UML

- Back Ground
- Class Diagrams
- Inheritance Hierarchy
- UML
- Patterns
UML Back Ground

• UML stands for Unified Modeling Language

• UML is a graphical language used for designing and documenting OO software

• In 1996, Brady Booch, Ivar Jacobson, and James Rumbaugh released an early version of UML

• 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)
UML Introduction

• Pseudo code is a way of representing a program in a linear and algebraic manner
  – It simplifies design by eliminating the details of programming language syntax

• Graphical representation systems for program design have also been used
  – Flowcharts and structure diagrams for example

• *Unified Modeling Language (UML)* is yet another graphical representation formalism
  – UML is designed to reflect and be used with the OOP philosophy
UML 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.
UML 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
UML 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
## Display 12.1  A UML Class Diagram

<table>
<thead>
<tr>
<th>Square</th>
</tr>
</thead>
<tbody>
<tr>
<td>- side: \texttt{double}</td>
</tr>
<tr>
<td>- \texttt{xCoordinate: double}</td>
</tr>
<tr>
<td>- \texttt{yCoordinate: double}</td>
</tr>
</tbody>
</table>

+ \texttt{resize(double newSide): void}  
+ \texttt{move(double newX, double newY): void}  
# \texttt{erase()}: void  
...
Inheritance Diagrams

• An *inheritance diagram* shows the relationship between a base class and 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
Inheritance Diagram Example

Display 12.2  A Class Hierarchy in UML Notation

Arrows go from a derived class to its base class.
A Detailed UML Inheritance Hierarchy

Display 12.3 Some Details of a UML Class Hierarchy

Person

- name: String

+ setName(String newName): void
+ getName(): String
+ toString(): String
+ sameName(Person otherPerson): boolean

Student

- studentNumber: int

+ set(String newName,
    int newStudentNumber): void
+ getStudentNumber(): int
+ setStudentNumber(
    int newStudentNumber): void
+ toString(): String
+ equals(Object otherObject): boolean
Patterns

• *Patterns* are design outlines that apply across a variety of software applications

  – To be useful, a pattern must apply across a variety of situations

  – To be substantive, a pattern must make some assumptions about the domain of applications to which it applies
A Sorting Pattern

- Patterns are best explained by looking at examples

- The most efficient sorting algorithms all seem to follow a divide-and-conquer strategy

- Given an array \( a \), and using the \(<\) operator, these sorting algorithms:
  - Divide the list of elements to be sorted into two smaller lists (\textit{split})
  - Recursively sort the two smaller lists (\textit{sort})
  - Then recombine the two sorted lists (\textit{join}) to obtain the final sorted list
A Sorting Pattern

- The method `split` rearranges the elements in the interval `a[begin]` through `a[end]` and divides the rearranged interval at `splitPoint`

- The two smaller intervals are then sorted by a recursive call to the method `sort`

- After the two smaller intervals are sorted, the method `join` combines them to obtain the final sorted version of the entire larger interval

- Note that the pattern does not say exactly how the methods `split` and `join` are defined
  - Different definitions of `split` and `join` will yield different sorting algorithms
Merge Sort

• The simplest realization of this sorting pattern is the *merge sort*

• The definition of *split* is very simple
  – It divides the array into two intervals without rearranging the elements

• The definition of *join* is more complicated

• Note: There is a trade-off between the complexity of the methods *split* and *join*
  – Either one can be made simpler at the expense of making the other more complicated
Quick Sort

• In the *quick sort* realization of the sorting pattern, the definition of *split* is quite sophisticated, while *join* is utterly simple
  – First, an arbitrary value called the *splitting value* is chosen
  – The elements in the array are rearranged:
    • All elements less than or equal to the splitting value are placed at the front of the array
    • All elements greater than the splitting value are placed at the back of the array
    • The splitting value is placed in between the two
Quick Sort

• Note that the smaller elements are not sorted, and the larger elements are not sorted
  – However, all the elements before the splitting value are smaller than any of the elements after the splitting value

• The smaller elements are then sorted by a recursive call, as are the larger elements

• Then these two sorted segments are combined
  – The **join** method actually does nothing
public class Parent {
    int anumber;
    public void PrintQuiz() {}
    void PrintA() {}
    protected int PrintB() {}
}

class Child1 extends Parent {
    public char ID;
    public int getStatus() {}
    public void setId() {}
}

class Child2 extends Parent {
    private double Value;
    private void PrintValues() {}
    public boolean Method1() {}
}

Unit 22