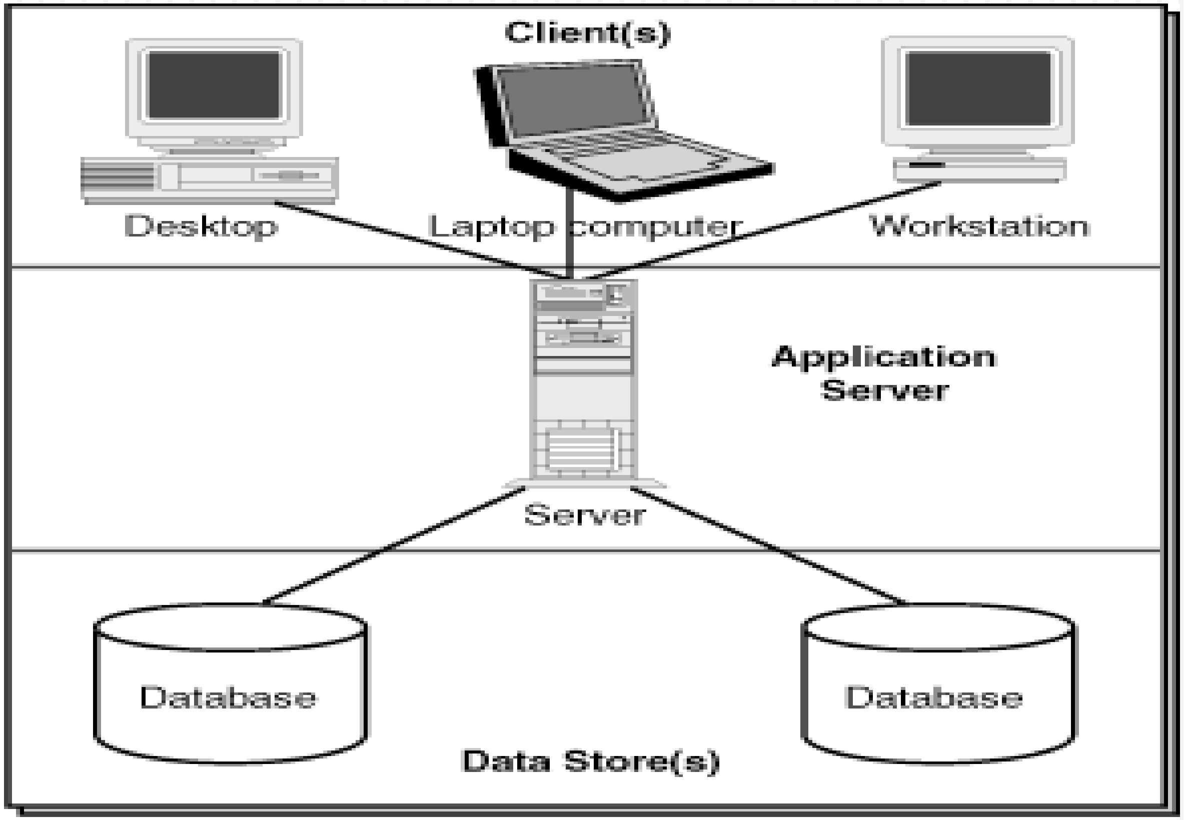
ADVANCED TOPICS

CIS*2430 (Fall 2010)

Three-Tier Architecture



Types of Applications

- **Standalone Applications:**

  - One machine for both server and client

- **Thick Client:**

  - Separate machines for server and clients

  - Clients are heavy and tend to be different for different users and machines

- **Thin Client:**

  - Separate machines for server, middleware, and clients
  - Clients are light and tend to the same for all users and machines

Unified Modeling Language

- UML is a graphical language for capturing and expressing knowledge about a subject

- UML is used for specifying, visualizing, constructing, and documenting systems

- UML is based on the OOP paradigm

- UML is the result of unifying the best engineering practices for modeling systems (principles, techniques, methods, and tools).

