What is modeling-in-the-small?

- Concerned with use of modeling techniques to describe program designs
- Focus on how modeling can be coupled with programming-in-the-small techniques
- Programs are relatively small and can be developed by 1 or 2 persons
- Program requirements are expressed in narrative form
What is a program design model?

- A program design model is a description of a solution implemented by a program
- A program design model should describe a program from different views
  - A view is a description of some aspect of a system
  - Separate descriptions of views are simpler to understand than a single description that integrates all views (an example of separation of concerns discussed later in this course)
- Two views of program design
  - Static view: Focuses on solution concepts and their relationships
  - Behavioral view: Focuses on interactions between objects and behavior of individual objects

Describing program design views

- UML is a defacto industry modeling language standard for describing systems from an OO perspective
- Program design models
  - **Program class model**: describes static aspect of a program
  - **Sequence diagrams**: describes interactions between design objects
  - Other behavioral models: activity diagrams, statemachine models
Overview

• Describing static design views using UML class diagrams
• OO Design: Assigning responsibility to classes

Program Design Class Models

Modeling the static design view
What is a program class?

• A class is a description of a set of objects that share the same properties (expressed as attributes, relationships, and operations)
  – At the programming level a class defines objects that will perform computations
  – Methods are code-specific implementations of operations

• An object is a concept, abstraction, or thing with identity that has meaning for an application.
  – An object is an instance of a class
  – Each object “knows” its class

What is a Program Class Model?

• Syntactically, a class model is a structure of classes.

• Semantically,
  – a program class model describes program-level objects (e.g., Java objects) and their links

• Key Question: What are the objects of interest in the solution space?
  – their responsibilities (in terms of attributes and methods)
  – their relationships
Class diagram example: Part of Movie Rental System class model

Diagram vs. model: A diagram can be a partial view of a model; large models can be described using multiple diagrams

Naïve Design: Code for Movie

```java
public class Movie{
    public static final int CHILDREN = 2;
    public static final int REGULAR = 0;
    public static final int NEW = 1;

    private string title;
    private int priceCode;

    public Movie (String mtitle, int mpriceCode){
        title = mtitle; priceCode=mpriceCode;}

    public int getPriceCode(){
        return priceCode;}

    public String getTitle(){
        return title;}

    public void setPriceCode (int arg){
        priceCode=arg;}
}
```

©Robert B. France
Code for Rental

```java
public class Rental {
    private Movie movie;
    private int daysRented;

    public Rental(Movie rmovie, int rdaysRented){
        movie = rmovie;
        daysRented = rdaysRented;
    }

    public int getDaysRented(){
        return daysRented;
    }

    public Movie getMovie(){
        return movie;
    }
}
```

Code for Customer

```java
public class Customer {
    private String name;
    private List<Rental> rentals = new ArrayList<Rental>();

    public Customer(String cname){
        name = cname;
    }

    public void addRental(Rental arg){
        rentals.addElement(arg);
    }

    public String getName(){
        return name;
    }

    public String statement(){...
    }
}
```
Basic class modeling notation

- Classes
- Class relationships
  - Associations
  - Generalization/Specialization
- Objects

Describing classes

- A class has the following structure:
  - Name compartment (mandatory)
  - Attributes compartment (optional)
  - Operations compartment (optional)
- Every class must have a unique name.
- An object of a class must have values associated with each attribute of the class
Style Guidelines for Classes

- Center class name.
- Capitalize the first letter of class names (if the character set supports uppercase).
- Left justify attributes and operations in plain face.
- Begin attribute and operation names with a lowercase letter.
- Put the class name in italics if the class is *abstract*.
  - An abstract class is one whose instances must be instances of a specialized class
  - At the implementation level, this translates to a class that cannot be instantiated
- Show full attributes and operations when needed and suppress them in other contexts or when merely referring to a class.

Depicting Classes

- Rectangle
  - getArea
  - height
  - width
  - resize

- Rectangle
  - height
  - width
  - resize

- Rectangle
  - height: int
  - width: int
  - getArea(): int
  - resize(int)
Attributes

- An attribute is a named property. Each object associates a value with each attribute of a class
- Properties with types that we want to treat as primitive are modeled as attributes
- Connections to other concepts are to be represented as associations, not attributes.

Attributes: Do’s & Don’t’s

<table>
<thead>
<tr>
<th>Person</th>
<th>Person</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name</td>
<td>street</td>
</tr>
<tr>
<td>address</td>
<td></td>
<td>municipality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>provOrState</td>
</tr>
<tr>
<td></td>
<td></td>
<td>country1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>postalCode1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>street2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>municipality2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>provOrState2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>country2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>postalCode2</td>
</tr>
</tbody>
</table>

Bad due to a plural attribute

Good solution. The type indicates whether it is a home address, business address etc.

