Software Systems Modelling

Mathenge Kanyaru

Welcome to SSM

- Personal introduction
- Course & prerequisites
- Objectives & topics
- logistics & mindset
- Diagnosis activity
- Getting started

My Research Interests

- Software Requirements Engineering
- Software design and applications in:
  - Product Lines
  - Automated program generation
  - Health informatics

Your projects

- Am happy to hear your suggestions within my areas of interest

Learning outcomes

- 1. Demonstrate expert knowledge of the changing nature of software-intensive systems.
- 2. Select appropriate techniques systematically from the range of methods and tools available to develop such systems.
- 3. Demonstrate expertise of object oriented modelling, and apply the Unified Modelling Language (UML) to produce appropriate software design models for software-intensive systems.
- 4. Apply Model Driven Architecture (MDA) and patterns in order to create designs for business applications effectively.
- 5. Understand the professional issues, implications and impact of the production of software systems models.

Topics

- Introduction
- Basics of UML
  - Class diagrams, state diagrams, use cases
- Design Patterns Review
- Modelling Beyond UML
  - SysML, Process Modelling
- MDA
  - Foundations
  - Frameworks
  - Application of patterns
  - Example environments (IDEs)
  - Introduction to code generation
  - Limitations

- Software Product Lines
  - Basic concepts
  - Variability Modelling
  - Basic variability implementation techniques
  - SPL and patterns
  - MDD and Software Product Lines

Contacts

- Office:
  - P104a
  - pls note – I share with us, don’t come whilst tipsy!
  - Email: jkanyaru@...
- Lecture notes:
  - Will be available beforehand
  - Download and read, do your part!

Assessment

- Course work – 30%
  - Modelling, patterns, and program construction
- Exam – 70%

How to gain the most

- Active learning
  - Read lecture notes
  - Bring questions
- Learn by doing
  - Pen and paper exercises
  - Tool exercises
- Constant practice
- Participation
- Nutshell – do not believe you know until you have tried it!

Diagnosis activity

- Please answer the exercise sheet
  - Does not count for your grade

<=25 mins

- Goal
  - Assess your Object Oriented background
  - Assess your knowledge of UML
  - Interests and expectations
  - Tease out your internship experience

Introduction to UML

- What is UML?
  - a general purpose visual modelling language that is used to specify, visualize, construct and document artefacts of a system
- UML provides notation to describe OO designs
- Geared for Object Oriented systems
  - Parts of UML could be applicable to other programming paradigms

History of UML

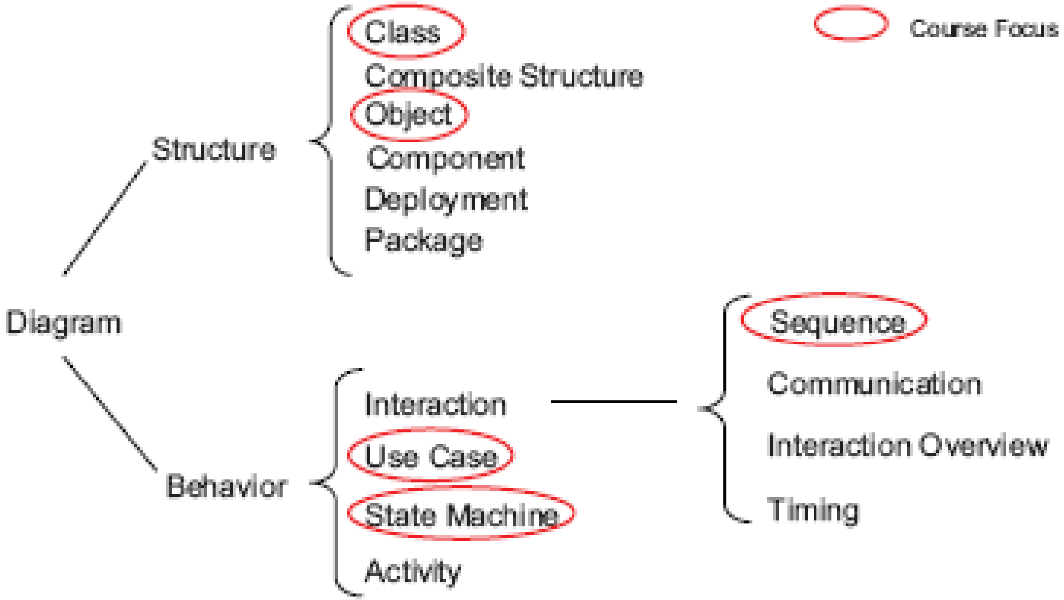
- Object Oriented Design
  - Started around 1990's
  - Collection of different modelling techniques
    - Booch, Jacobson, Rumbaugh, Coad, etc.
- By early 1990's
  - Disparity between different camps
  - A standard was needed

