model

- UML diagrams can be used in forward engineering and to some extent also in reverse engineering
UML Diagrams

Y. Narahari

Computer Science and Automation

Indian Institute of Science
Bangalore - 560 012

53
OUTLINE

1. Class Diagrams
2. Object Diagrams
3. Use Case Diagrams
4. Component Diagrams
5. Deployment Diagrams
6. Sequence Diagrams
7. Collaboration Diagrams
8. Activity Diagrams
9. State Chart Diagrams
A Summary of UML Notation

Class

Class Name

Class Name
attribute : Type = initialValue
operation(arg list) : return type

Generalization

Supertype

Subtype 1

Subtype 2

Constraint
{description of constraint}

Stereotype
"stereotype name"

Note
some useful text

Object
object name : Class Name

Association

Class A
role A

Class B
role B

Multiplicities

1
Class
exactly one

* Class
many (zero or more)

0..1 Class
optional (zero or one)

Class
numerically specified

Class
aggregation

Class
composition

Class
{ordered} * ordered role

Qualified Association

Class qualifier

Navigability

Source
role name

Target

Dependency

Class A

Class B
Class Diagrams

- Describes types of objects in the system and various kinds of relationships between the objects, such as:
  - Association
  - Aggregation
  - Generalization

- Interpretation of class diagrams can proceed along three perspectives:
  - **Conceptual**: Only the concepts in the domain are modeled here
  - **Specification**: Look into interfaces but not implementation
  - **Implementation**
Modeling the Vocabulary of a System

<table>
<thead>
<tr>
<th>Customer</th>
<th>Order</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>name address phone birthDate</td>
<td>item quantity</td>
<td>id name price location</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Shipment</th>
</tr>
</thead>
<tbody>
<tr>
<td>actions</td>
<td>Responsibilities</td>
</tr>
<tr>
<td>commit() rollBack() wasSuccessful()</td>
<td>- maintain the information regarding products shipped against an order</td>
</tr>
<tr>
<td></td>
<td>- track the status and location of the shipped products</td>
</tr>
</tbody>
</table>

Modeling the Vocabulary of a System
Qualified Association

- Improves semantic accuracy
- like a tennary association
- reduces effective multiplicity of an association
A Class Diagram
Modeling a Schema
An Object Diagram

\[
c : \text{Company}
\]

\[
d1 : \text{Department} \\
\text{name} = "Sales"
\]

\[
d2 : \text{Department} \\
\text{name} = "R&D"
\]

\[
d3 : \text{Department} \\
\text{name} = "US Sales"
\]

\[
p : \text{Person} \\
\text{name} = "Erin" \\
\text{employeeID} = 4362 \\
\text{title} = "VP of Sales"
\]

\[
\text{ContactInformation} \\
\text{address} = "1472 Miller St"
\]
Common Mechanisms

1. Specifications
   - textual statement of the syntax and semantics of each building block (which supplements the graphical notation of the building block)

2. Adornments
   - detail from an element’s specification added to its basic graphical notation

3. Common Divisions
   - class-object dichotomy
   - interface-implementation dichotomy

4. Extensibility Mechanisms
   - Stereotypes
   - Tagged Values
   - Constraints
Common Mechanisms

Transaction

+ execute()
+ rollback()
# priority()
- timestamp()

Adornments

Customer

name
address
phone

Jan : Customer

: Customer

Elyse

Classes And Objects

Interfaces And Implementations

EventQueue

(version = 3.2
author = egb)

"exception"
Overflow

add()
remove()
flush()

Extensibility Mechanisms
Use Cases

- Specify the behavior of a system or a part of a system and describe a set of actions that a system performs to yield an observable result of value to an actor

- Use cases are important to:
  - Capture intended behavior of the system being developed, without having to specify how that behavior is implemented
  - Provide a way for developers to come to a common understanding with the systems end users and domain experts
  - Help validate the architecture and verify the system as it evolves during development
  - Use cases are realized by collaborations whose elements work together to carry out each use case

- Rational Unified Process is use-case driven
Why Use Cases?