UML Diagrams

- UML defines nine types of diagrams: class, object, use case, sequence, collaboration, statechart, activity, component, and deployment.
- For all diagrams, concepts are depicted as symbols and relationships among concepts are depicted as links connecting symbols.
- Both concepts and links can be named.

History of UML

- As OOP has developed, different groups have developed graphical or other representations for OOP design
- In 1996, Brady Booch, Ivar Jacobson, and James Rumbaugh released an early version of UML
  - Its purpose was to produce a standardized graphical representation language for object-oriented design and documentation
- Since then, UML has been developed and revised in response to feedback from the OOP community
  - Today, the UML standard is maintained and certified by the Object Management Group (OMG)

Class Diagrams

- Classes are central to OOP, and the *class diagram* is the easiest of the UML graphical representations to understand and use

- A class diagram is divided up into three sections
  - The top section contains the class name
  - The middle section contains the data specification for the class
  - The bottom section contains the actions or methods of the class

A Class Diagram

Display 12.1 A UML Class Diagram

```
Square
------------------------------------------
- side: double
- xCoordinate: double
- yCoordinate: double
------------------------------------------
+ resize(double newSide): void
+ move(double newX, double newY): void
# erase(): void
...
```

Class Diagrams

- The data specification for each piece of data in a UML diagram consists of its name, followed by a colon, followed by its type

- Each name is preceded by a character that specifies its access type:
  - A minus sign (-) indicates private access
  - A plus sign (+) indicates public access
  - A sharp (#) indicates protected access
  - A tilde (~) indicates package access

Class Diagrams

- Each method in a UML diagram is indicated by the name of the method, followed by its parenthesized parameter list, a colon, and its return type
- The access type of each method is indicated in the same way as for data
- A class diagram need not give a complete description of the class
  - Missing members are indicated with an ellipsis (three dots)

Class Interactions

- UML has various ways to indicate the information flow from one class/object to another using different sorts of annotated links
- UML has annotations for class groupings into packages, for inheritance, and for other interactions
- In addition to these established annotations, UML is extensible