History of UML…

- Object Management Group (OMG)
  - Set up a group to establish a standard
  - Mostly people from industry
- First draft published in 1997
- Current version UML 2.3 – but using diagrams from UML 2.1

UML diagrams

- UML 2.1 supports 13 kinds of diagrams



Course Focus

Diagram

Structure

Class

Composite Structure

Object

Component

Deployment

Package

Behavior

Interaction

Use Case

State Machine

Activity

Sequence

Communication

Interaction Overview

Timing

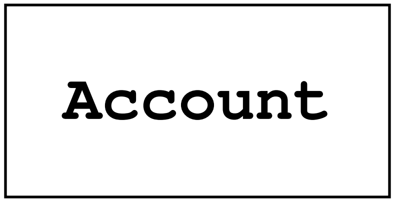
Why bother with UML?

- It is THE de facto standard for OO design
- Commonly used in industry and research
  - Chances are you will actually need it at some point in your career

Class diagrams

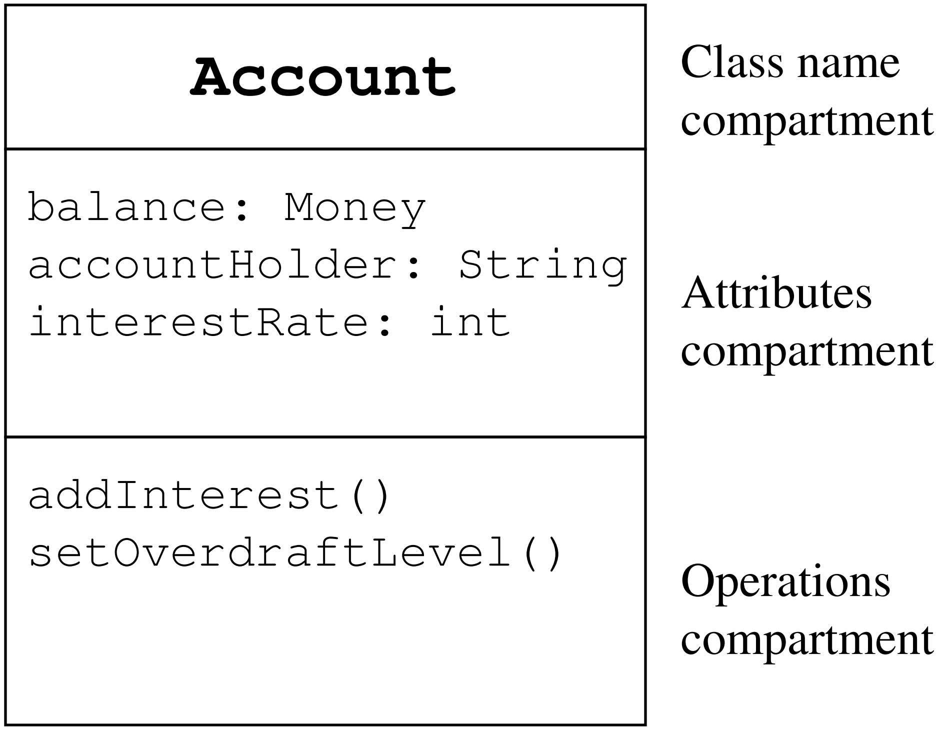
- Class diagram
  - Describes the types of objects in a system and the relationships among them
  - Class structure:
    - Name
    - Fields
    - Methods/operations

Simple class syntax



Account

“A class is a description of a set of objects that share the same attributes, operations relationships, and semantics.”