- They offer a systematic and intuitive means of capturing **functional requirements**, with a focus on value added to the user.
- They drive the whole development process:
  - Analysis, Design, and Testing are performed starting from the use cases
  - Design and Testing can be planned and coordinated in terms of use cases
- They help devise a suitable system architecture
- They facilitate an iterative and incremental process
Use Case Relationships

Place order
Extension points
set priority

Place rush order
(set priority)

Check password

Retinal scan

Generalization, Include, and Extend

Track order

include relationship

Validate user
Use Case Diagrams

- A use-case diagram contains:
  - Use cases
  - Actors
  - Dependency, generalization, and association relationships
  - Notes and constraints

- Use case diagrams provide a static use case view

- Use case diagrams can be used to model:
  - context of a system
  - requirements of a system
Customer Credit Card Validation System

- Perform card transaction
- Process customer bill
- Reconcile transactions
- Manage customer account

Customer

- Retail institution
- Sponsoring financial institution

Individudal customer

Corporate customer

Modeling the Context of a System
Modeling Context of a System

Credit Card Validation System

- Perform card transaction
- Process customer bill
- Reconcile transactions
- Manage customer account

- Customer
  - Individual customer
  - Corporate customer
- Retail institution
- Sponsoring financial institution

Modeling the Context of a System
Modeling the Requirements of a System

Customer

Retail institution

Sponsoring financial institution

Credit Card Validation System

- Perform card transaction
- Report on account status
- Process customer bill
- Detect card fraud
- Reconcile transactions
- Manage network outage
- Manage customer account

Modeling the Requirements of a System
**Interfaces**

- An interface is a collection of operations that are used to specify a service of a class or a component

- **Export Interface**: an interface that a component realizes
  - A component may provide many export interfaces

- **Import Interface**: an interface that a component uses
  - A component may conform to many import interfaces
Components

- A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces
  - model: executables, libraries, tables, files, documents, APIs, source code, etc.
  - represent physical packaging of otherwise logical elements such as classes, interfaces, and collaborations
  - higher level of abstraction than classes
- Define crisp abstractions with well-defined interfaces, enabling to easily replace older components with newer, compatible ones
- Components only have operations which are reachable only through their interfaces
Types of Components

- **Deployment components**: components necessary and sufficient to form an executable system, such as dynamic libraries and executables; Examples - COM, CORBA, Enterprise Java Beans

- **Work Product Components**: source code files and data files from which deployment components are created

- **Execution Components**: created as a consequence of an executing system, for example a COM+ object that is instantiated from a DLL
Component Diagrams

- Shows the organization and dependencies among a set of components
- Comprises:
  - Components
  - Interfaces
  - Dependency, generalization, association, and realization relationships
  - notes and constraints
- Models static implementation view of a system
- Can be used:
  - to model source code
  - to model executable releases
  - to model physical databases
  - to model adaptable systems
• A node is a physical element that exists at run time and represents a computational resource and usually has memory and processing capability
  – A processor is a node that has processing capability and therefore can execute a component
  – A device is a node with no processing capability and provides only an interface to the real world

• Nodes represent physical deployment of components and are responsible for executing components

• A component may be deployed on multiple nodes; also a component need not be statically located in a given node
Deployment Diagrams

- Shows the configuration of run time processing nodes and the components that live on them
- Models the static deployment view of the system (model the topology of the hardware on which the system executes)
- Comprises:
  - Nodes and components
  - Dependency and association relationships
  - Notes and constraints
- Not necessary in some situations
- Very useful in modeling: embedded systems, client/server systems, and fully distributed systems
**Interactions**

- **An interaction** is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose

- **Link**: A semantic connection among objects; an instance of an association

- **Message**: A specification of a communication between objects that conveys information with the expectation that activity will ensue; Messages could be in the form of
  - call
  - return
  - send
  - create
  - destroy
Interaction Diagrams

• Shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them

• Involves modeling concrete instances of classes, interfaces, components, and nodes, along with the messages that are dispatched among them, all in the context of a scenario

• Interaction diagrams are of two types:
  – Sequence diagrams
  – Collaboration diagrams

• Sequence diagram shows time ordering of messages (flow of control by time ordering)