Bad due to too many attributes, and inability to add more addresses
Do not use attributes to relate classes in class models!

<table>
<thead>
<tr>
<th>Worse</th>
<th>Customer</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rentedVideos: List of Video</td>
<td>renter : Customer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Better</th>
<th>Customer</th>
<th>1</th>
<th>Rents</th>
<th>1..*</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Modeling static relationships

- Two kinds of static relationships:
  - **Associations**
    - Model structural relationships among problem concepts
    - An association specifies a collection of links, where a link is a physical or conceptual connection among objects (an object is an instance of a class).
  - **Generalizations**
    - Represent generalization/specialization class structures
- The two kinds of relationships are orthogonal
Associations & Multiplicities

- Association Roles
  - When a class is part of an association its objects that are linked via the association play a *role* in the relationship.
  - You can name the role that an object plays in an association by placing the name at the class’s association end.
  - Formally, a class role is the set of objects that are linked via the association.
My class modeling rules

- All association ends must have multiplicities
  - There is a default, but most people do not remember it; better to be explicit to avoid confusing the reader
- All associations must be named (either an association name or a role name at each end)
Association classes
– Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes

Challenge: How do you model the above without an association class?

Reflexive associations
• It is possible for an association to connect a class to itself
Navigability

• In a design model one can indicate that an object “knows about” another object it is linked to by using navigation arrows on associations
  – In UML 2.0 one can also explicitly show that one object does not know about the objects it is linked to.

The constructs in diagrams 1, 2, and 4 are new to UML 2.0 and thus are most likely not supported by UML tools as yet.

*The top pair AB shows a binary association with two navigable ends.
*The second pair CD shows a binary association with two non-navigable ends.
*The third pair EF shows a binary association with unspecified navigability.
*The fourth pair GH shows a binary association with one end navigable and the other non-navigable.
*The fifth pair IJ shows a binary association with one end navigable and the other having unspecified navigability.
Navigability example

UML forms of aggregation

• Composition (strong aggregation)
  – parts are existent-dependent on the whole
  – parts are generated at the same time, before, or after the whole is created (depending on cardinality at whole end) and parts are deleted before or at the same time the whole dies
  – multiplicity at whole end must be 1 or 0..1

• (weak) Aggregation
Guidelines for identifying associations

- Focus on associations for which knowledge of the relationship must be preserved for some duration
  - An association should exist if a class
    - possesses
    - controls
    - is connected to
    - is related to
    - is a part of
    - has as parts
    - is a member of, or
    - has as members
  another class
- More important to identify concepts than associations
  - Too many associations can lead to confusing models
- Avoid showing redundant/derivable associations
Actions versus associations

• A common mistake is to represent actions as associations

Better: The `borrow` operation creates a `Loan`, and the `return` operation sets the `returnedDate` attribute.

Generalization/Specialization

• A generalization (or specialization) is a relationship between a general concept and its specializations.
  – Objects of specializations can be used anywhere an object of a generalization is expected (but not vice versa).

• Example: *Mechanical Engineer* and *Aeronautical Engineer* are specializations of *Engineer*
Generalization

Separate Target Style

Shared Target Style

© Robert B. France
Avoiding having instances change class

- An instance should never need to change class

```
Student
   ▲ attendance
  /       \
FullTimeStudent PartTimeStudent
```

Identifying generalizations and interfaces

- There are two ways to identify generalizations:
  - bottom-up
    - Group together similar classes creating a new superclass
  - top-down
    - Look for more general classes first, specialize them if needed
Object diagrams

- An object diagram describes a configuration of objects.
- One can view a class diagram as specifying a set of system states modeled by object configurations
  - A state (object configuration) that satisfies the constraints expressed in a class model is said to be **valid**.

Valid vs. Invalid states

Valid or Invalid?

Valid or Invalid?
Associations versus generalizations in object diagrams

- Associations describe the relationships (links) that will exist between objects at run time.
  - When you show an object diagram generated from a class diagram, there will be an object for each associated class joined by a link that is an instance of the association.

- Generalizations describe relationships between classes in class diagrams.
  - They do not appear in object diagrams at all.
  - An instance of any class should also be considered to be an instance of each of that class’s superclasses.