Account

```
balance: Money
accountHolder: String
interestRate: int
```

```
addInterest()
setOverdraftLevel()
```

Class name compartment

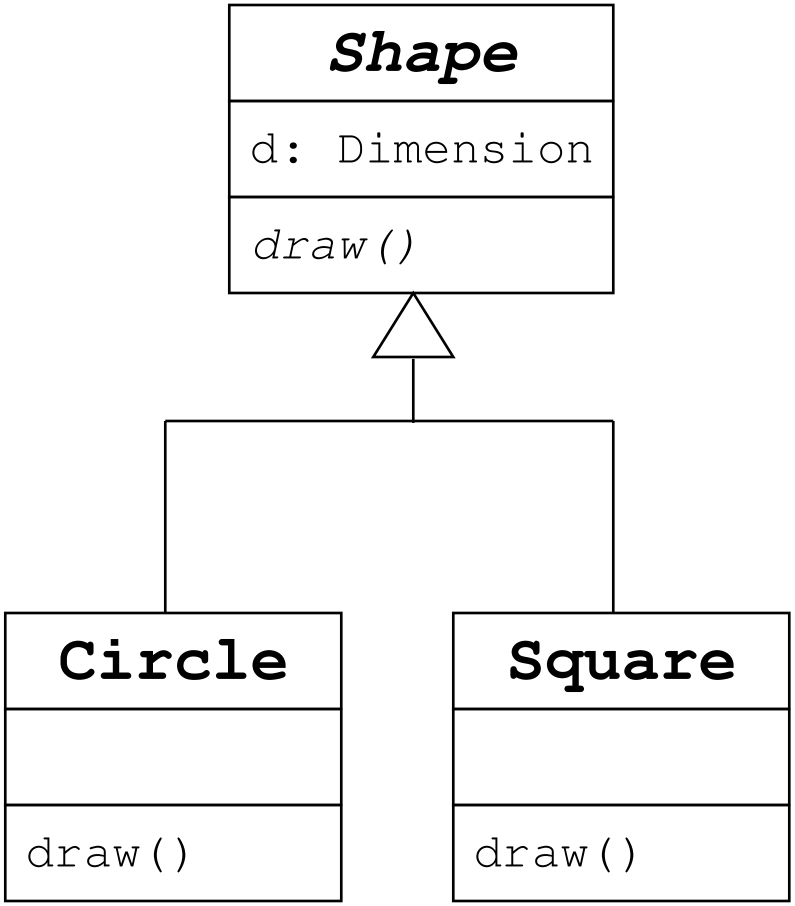
Attributes compartment

Operations compartment

Modifiers (or access privileges in Java)

- **public** members
  - referenced from anywhere
  - UML denoted with +
- **private** members
  - referenced in instances of the class that declares them
  - UML denoted with -
- **protected** members
  - referenced in subclasses and classes in same package
  - UML denoted with #
- default members also known as *package privilege*
  - Referenced in classes in the same package
  - UML denoted with ~

Abstract classes



Shape

d: Dimension

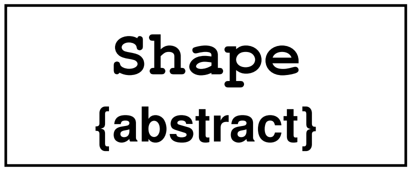
draw()

Circle

draw()

Square

draw()