• Collaboration diagram shows structural organization of objects that send and receive messages (flow of control by organization)

• semantically equivalent
Flow of Control by Time

p : StockQuotePublisher

attach(s1)

s1 : StockQuoteSubscriber

notify()

update()

getState()

attach(s2)

update()

getState()

s2 : StockQuoteSubscriber

Flow of Control by Time
Flow of Control by Organization

p : StockQuotePublisher

3 : notify()
4 : update()

s1 : StockQuoteSubscriber

1 : attach(s1)
6 : getState()

2 : attach(s1)
7 : getState()

s2 : StockQuoteSubscriber

5 : update()
Interaction at a High Level of Abstraction

Interaction Diagram at a High Level of Abstraction
Interaction at a Lower Level

Interaction Diagram at a Low Level of Abstraction

See Credit Failure for a variation of this scenario
Events and Signals

- An **event** is the specification of a significant occurrence that has a location in time and space; can trigger a state transition

- A **signal** is a kind of event that represents the specification of an asynchronous stimulus communicated between instances

- Kinds of Events:
  - Signal: a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another
    - Exceptions are a common kind of internal signals
  - Call: represents the dispatch of an operation (synchronous)
  - Time event: represents passage of time
  - Change event: represents a change in state or the satisfaction of a condition
Exceptions

- Exceptions are kinds of signals (internal)
- Modeled as stereotyped classes
- Specify the kinds of exceptions that an object may throw through its operations
- Exceptions can be arranged in a hierarchy
- An important part of visualizing, specifying, and documenting the
- An important part of visualizing, specifying, and documenting the behavior of a class or an interface
State Machines

- Specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.

- A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.

- Commonly used to specify the behavior of objects or instances that must respond to asynchronous stimulus or whose current behavior depends on the past.

- Two kinds of state machines:
  - Activity Diagrams: emphasize flow of control from activity to activity.
  - Statechart Diagrams: emphasize potential states of the objects and transitions among those states.
**States and Transitions**

- States have the following parts:
  - Name
  - Entry/exit actions
  - Internal transitions (transitions that do not cause a change in state)
  - Substates
    - sequential substates
    - concurrent substates
  - Deferred events

- A transition has five parts:
  - Source state
  - Event trigger
  - Guard condition
  - Action
  - Target state
Statechart Diagrams

- Model the event-ordered behavior of reactive objects
- Provide a most natural way to visualize, specify, construct, and document the behavior of interesting, non-trivial objects and instances
- Can be attached to classes, use cases, or entire systems
- A natural tool to model reactive (event-driven) objects
  - objects whose behavior is best characterized by response to events dispatched from outside their context
- Can be used for constructing executable systems through forward and reverse engineering
Active

Heating ready / turnOn()

Activating

tooHot(desiredTemp)

tooCold(desiredTemp)

Heating

ready / turnOn()

Active

tooHot(desiredTemp)

tooCold(desiredTemp)

ShutDown

State Machines
A State Chart Diagram

Statechart Diagram
Concurrent Substates

Idle

maintain

join

Maintenance

Testing

Testing devices

Self diagnosis

Commanding

Waiting

KeyPress

Command

[Continue]

[not continue]

Cuncurrent Substances
Modeling the Lifetime of an Object

Modeling the Lifetime of An Object
Activity Diagrams

• Shows flow of control from activity to activity and models:
  – dynamics of a society of objects through sequential and concurrent steps in a process
  – flow of an object as it moves from one state to another
  – flow of control of an operation

• Activity: an ongoing non-atomic execution within a state machine: an activity results in some action

• An action is made up of executable atomic computations that result in a change of the state of system or the return of a value
  – calling another operation
  – sending a signal
  – creating or destroying an object
  – a pure computation
An Activity Diagram

- Select site
  - Commission architect
    - Develop plan
      - Bid plan
        - [not accepted]
        - [else]
          - Do site work
          - Do trade work()

- Finish construction
  - CertificateOfOccupancy [completed]

Activity Diagrams
Activity Diagrams

- An activity diagram is a state machine in which
  1. almost all states are activity states
  2. almost all transitions are triggered by completion of activities in the source state