Assigning Responsibilities
Allocating responsibilities to classes

- A responsibility is something that the system is required to do.
  - Each functional requirement must be attributed to one of the classes
    - All the responsibilities of a given class should be clearly related.
    - If a class has too many responsibilities, consider splitting it into distinct classes
    - If a class has no responsibilities attached to it, then it is probably useless
    - When a responsibility cannot be attributed to any of the existing classes, then a new class should be created

- To determine responsibilities
  - Perform use case analysis
  - Look for verbs and nouns describing actions in the system description

Categories of responsibilities

- Setting and getting the values of attributes
- Creating and initializing new instances
- Loading to and saving from persistent storage
- Destroying instances
- Adding and deleting links of associations
- Copying, converting, transforming, transmitting or outputting
- Computing numerical results
- Navigating and searching
- Other specialized work
Key class modeling activities

- Identify key solution concepts that will be represented by program classes
- Assign responsibilities to classes
  - Determines class attributes and operations
  - Add new classes if none of the existing classes are not suitable for particular responsibilities
- Determine which objects need to know of other objects
  - Determines navigability

Identifying operations

- Operations are needed to realize the responsibilities of each class
  - There may be several operations per responsibility
  - The main operations that implement a responsibility are normally declared public
  - Other methods that collaborate to perform the responsibility must be as private as possible
An example (responsibilities)

- Creating a new regular flight
- Searching for a flight
- Modifying attributes of a flight
- Creating a specific flight
- Booking a passenger
- Canceling a booking

An example of a design class diagram
• Making a bi-directional link between two existing objects;
  • e.g. adding a link between an instance of SpecificFlight and an instance of Airplane.

  a1. (public) The instance of SpecificFlight
      • makes a one-directional link to the instance of Airplane
      • then calls operation a2.

  a2. (non-public) The instance of Airplane
      • makes a one-directional link back to the instance of SpecificFlight

• Creating an object and linking it to an existing object
  • e.g. creating a FlightLog and linking it to a SpecificFlight.

  b1. (public) The instance of SpecificFlight
      calls the constructor of FlightLog (operation b2)
      then makes a one-directional link to the new instance of FlightLog.

  b2. (non-public) Class FlightLog’s constructor
      makes a one-directional link back to the instance of SpecificFlight.
• Creating an association class, given two existing objects
• e.g. creating an instance of Booking, which will link a SpecificFlight to a PassengerRole.

c1. (public) The instance of PassengerRole
   • calls the constructor of Booking (operation 2).

c2. (non-public) Class Booking’s constructor, among its other actions
   • makes a one-directional link back to the instance of PassengerRole
   • makes a one-directional link to the instance of SpecificFlight
   • calls operations 3 and 4.

c3. (non-public) The instance of SpecificFlight
   • makes a one-directional link to the instance of Booking.

c4. (non-public) The instance of PassengerRole
   • makes a one-directional link to the instance of Booking.

• Changing the destination of a link
• e.g. changing the Airplane of a SpecificFlight, from airplane1 to airplane2

d1. (public) The instance of SpecificFlight
   • deletes the link to airplane1
   • makes a one-directional link to airplane2
   • calls operation d2
   • then calls operation d3.

d2. (non-public) airplane1
   • deletes its one-directional link to the instance of SpecificFlight.

d3. (non-public) airplane2
   • makes a one-directional link to the instance of SpecificFlight.
• Searching for an associated instance
  • e.g. searching for a crew member associated with a SpecificFlight that has a certain name.

  e1. (public) The instance of SpecificFlight
      • creates an Iterator over all the crewMember links of the SpecificFlight\n      • for each of them call operation e2, until it finds a match.

  e2. (may be public) The instance of EmployeeRole returns its name.

Implementing Class Diagrams in Java

• Attributes are implemented as instance variables
• Generalizations are implemented using extends
• Interfaces are implemented using implements
• Associations are normally implemented using instance variables
  • Divide each two-way association into two one-way associations
    — so each associated class has an instance variable.
  • For a one-way association where the multiplicity at the other end is ‘one’ or ‘optional’
    — declare a variable of that class (a reference)
  • For a one-way association where the multiplicity at the other end is ‘many’:
    — use a collection class implementing List, such as Vector
Example: **SpecificFlight**

class SpecificFlight
{
  private Calendar date;
  private RegularFlight regularFlight;
  private TerminalOfAirport destination;
  private Airplane airplane;
  private FlightLog flightLog;
  private ArrayList crewMembers;
  // of EmployeeRole
  private ArrayList bookings
  ...
}

Example: **SpecificFlight**

- // Constructor that should only be called from
- // addSpecificFlight
- SpecificFlight(
  Calendar aDate,
  RegularFlight aRegularFlight)
- {
  date = aDate;
  regularFlight = aRegularFlight;
  }

© Robert B. France 2-55

© Robert B. France 2-56
Example: **RegularFlight**

```java
class RegularFlight
{
    private ArrayList specificFlights;
    ...
    // Method that has primary
    // responsibility
    public void addSpecificFlight(
            Calendar aDate)
    {
        SpecificFlight newSpecificFlight;
        newSpecificFlight =
            new SpecificFlight(aDate, this);
        specificFlights.add(newSpecificFlight);
    }
    ...
}
```

**Responsibility Assignment Principles**

What principles should we adhere to when developing OO designs?