Shape

{abstract}

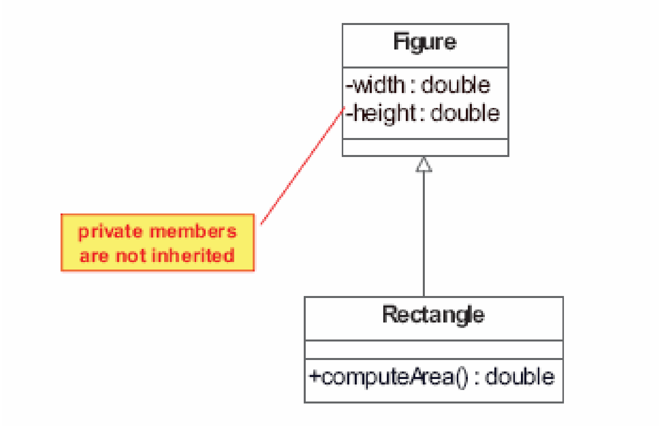
Relationship types

- Generalization
- Association
- Aggregation
- Composition

Generalisation

- Definition
  - Taxonomic relationship between a more general description and a more specific one that extends it
- In OO generalization relates to inheritance
- In UML denoted with an arrow line with an empty arrowhead from subclass to superclass

Generalisation example



Simple Exercise

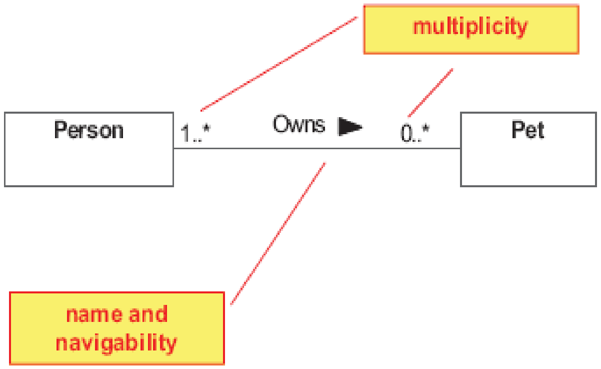
- Write in Java a class called **Person** that has the **name**, **last name**, and **age** of a person.
- Define a second class called **Employee** that extends **Person** and adds **salary** and **job title** information.
- Draw a UML class diagram to depict **Person** and **Employee** classes.

Name, Navigability, Multipilicity

- Name – Optional
  - Related to problem domain
  - Typically a verb
- Navigability
  - Establishes the direction of the relation
  - Denoted with a filled arrow head
  - Can be bidirectional
- Multiplicity
  - Establishes how many objects participate in the relation
  - Typical: 0, 1, 0..1, 1..*, *
  - Default: 1

Association

- Definition
  - Connections between two classes
  - implies a connection of instances of both classes
    - class X uses/references/knows class Y
- Denoted in UML with a solid line
- Example: A Person owns zero or more Pets

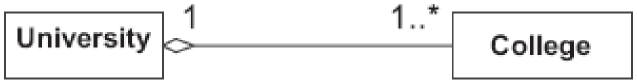


Aggregation

- Specialized case of association
- Describes a whole-part relationship between classes
  - Whole – aggregate
  - Part – constituent
- Aggregation characteristics
  - Aggregate can exists without parts
  - An object can belong to more than one aggregate
  - Constituents tend to be of the same class

Aggregation Example

An university is comprised of many colleges

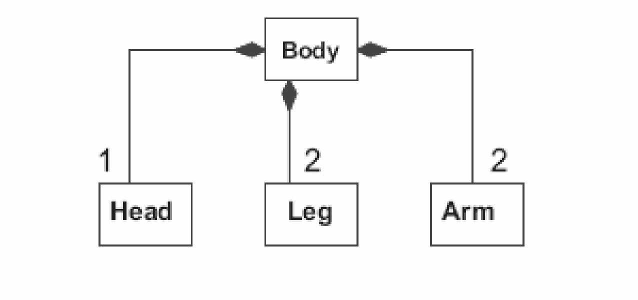


Composition

- Specialized case of association
  - Stronger form of ownership than aggregation
    - Objects have same life time
- Describes a whole-part relationship between classes
  - Whole – composite
  - Part – component
- Composition characteristics
  - Composite cannot exists without its components
  - An object can belong to only one component
  - Components tend to be of different classes