- Important features that can be modeled:
  - Sequential execution
  - Concurrency (forking)
  - Synchronization (joining)
  - Swimlanes
  - Object flow

- Primary uses:
  - Modeling of workflows and business processes
  - Model an operation (like a flowchart)
  - Forward and reverse engineering
Object Flow in an Activity Diagram

Customer
- Select site
  - Process order
    - 0: Order
      [in progress]
  - Receive order
  - Pay bill
    - b: Bill
      [unpaid]
  - Close order
    - b: Bill
      [paid]

Sales
- Process order
  - 0: Order
    [filled]

Warehouse
- Pull materials
- Ship order
  - 0: Order
    [filled]
Processes and Threads

- **Active Object**: an object that owns a process or thread and can initiate control activity

- **Active Class**: a class whose instances are active objects

- **Process**: a heavyweight flow that can execute concurrently with other processes
  - runs in an independent address space known to the operating system

- **Thread**: a lightweight flow that can execute concurrently with other threads in the same process
  - most often, hidden inside a heavier-weight process and runs inside the address space of an enclosing process

- An active class represents a process or thread that is the root of an independent flow of control
• **Communication**: 
  - Passive object to passive object
  - Active object to active object 
    (inter-process communication)
    * synchronous (rendezvous)
    * asynchronous (mailbox)
  - Active object to passive object
  - Passive object to active object

• **Synchronization**: This problem arises when two or more flows of control are in the same object at the same time
  - Sequential approach
  - Guarded semantics
  - Concurrent approach
Time and Space

- UML provides modeling infrastructure for real-time and distributed systems:
  - *Timing Mark*: a denotation for the time at which an event occurs
  - *Time Expression*: an expression that evaluates to an absolute or relative value of time
  - *Timing Constraint*: a semantic statement about the relative or absolute value of time
  - *Location*: placement of a component on a node

- A well-structured UML model:
  - exposes all and only those time and space properties that capture the desired behavior of the system
  - centralizes the use of those properties so that they are easy to find and easy to modify
A Case Study

Y. Narahari

Computer Science and Automation

Indian Institute of Science

Bangalore - 560 012
A Requirements Specification

- The library lends books and magazines to borrowers, who are registered in the system.
- The library handles purchase of new titles. Popular titles are bought in multiple copies. Old books and magazines are often removed.
- The librarian interacts with the customers
- A borrower can reserve a book or magazine that is not currently available in the library and will get a notification when the book is returned or purchased. The reservation is canceled in appropriate conditions.
- The library can create, update, and delete information about titles, borrowers, loans, and reservations
- The system can run on popular technical environments (Unix, Windows, OS/2, etc.) and has a modern GUI.
Different Steps in the Process

- **Analysis**: capture and describe all the requirements of the system, and to create a model that defines the key domain classes
  - Requirements Analysis: Use-case Diagrams
  - Domain Analysis: Class Diagrams, Interaction Diagrams, State Machines

- **Design**: Specify a working solution that can be transformed into programming code
  - Architectural Design
  - Detailed Design

- **Implementation**: Develop or generate the code

- **Test and Deployment**
• Use cases describe what the library system provides in terms of functionality

• Actors: Librarians, Borrowers

• Use cases:
  – Lend Item
  – Return Item
  – Make Reservation
  – Remove Reservation
  – Add Title
  – Update or Remove Title
  – Add Item
  – Remove Item
  – Add Borrower
  – Update or Remove Borrower
  – Maintenance

• Each use-case is documented with text
Use Case Diagram

Library System

- Add Title
- Remove or Update Title
- Add Item
- Remove Item
- Lend Item
- Return of Item
- Remove Reservation
- Make Reservation
- Add Borrower
- Remove or update Borrower

Borrower
Librarian
Librarian
Use Case: Lending Item

1. If the borrower has no reservation:
   - A title is identified
   - An available item of the title is identified
   - The borrower is identified
   - The library lends the item
   - A new loan is registered

2. If the borrower has a reservation:
   - The borrower is identified
   - The title is identified
   - An available item of the title is identified
   - The library lends the item
   - The reservation is removed
Domain Analysis

