

Object-Oriented Analysis and Design (OOAD)

Outline

- Object-Oriented (OO) System
- OO Software Development
- OOA vs. OOD vs. OOP
- Types of Classes
- Design Principles
- Assessing Design Quality
- OO Design Process



Object-Oriented System

- An object-oriented system consists of a set of interacting or collaborating objects
- Terminology
 - Objects
 - Classes
 - Services
 - Messages
- Objects
 - Entities in a software system that encapsulate state and behavior.
 - State is represented as a set of attributes/fields.
 - Behavior is represented as a set of operations/methods.

Nandigam

3



Object-Oriented System

- Classes
 - Type specification that describes all objects of a particular kind.
 - Template for creating objects of a specific type.
 - Includes declarations of all the attributes (fields) and operations (methods) associated with an object of that class.
- Services
 - Operations associated with an object that are visible to other objects (clients).
 - Operations that form the public interface of an object.
 - Implemented as public (or protected) methods in the class definition.

Nandigam

4

Object-Oriented System

■ Messages

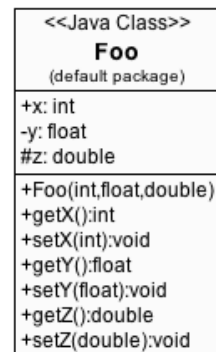
- Objects communicate by message passing.
- A message contains
 - The name of the service requested by the calling object
 - Copies of the information required to execute the service
- In practice, messages are often implemented by method calls
 - Name = method name
 - Information = parameter list

Nandigam

5

Object-Oriented System

- In UML, a class is represented as a named rectangle with three sections.
- The top section contains the class name.
- The object attributes are listed in the middle section.
- The operations are listed in the bottom section.
- Visibility information is often associated with the attributes and operations of a class.



Nandigam

6



OO Software Development

- OO software development consists of
 - Object-Oriented Analysis (OOA)
 - Object-Oriented Design (OOD)
 - Object-Oriented Programming (OOP)
- OOA
 - Concerned with developing an object-oriented model of the application domain.
 - Objects/entities and operations associated with the problem are identified.

Nandigam

7



OO Software Development

- OOD
 - Concerned with developing an object-oriented model of system to implement requirements.
 - Designer may add new objects to the ones already identified in the OOA phase to implement the solution.
- OOP
 - Concerned with realizing an OOD using an object-oriented programming language.
 - May have to use additional objects that are language specific (API) to implement the solution.

Nandigam

8

OOA vs. OOD vs. OOP

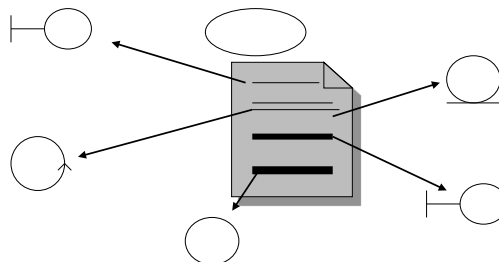
- Comparison of OOA, OOD, and OOP
 - See the OOA_OOD_OOP.pdf
- Also, see the following article
 - "A Road Map for OOA and OOD" article by Noel Rappin (previously at Georgia Tech University)
- These are available in the "Course Documents" folder on Blackboard!!

Nandigam

9

Types of Classes

- The behavior of a use case has to be distributed to a set of classes.



Nandigam

10




Types of Classes

- The technique of identifying classes that will part of a system uses three different perspectives of the system:
 - The boundary between the system and its actors.
 - The information the system uses.
 - The control logic of the system.

Nandigam

11



Types of Classes

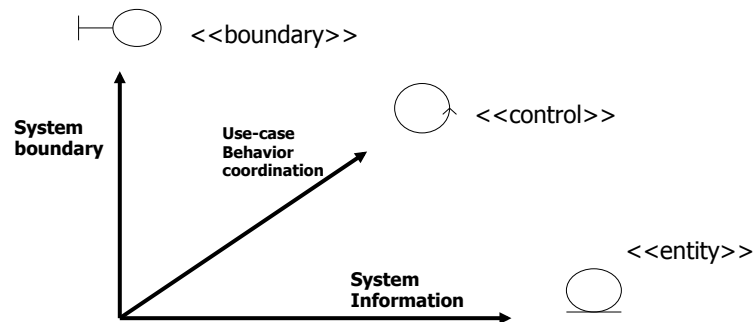
- Boundary class
 - Model interaction between the system and its actors.
 - Encapsulates connections between actors and use cases.
 - Lie on the periphery of a system or subsystem, but within it.
- Control class
 - Mostly performs behaviors associated with inner workings of use cases.
 - Responsible for the flow of the scenario of a use case.
- Entity class
 - Used to model information that is long-lived, passive, and often persistent.
 - Mostly corresponds to conceptual data entities in the problem domain.