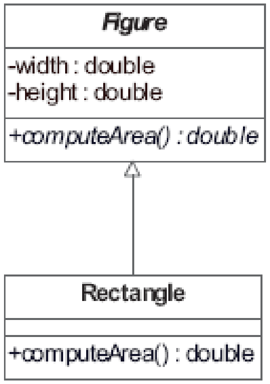
Composition Example

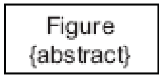
- A human body is composed of 1 head, 2 arms and 2 legs



Abstract classes

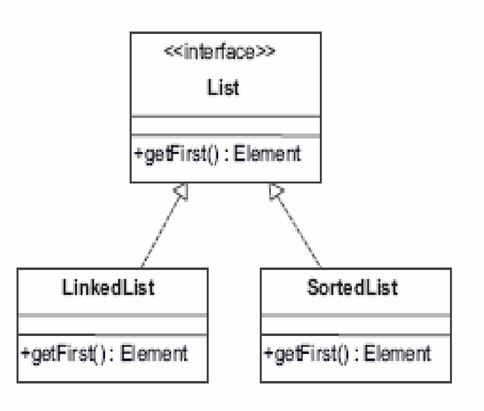
- Abstract classes
  - Their definition is incomplete
  - They are template classes
  - They are meant to be sub-classed
- In UML:
  - Class name in italics
  - Method name in italics
  - Add {Abstract} to name compartment





Interfaces

- Interface
  - Defines a set of methods and fields
- Classes should provide implementation for all the methods in the interface

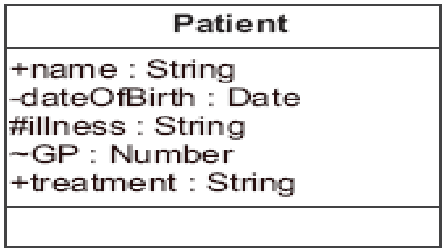


Java & UML

Mapping UML to Java

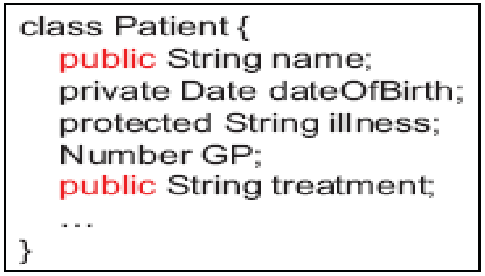
- Fact
  - In general there is not a one-one mapping from UML to Java or any other OO language
- Why?
  - UML was designed to be language independent
- So …
  - Examples of mappings for particular cases do not constitute a generalization

Example mapping attempt



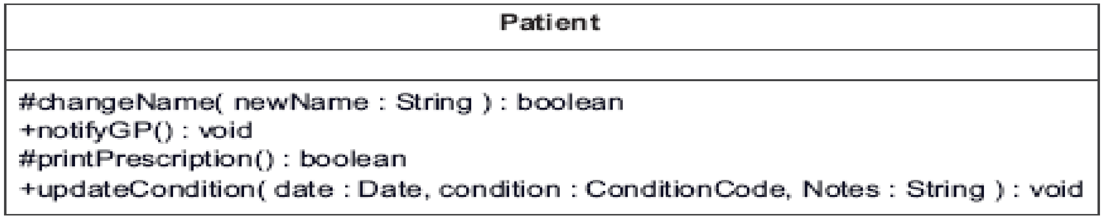
Patient

+name : String
-dateOfBirth : Date
#illness : String
~GP : Number
+treatment : String