- Identify the classes from the specifications and the use cases
- Domain classes: BorrowerInformation, Title, Book Title, Magazine Title, Item, reservation, and Loan
- The stereotype Business Object specifies that the object is a key domain class and should be stored persistently in the system
- Some broad issues regarding GUI may have to be considered during the analysis stage
Domain Class Diagram

```
"Business Object"
Item
- id : Integer
+ find on title ()
+ find on id ()
+ find on reservation ()
create ()
destroy ()

"Business Object"
Title
(abstract)
- name : String
- /number of reservation
+ find ()
create ()
destroy ()

"Business Object"
Book Title
- lending time : Days = 30
create ()
destroy ()

"Business Object"
Magazine Title
- lending time : Days = 30
create ()
destroy ()

"Business Object"
Loan
- date: Date = current date
create ()
destroy ()

"Business Object"
Borrower Information
- name : String
- address : String
+ find ()
create ()
destroy ()

"Business Object"
Reservation
- date: Date = current date
+ find ()
create ()
destroy ()

may be loaded in a
refers to
0..1

0..*

has

has

{ ordered }

may be reserved in a
refers to

create ()
destroy ()
```

110
Not Reserved

entry : Number of reservations := 0

Reserved

Title reservation
/ Number of reservations ++

Reservation removed
[Number of reservations = 1]
/ Number of reservations --

Title reservation
/ Number of reservations ++

Reservation removed
[Number of reservations = 1]
/ Number of reservations --
Sequence Diagram for Lend Item Use Case

1: find title ()
2: find (String)
3: find item ()
4: find on title (Title)
5: identity borrower ()
6: find (String)
7: create (Borrower information, item)
Architecture Design

- High level design where subsystems (packages) are defined, including dependencies and communication mechanisms between between the packages
- A well-designed architecture is the foundation for an extensible and changeable system
- Packages in the Library System:
  - User Interface Package: based on the Java AWT package
  - Business Objects Package: includes domain classes
  - Database Package: provides persistence to business objects
  - Utility Package: services that are used in other packages
Architectures of Subsystems

UI Package

Business Objects Package

Utility Package

Database Package
Detailed Design

- Describe the new technical classes; expand and detail the descriptions of business object classes; (through more detailed UML diagrams)

- **Database Package:**
  - *Persistent* is a class that all classes that need persistent objects must inherit

- **Utility Package:**
  - *ObjId* is a class whose objects are used to refer to any persistent object in the system; used by all packages in the system

- **Business Objects Package:**
  - Class descriptions are detailed
  - Diagrams are refined further taking into account design-level details

- **User Interface Package**
Design State Diagram for Title Class

- **Not Reserved**
  - Vector reservations
  - entry: reservations.clear()

- **Reserved**
  - Vector reservations

- **Reservations.size() == 0**
  - Reservations.remove(R)

- **Reservations.size() > 0**
  - addReservation(R)
  - /reservations.add(R)

- removeReservation(R)
  - [reservations.size() == 1]
  - /reservations.remove(R)

- removeReservation(R)
  - [reservations.size() > 1]
  - /reservations.remove(R)

- addReservation(R)
  - /reservations.add(R)
Information about the title is entered into the edit fields in the window.

Check if title or ISBN already exists.

Create title and item(s); store them.
Design Level Collaboration Diagram

1: TitleFrame ()
2: addButton_Clicked ()

:Librarian

2.1: findOnName(String)
2.2: findOnISBN(String)
2.3: Title(String,String,String,int)
2.6: addItem(ObjId)
2.7: store ()

2: addButton_Clicked ()

2.4: Item(ObjId,int)
2.5: store ()

:TitleFrame

:Title {new}

Item {new}
User Interface Design

- Windows for functions such as lending and returning items, and making reservations of titles
- Windows for viewing information in the system, about titles and borrowers
- Windows for maintenance functions such as, adding, updating, and removing
- User Interface classes:
  - MainWindow
  - LendingItemFrame
  - CancelReservationFrame
  - ReturnItemFrame
  - ReservationFrame