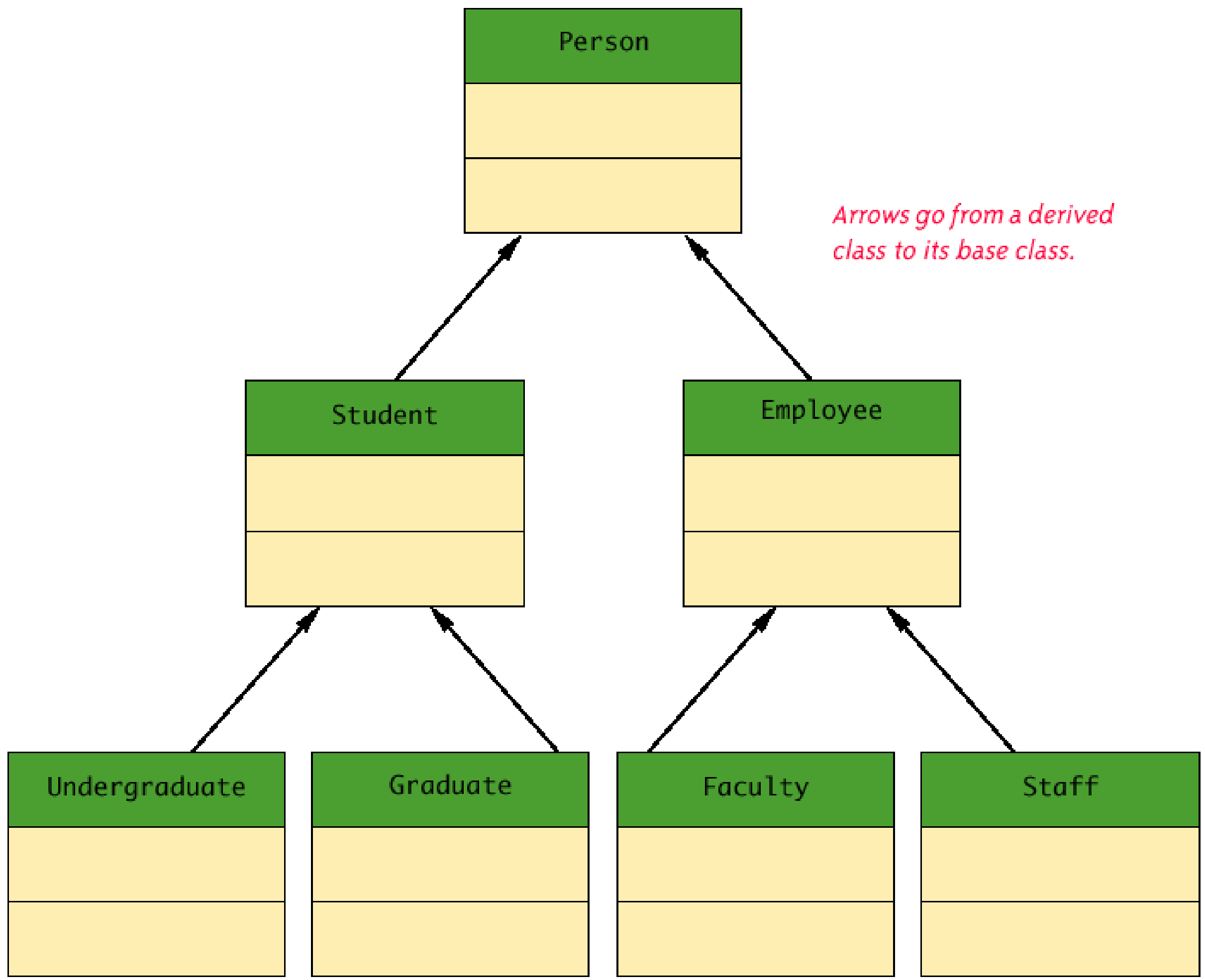
Inheritance Links

- *Inheritance links* show the relationship between a base class and its derived class(es)
  - Normally, only as much of the class diagram is shown as is needed
  - Note that each derived class may serve as the base class of its derived class(es)

- Each base class is drawn above its derived class(es)
  - An upward pointing arrow is drawn between them to indicate the inheritance relationship