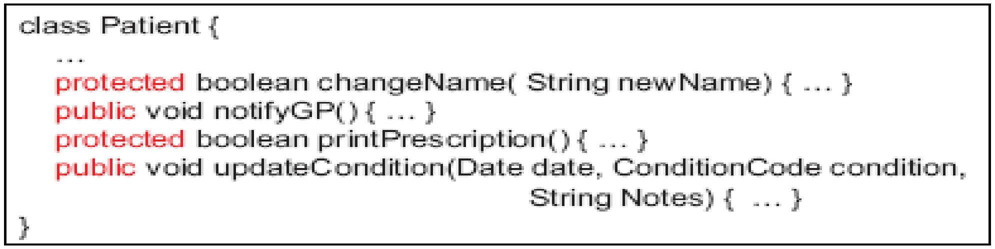
```
class Patient {
    public String name;
    private Date dateOfBirth;
    protected String illness;
    Number GP;
    public String treatment;
    . . .
}
```

Example mapping attempt 2



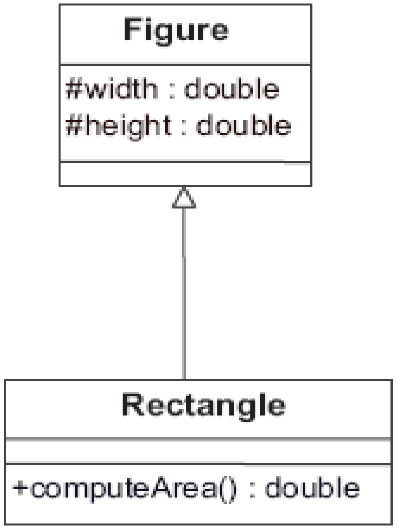
Patient

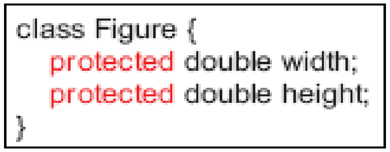
#changeName(newName : String) : boolean
+notifyGP() : void
#printPrescription() : boolean
+updateCondition(date : Date, condition : ConditionCode, Notes : String) : void



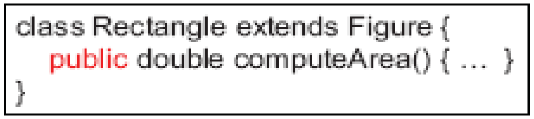
```
class Patient {
    ...
    protected boolean changeName( String newName) { ... }
    public void notifyGP() { ... }
    protected boolean printPrescription() { ... }
    public void updateCondition(Date date, ConditionCode condition,
                                                    String Notes) {  ... }
}
```

Another mapping -generalisation example



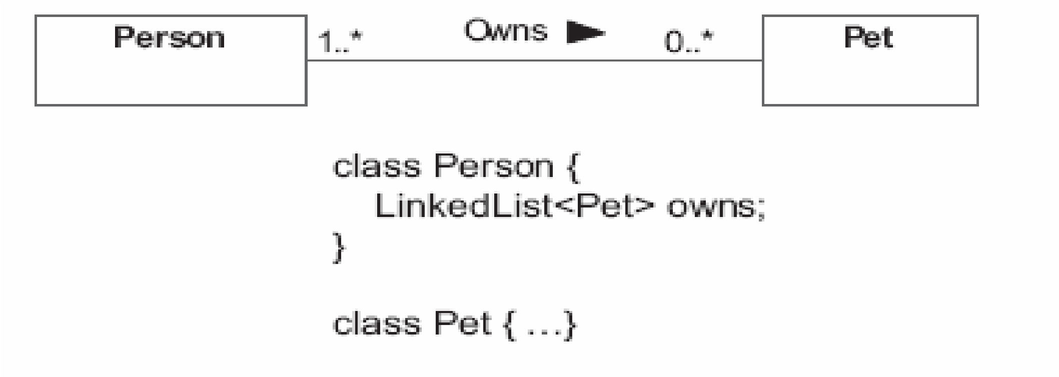


```
class Figure {
    protected double width;
    protected double height;
}
```



```
class Rectangle extends Figure {
    public double computeArea() { … }
}
```

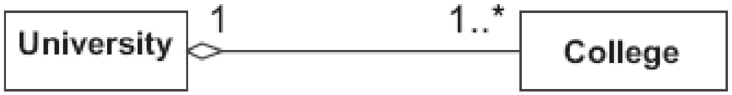
Association example



```
class Person {
    LinkedList<Pet> owns;
}

class Pet { ...}
```

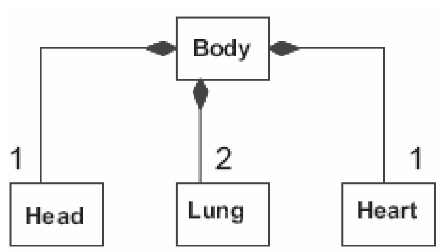
- What about cardinality?
    - Provide a runtime mechanism to enforce it!