Key activity: determining what each class in an OO design is responsible for.

Principles
- Cohesion
- Expert
- Creator
- Low Coupling
- Controller
High Cohesion

• Cohesion is a measure of how diverse an entity’s features are.
  – A highly cohesive class has features that pertain to a single concept
  – A highly cohesive class has one general responsibility
    • Guideline: Should be able to describe responsibility of a highly cohesive class in one sentence
    • Use sentence as comment in code
• Guideline: Assign a responsibility so that parts of the class are strongly related and the class responsibility is tightly focused
  – Class easier to understand
  – Easier to maintain and reuse

Levels of Cohesion

• Very low cohesion: class is responsible for many things in different functional contexts
• Low cohesion: class is solely responsible for a complex group of tasks in a single functional area
• Moderate cohesion: class is responsible for relatively simple tasks in different but related functional areas
• High cohesion: class is responsible for a group of tasks in a single functional area and discharges its responsibility by delegating some of its responsibilities to other classes.
• Example: Machine-Machine Schedule association
When to ignore high cohesion guidelines

• A class that provides a single point of entry into a system may sometimes be desirable
  – Such a class is called a Façade and provides external clients with a single point of access to services offered by a system
• For efficiency reasons it may be more appropriate to place two diverse classes in the same class
  – Rather than an object delegating responsibility for a service to another object it may carry it out itself to avoid delegation performance overhead

Expert

• Assign responsibility to the class that has the information necessary to discharge the responsibility.
• Naïve use can lead to undesirable coupling and low cohesion.
  – Giving a class the responsibility for storing its objects in a database leads to low coupling and undesirable coupling
    • Low cohesion: class contains code related to database handling between the
    • Undesirable coupling: class is tightly coupled to database services provided by another system
  – Example: Elevator Control System
Creator

- Class B can be responsible for creating objects of A in the following situations:
  - A is a part class of B
  - B is a container of A objects
  - B records A objects
  - B has the data needed to initialize A objects

Low Coupling

- Assign responsibilities to reduce high coupling to unstable classes (i.e., classes with high probability of significant changes)
  - Reduces impact of change
  - Classes can be understood in relative isolation
- Forms of coupling in OO designs
  - Class X contains a reference to Class Y objects
  - Class X operation includes calls to Class Y operations
  - Class X operation has a Class Y object as a parameter or declares a Class Y object as a local variable
  - Class X is a direct or indirect subclass of Class Y
  - Class X implements an interface Y
- Classes designed for reuse should have low coupling. Why?
Controller

- Assign responsibility for handling a system event to a class representing the system or a class that is responsible for handling the events in a group of related use cases.
  - A system event is an event generated by an actor. A system event results in the execution of a system operation.
  - A controller is a non-user interface class responsible for receiving and handling system events. A controller defines the method for the system operation.
- A good controller delegates the work needed to handle a system event to other objects.
  - A controller controls and coordinates the collaborating objects.
  - A controller does not do much of the actual work.

Controller Options

- Presentation objects (UI objects) should not be responsible for handling events
  - Decouple presentation layer from application processing layer. Why?
- System as controller
  - Referred to as a façade controller
  - Use when number of system events is not large
    - Large number of events can lead to a controller with low cohesion and high coupling
- Use case handlers
  - For each use case design a controller that handles the use case events
  - Use when number of system events is large
Bloated Controllers

• Signs of problematic design
  – Interface objects handle system events directly
  – Controller object handles many events
  – Controller object performs bulk of work needed to handle event.
  – Controller class has many attributes because of its many responsibilities.

General Guidelines

• Avoid dumb objects: objects that hold data and provide only get/set methods
• Avoid “god” controllers: a “god’ controller is one that requests state information (e.g., using a get method) and uses the information to make decisions or perform calculations
• Avoid coupling by having services above and beyond get/set services in interface of objects
• A client should request an object to do something on its behalf, not request information about an object’s state.