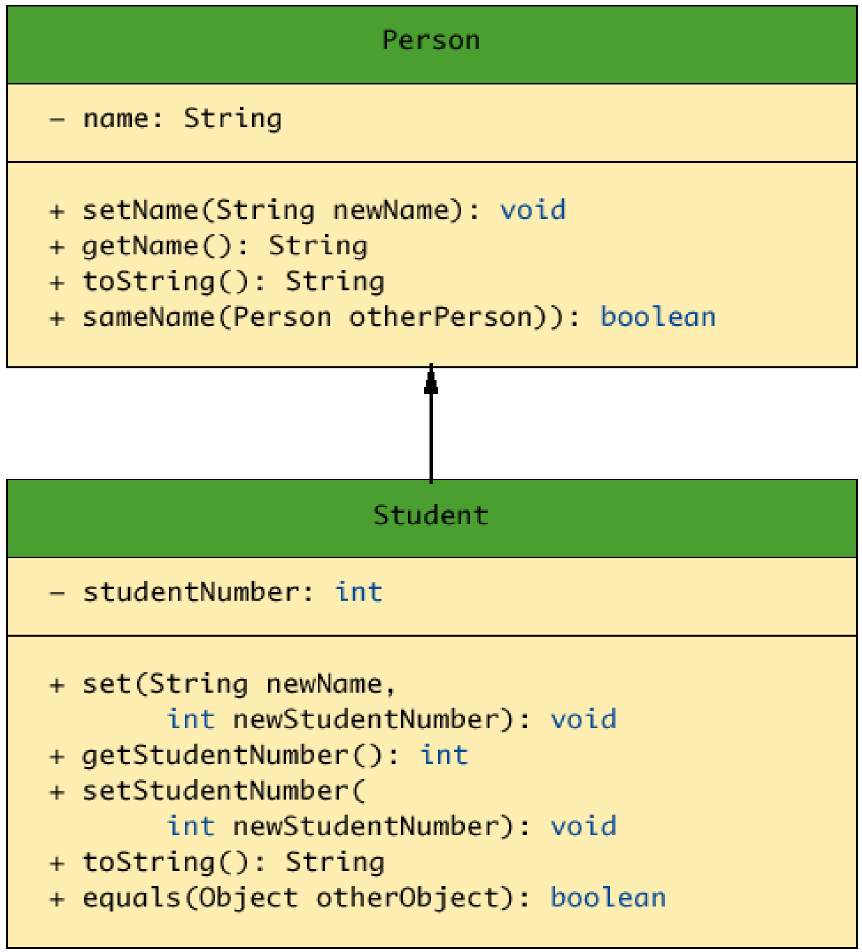
A Class Hierarchy in UML

Display 12.2 A Class Hierarchy in UML Notation

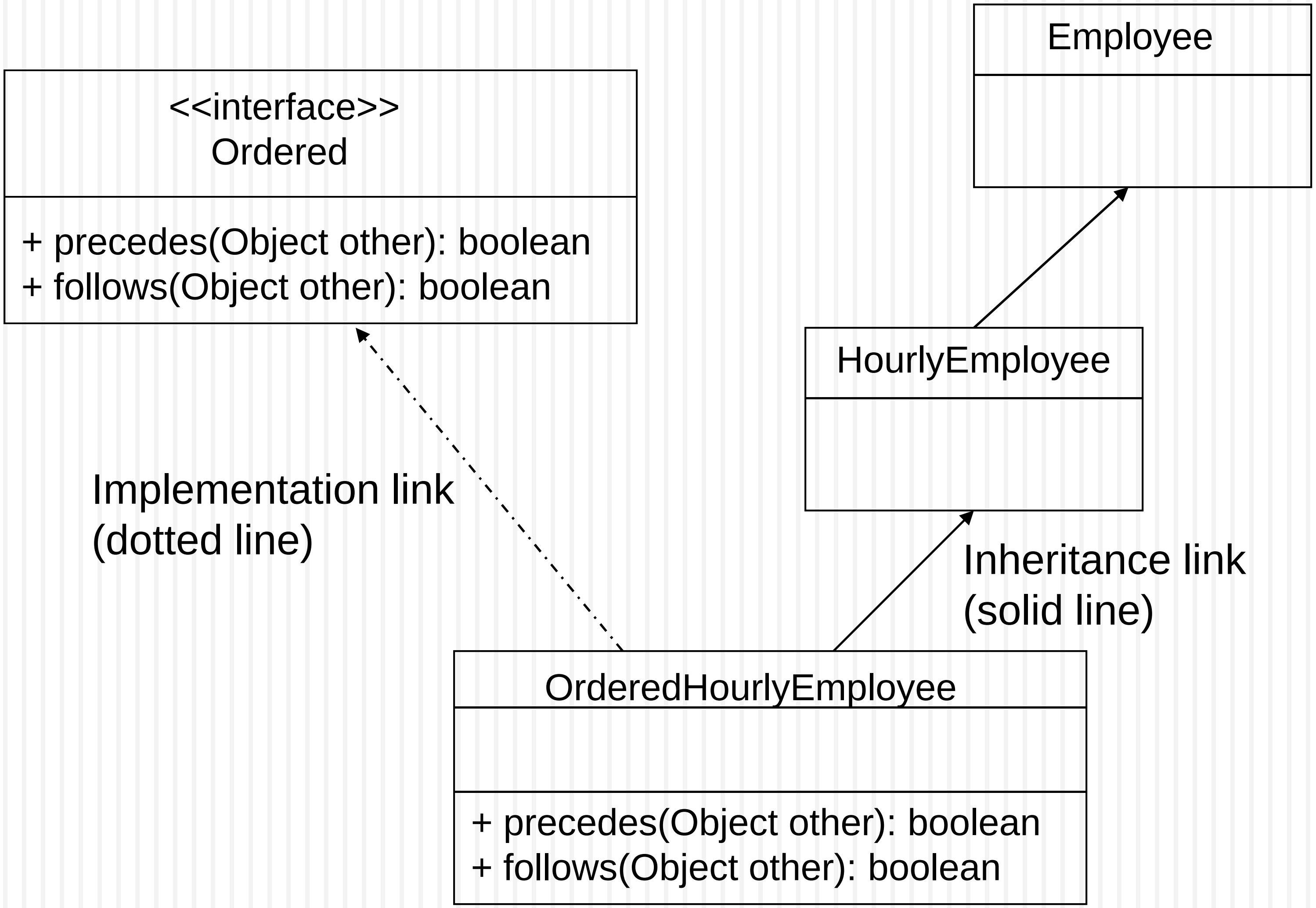


Details of a Class Hierarchy

Display 12.3 Some Details of a UML Class Hierarchy



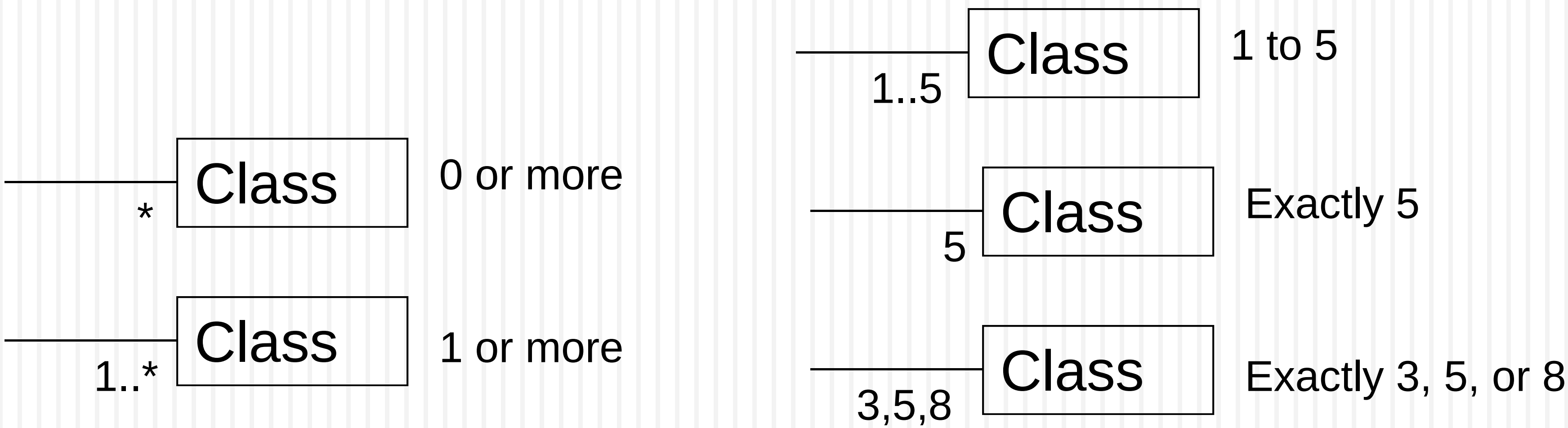
Interfaces and Implementations



Composition Links

- Composition links model the “has-part” relationship:

- Multiplicity:



Design Patterns

- A design pattern names and identifies the key aspects of a common solution that makes it useful for creating a reusable object-oriented design/software.