Aggregation example



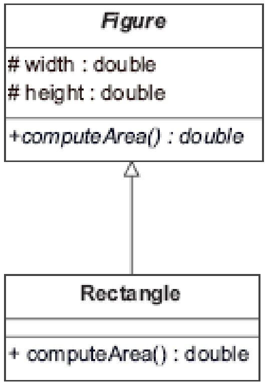
```
public class University {
    List colleges;
}

class College {
    University univ;
}
```

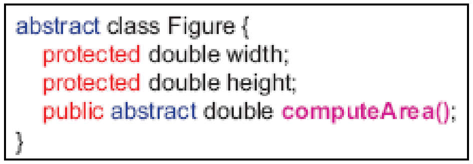
Composition example



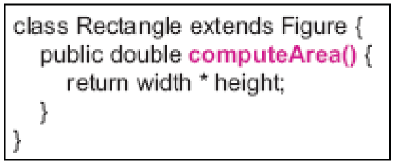
```
class Body {
    Head head;
    Lung[ ] lungs = new Lung[2];
    Heart heart;
}

class Head {
   Body body;
}

class Lung {
    Body body;
}

class Heart {
    Body body;
}
```

Abstract classes



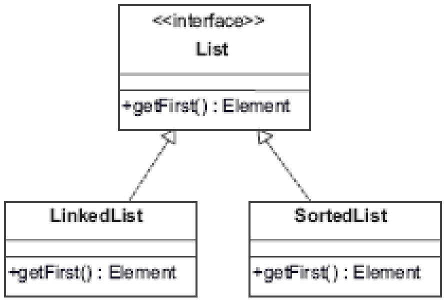


```
abstract class Figure {
   protected double width;
   protected double height;
   public abstract double computeArea();
}
```



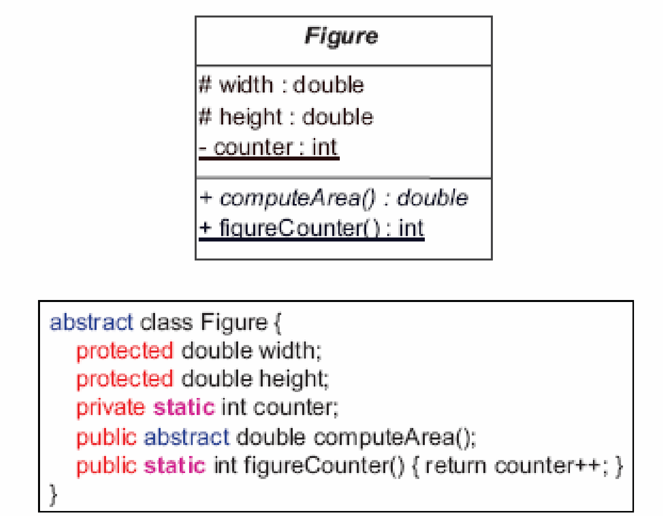
```
class Rectangle extends Figure {
   public double computeArea() {
      return width * height;
   }
}
```

Interfaces



```
interface List {
    public Element getFirst();
}

class LinkedList implements List {
    public Element getFirst() { … }
}

class SortedList implements List {
    public Element getFirst() { … }
}
```

Static members



Figure

# width : double
# height : double
- counter : int

+ *computeArea() : double*
+ figureCounter() : int

```
abstract class Figure {
    protected double width;
    protected double height;
    private static int counter;
    public abstract double computeArea();
    public static int figureCounter() { return counter++; }
}
```