Nandigam

12

Types of Classes

- UML notation for boundary, control, and entity classes
 - Stereotypes and icons

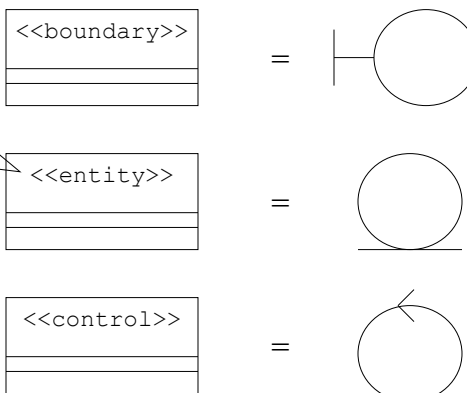


Nandigam

13

Types of Classes

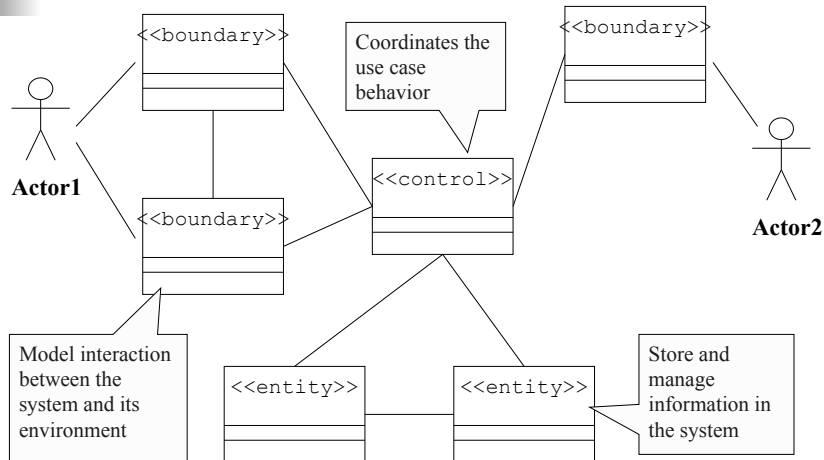
A stereotype defines a new model element in terms of another model element.



Nandigam

14

Types of Classes



Nandigam

15

Boundary Classes

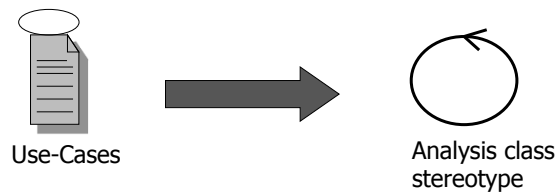
- Boundary classes model the interaction between the system's surroundings and its inner workings.
 - They insulate the system from changes in the environment
 - They are environment dependent
 - UI dependent
 - Communication protocol dependent
- Several types of boundary classes
 - User interfaces classes
 - System interface classes
 - Device interface classes
- At least one boundary class per actor/use-case pair

Nandigam

16

Control Classes

- Controls and coordinates the behavior of a use case
- Delegates the work of the use case to classes
- Control classes decouple boundary and entity classes
- Use case dependent
- One control class per use case (initially)



Nandigam

17

Entity Classes

- Entity classes model the key concepts in the problem domain
- Usually models information that is persistent
- Store and manage information in the system
- Contains the logic that solves the system problem
- Environment independent
- Can be used in multiple use cases
- Example entity classes:
 - Customer
 - Account
 - Reservation
 - Invoice

Nandigam

18

Entity Classes

- Techniques for finding entity classes
 - Noun Extraction
 - CRC (Class Responsibility Collaborator)
- Noun Extraction Method
 - Candidate entity classes are identified by examining the nouns and noun phrases in problem description, requirements document, use cases, problem domain information, and other documentation
 - Nouns identified may be
 - Objects
 - Description of an object's state (attributes)
 - Actors
 - None of the above
 - Filter nouns to identify candidate entity classes

Nandigam

19

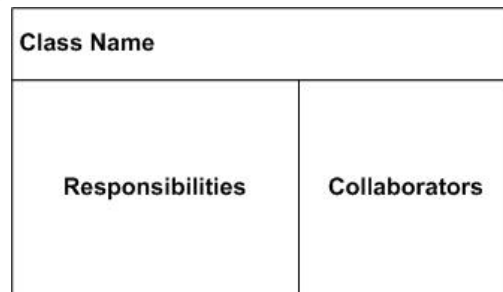
Entity Classes

- Class Responsibility Collaborator (CRC) Technique
 - **C**lass – the name of an OO class (a good descriptive noun)
 - **R**esponsibility – the things the OO class does (behavior responsibility)
 - **C**ollaboration – the relationship the class has with other classes
- Invented in 1989 by Kent Beck and Ward Cunningham.
- Uses standard index cards that have been divided into three sections (see next slide)
- A brainstorming technique that works with scenario walkthroughs to identify classes and/or stress-test a design.

Nandigam

20

Entity Classes



- Some also suggest writing the attributes or properties (knowledge responsibilities) of the class on the back of the card.

Nandigam

21

Using CRC Technique

- Step 1: Identify initial set of candidate classes (using noun extraction method)
- Step 2: Choose a coherent set of scenarios
 - Select a set of use cases which look as though they will touch a related set of object types. These provide the scenarios to walk through in the CRC Card session
- Step 3: Walk through the scenario naming cards and responsibilities
 - Walk through the handling of a scenario case pointing to or picking up the cards, naming their responsibilities and how they handle and delegate each request.
 - Add new cards as functions are needed, or reallocate the responsibilities of the cards already on the table.

Nandigam

22

Using CRC Technique

- Step 4: Vary the situations to stress test the cards
- Step 5: Add cards, push cards to the side, to let the design evolve
 - CRC cards permit several design alternatives to sit on the table at the same time. An unpopular initial design may turn out to be a popular later design, or perhaps the final design is a small alteration of an initially rejected design.
 - Do not throw away cards, but push them to the side, in case it turns out later they are useful.
- Step 6: Write down the key responsibility decisions and interactions.

Nandigam

23

Using CRC Technique

- Strengths of CRC Cards
 - When utilized by a team, the interactions among the members can highlight missing or incorrect members in a class, whether attributes or methods.
 - Relationships between classes can be clarified.
 - Effective means of verifying that the class diagrams are complete and correct.
- Weakness of CRC Cards
 - Generally not a good way of identifying entity classes if developers have little or no experience in the application domain.

Nandigam

24

Links Between Classes

- A class (boundary, control, or entity) can communicate with other classes of the same kind.
- A control class can communicate with other kinds.
- Entity and boundary classes should not directly communicate.

	Entity	Boundary	Control
Entity	X		X
Boundary			X
Control	X	X	X

Nandigam

25

Design Principles

- Encapsulation
- Information hiding
- Object Reuse
 - Inheritance vs. composition
 - Delegation with composition
 - Favor object composition over class inheritance
- Object Interfaces
 - Program to an interface, not an implementation

Nandigam

26

Encapsulation

- Encapsulation – *bundling of data with the operations that operate on that data.*
 - Operations → methods in OO PLs
 - Operations → functions and procedures in classical PLs
- Encapsulation is a language facility/construct.
- Encapsulation is not specific to just OO PLs.
- It can be achieved in PLs that provide a facility to bundle data and operations together.
 - In Ada, it is achieved using the *package* construct
 - In Modula, it is achieved using the *module* construct
 - In ML, it is achieved using the *abstype* construct
- Encapsulation is not same as information hiding.
 - In Java, you can have encapsulated data that is not hidden at all

Nandigam

27

Information Hiding

- Concept first introduced by David Parnas in 1972.
- Information hiding – *design principle that strives to shield client modules from the internal workings of a module.*
- Hiding data is not the full extent of information hiding.
- Parnas stressed hiding
 - “difficult design decisions or design decisions which are likely to change”
- Isolate clients from requiring intimate knowledge of the design to use a module, and from the effects of changing those decisions.
- Encapsulation facilitates, but does not guarantee, information hiding.

Nandigam

28

Information Hiding

- Rule 1 – *Don't expose data*
 - Make all data items private and use getters and setters
 - Makes defensive programming possible
- Rule 2 – *Don't expose the difference between stored data and derived data*
 - Whether a data value is stored or derived is a design decision best kept hidden.
 - Example – Use an accessor method named `speed()` or `getSpeed()` rather than `calculateSpeed()` to return an attribute called *speed* of an object.
- Rule 3 – *Don't expose a class's internal structure*
 - Clients should remain isolated from the design decisions driving the selection of internal class structure.
- Rule 4 – *Don't expose implementation details of a class*
"Remember encapsulation is not information hiding"

Nandigam

29

Object Reuse

- Two common techniques for reusing functionality in object-oriented system are
 - Class inheritance
 - Object composition
- Class inheritance
 - Reuse by sub classing
 - Often referred to as **white-box reuse**
- Object composition
 - New functionality is obtained by creating an object composed of other objects.
 - Also known as **black-box reuse**

Nandigam

30

Reuse with Inheritance

- A method of reuse in which new functionality is obtained by extending the implementation of an existing object.
- The generalization class (superclass) explicitly captures the common attributes and methods.
- The specialization class (subclass) extends the implementation with additional attributes and methods.
- Advantages:
 - New implementation is easy, since most of it is inherited
 - Easy to modify or extend the implementation being reused
- Disadvantages:
 - Breaks encapsulation, since it exposes a subclass to implementation details of its superclass.
 - "White-box" reuse, since internal details of superclasses are often visible to subclasses
 - Subclasses may have to be changed if the implementation of the superclass changes.
 - Implementations inherited from superclasses can not be changed at runtime (class inheritance is defined statically at compile-time)

Nandigam

31

Reuse with Composition

- New functionality is obtained by delegating functionality to one of the objects being composed.
- Object composition is defined dynamically at run-time through objects acquiring references to other objects.
- Advantages:
 - Contained objects are accessed by the containing class solely through their interfaces.
 - "Black-box" reuse, since internal details of contained objects are *not* visible.
 - Good encapsulation
 - Fewer implementation dependencies
- Disadvantages:
 - Resulting systems tend to have more objects
 - Interfaces must be carefully defined in order to use many different objects as composition blocks.

Nandigam

32

Object Interfaces

- An interface is the set of methods one object knows it can invoke on another object.
- An object can have many interfaces – an interface, essentially, is a subset of all the methods that an object implements.
- A *type* is a specific interface of an object.
- Different objects can have the same type and the same object can have many different types.
- An object is known by other objects only through its interface.
- Interfaces are the key to **pluggability**.
- A principle of reusable object-oriented design:
 - “Program to an interface, not an implementation”

Nandigam

33

Object Interfaces

- Advantages of interfaces:
 - Clients are unaware of the specific class of the object they are using
 - One object can be easily replaced by another
 - Object connections need not be hardwired to an object of a specific class, thereby increasing flexibility
 - Loosens coupling
 - Increases likelihood of reuse
 - Improves opportunities for composition since contained objects can be of any class that implements a specific interface
- Disadvantages of interfaces:
 - Modest increase in design complexity

Nandigam

34

Object Interfaces

■ Example with interfaces

```
/**
 * Interface IManeuverable provides the specification
 * for a maneuverable vehicle.
 */
public interface IManeuverable {
    public void left();
    public void right();
    public void forward();
    public void reverse();
    public void climb();
    public void dive();
    public void setSpeed(double speed);
    public double getSpeed();
}
```

Nandigam

35

Object Interfaces

■ Example with interfaces continued...

```
public class Car implements IManeuverable {
    // Code here.
}

public class Boat implements IManeuverable {
    // Code here.
}

public class Submarine implements IManeuverable {
    // Code here.
}
```

Nandigam

36

Object Interfaces

- Example with interfaces continued...
 - The method below from some other class can maneuver the vehicle without being concerned about what the actual class is (Car, Boat, Submarine) or what inheritance hierarchy it is in.

```
public void travel(IManeuverable vehicle) {
    vehicle.setSpeed(35.0);
    vehicle.forward();
    vehicle.left();
    vehicle.climb();
}
```

Nandigam

37

Assessing Design Quality

- "A module is a lexically contiguous sequence of program statements, bounded by boundary elements, with an aggregate identifier" by Yourdon and Constantine.
- By this definition, a module can be
 - Function, procedure, or method
 - Class
- Is there any criteria for judging the quality of a module?
- Two software quality metrics (originally proposed in early 1970s by W. Stevens, G. Myers, L. Constantine)
 - Module cohesion
 - Module coupling

Nandigam

38

Module Cohesion

- Cohesion is the degree of relatedness between elements **within** a module. It is an intra-module measure.
- Cohesion Levels
 - Functional cohesion (best or most desirable)
 - Sequential cohesion
 - Communicational cohesion
 - Procedural cohesion
 - Temporal cohesion
 - Logical cohesion
 - Coincidental cohesion (worst or least desirable)
- Modules with high cohesion tend to be more maintainable (modifiable and understandable) and reusable compared to modules with low cohesion.
- How do you improve the cohesion of a module?
 - Split it into two or more modules with higher cohesion

Nandigam

39

Module Coupling

- Coupling is the degree of dependence between two modules.
- Types of coupling
 - Normal coupling (low/best)
 - Data
 - Stamp
 - Control
 - Common coupling
 - Content coupling (high/worst)
- Normal coupling – coupling by parameters
- Common coupling – coupling via global data
- Content coupling – coupling that happens when one module directly references (or changes) the contents (data or code) of another module.

Nandigam

40



OO Design Process

- Design the system architecture or overall structure
 - Client/server architecture
 - 2-tier, 3-tier, n-tier
 - Console vs. desktop vs. Web-based
- Identify principle objects in each layer/tier of the system (presentation tier, application/business logic tier, database tier).
- Develop design models using UML
 - Static models
 - Dynamic models
- Specify object interfaces
- Specify algorithmic details of operations/methods
- Learn and use design patterns