- A pattern generally has four components:
  - Name: used to refer to the pattern
  - Problem: when to apply the pattern
  - Solution: the elements that make up the design, their relationships, responsibilities, and collaborations
  - Consequences: the results and trade-offs of applying the pattern

Container-Iterator Pattern

- A *container* is a class or other construct whose objects hold multiple pieces of data
  - An array is a container
  - Vectors and linked lists are containers
  - A String value can be viewed as a container that contains the characters in the string

- Any construct that can be used to cycle through all the items in a container is an *iterator*
  - An array index is an iterator for an array

- The *Container-Iterator* pattern describes how an iterator is used on a container

Adaptor Pattern

- The Adaptor pattern transforms one class into a different class without changing the underlying class, but by merely adding a new interface
  - For example, one way to create a stack data structure is to start with an array, then add the stack interface

Model-View-Controller Pattern

- The *Model-View-Controller* pattern is a way of separating the I/O task of an application from the rest of the application
  - The Model part of the pattern performs the heart of the application
  - The View part displays (outputs) a picture of the Model's state
  - The Controller is the input part: It relays commands from the user to the Model

Model-View-Controller Pattern

- Each of the three interacting parts is normally realized as an object with responsibilities for its own tasks

- The Model-View-Controller pattern is an example of a divide-and-conquer strategy
  - One big task is divided into three smaller tasks with well-defined responsibilities

Model-View-Controller Pattern

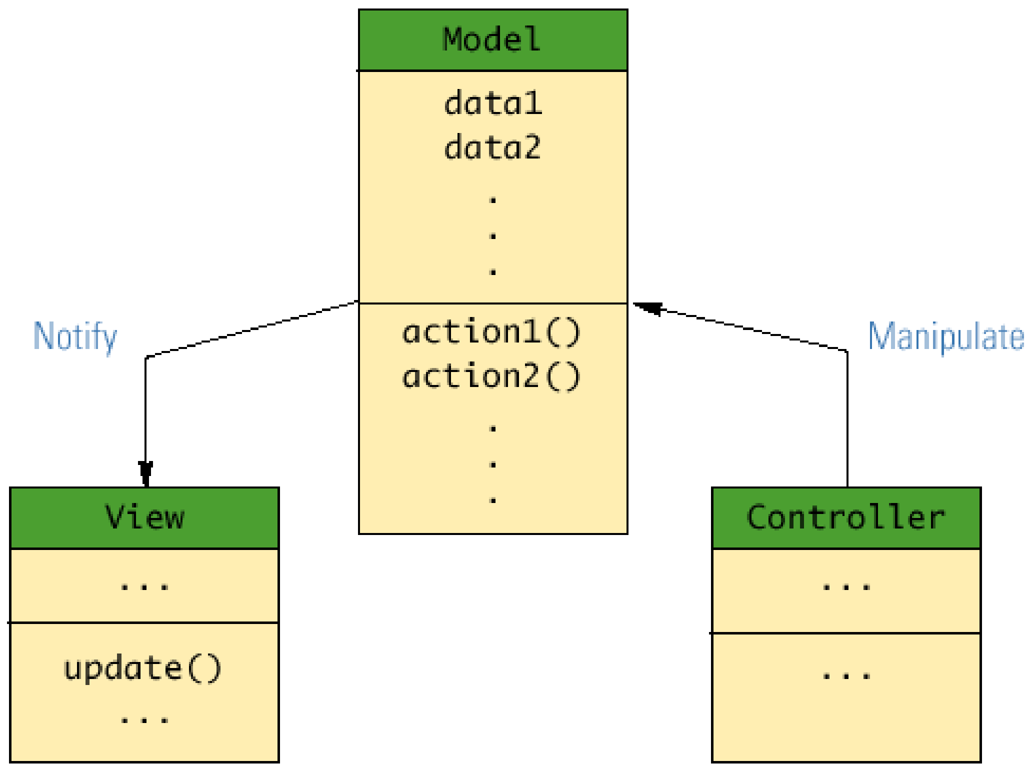
- As an example, the Model might be a container class, such as an array.
- The View might display one element of the array
- The Controller would give commands to change the element at a specified index
- The Model would notify the View to display a new element whenever the array contents changed or a different index location was given

Model-View-Controller Pattern

- Any application can be made to fit the Model-View-Controller pattern, but it is particularly well suited to GUI (Graphical User Interface) design projects
  - The View can then be a visualization of the state of the Model

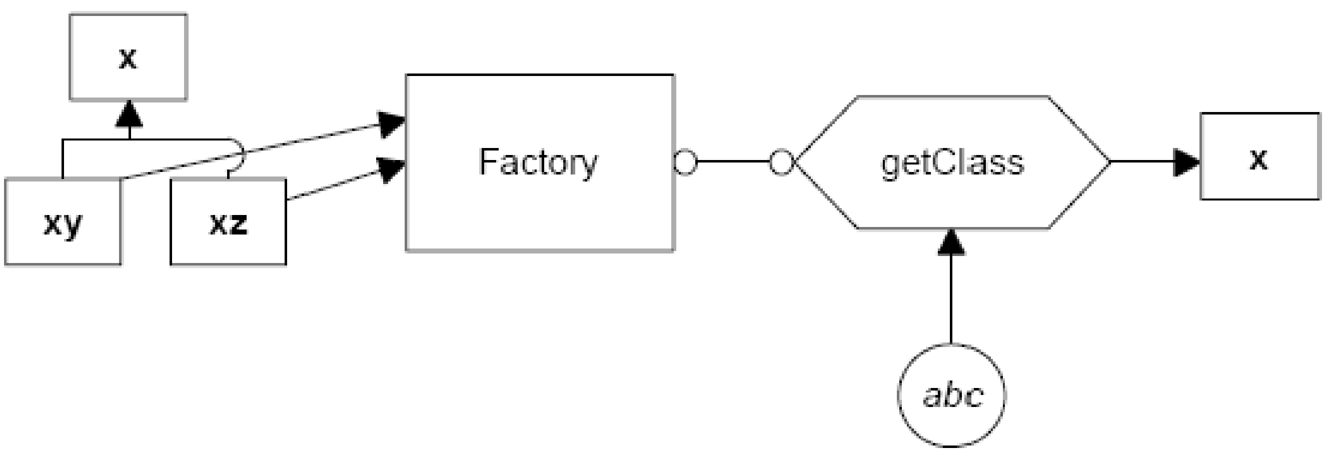
Model-View-Controller Pattern

Display 12.4 Model-View-Controller Pattern



Factory Method

- Define an interface for creating an object, but let subclasses decide which class to instantiate
  - Factory Method lets a class defer instantiation to subclasses



Factory Method (1/5)

- Implement a simple entry form that allows the user to enter name either as “firstname lastname” or as “lastname, firstname”

- The base class:

```
public class Namer {
   protected String last;
   protected String first;
   public String getLast() { return last; }
   public String getFirst() { return first; }
}
```

Factory Method (2/5)

- Two derived classes:

```
public class FirstFirst extends Namer {
   public FirstFirst(String s) {
      int i = s.lastIndexOf(" ");
      if (i > 0) {
         first = s.substring(0, i).trim();
         last = s.substring(i+1).trim();
      } else {
         first = "";
         last = s;
      }
   }
}
```

```
public class LastFirst extends Namer {
   public LastFirst(String s) {
      int i = s.indexOf(",");
      if (i > 0) {
         last = s.substring(0, i).trim();
         first = s.substring(i+1).trim();
      } else {
         last = s;
         first = "";
      }
   }
}
```

Factory Method (3/5)

- Building the factory class:

```
public class NameFactory {
   public Namer getNamer(String entry) {
       int i = s.indexOf(“,”);    // comma determines name order
       if (i > 0)
          return new LastFirst(entry);
       else
          return new FirstFirst(entry);
   }
}
```

Factory Method (4/5)

- In the constructor of the GUI program, create a factory instance:

```
NameFactory nfactory = new NameFactory();
```

- In responding to a button event, call the computeName method:

```
private void computeName() {
  namer = nfactory.getNamer(entryField.getText());
  txtFirstName.setText(namer.getFirst());
  txtLastName.setText(namer.getLast());
}
```

Factory Method (5/5)

- The GUI interface: