MechEng SE3
Lecture 7
Domain Modelling

Simon Gay (slides by Phil Gray)

17 February 2010

This week's supplementary reading

Zero Balances and Zero Responsibility

Michael Bolton

http://www.developsense.com/essays/zero.html

Where to go for more

- Sommerville, 7th & 8th editions. Section 8.4, Chap 14 (especially 14.1 and 14.2.3)

What is object-oriented programming?

Imagine that we want to work with shapes: squares, circles, rectangles etc.

Calculating the area of a shape is important.

In a programming language, we can define functions to calculate the areas of various shapes.

Shape calculations in Python

```
def squareArea(side):
    return side*side

def rectangleArea(width,height):
    return width*height

def circleArea(radius):
    return pi*radius*radius
```

Within a program, `squareArea(4)` evaluates to `16`.

Shapes in Java

```
class Square {
   int side;

   Square(int x) {
      side = x;
   }

   int area() {
      return side*side;
   }
}
```

```
class Rectangle {
   int width,height;

   Rectangle(int x, int y) {
      width = x;
      height = y;
   }

   int area() {
      return width*height;
   }
}
```

Shapes in Java

Within a program:

```
Square s = new Square(4);
```

and then `s.area()` evaluates to `16`.

Shapes in Java

Other properties of shapes, for example position, colour,… can be stored in the same way as the lengths of the sides.

Other functions (methods) can be defined in the same way as `area`.

We can go further by explicitly showing that `Square`, `Circle`, `Rectangle` have something in common: they are all shapes.

Shapes in Java

```
abstract class Shape {
   int area();
}
```

```
class Square extends Shape {
   int side;

   Square(int x) {
      side = x;
   }

   int area() {
      return side*side;
   }
}
```

```
class Rectangle
   extends Shape {
   int width,height;

   Rectangle(int x, int y) {
      width = x;
      height = y;
   }

   int area() {
      return width*height;
   }
}
```

Shapes in Java

Now, if `s` is any object of class `Shape`, we can use `s.area()` and the area will be calculated by the method definition in the particular class that `s` belongs to.

Shapes in Java

Finally, imagine that a picture contains a list of shapes:

```
class Picture {
   List<Shape> shapes;

   Picture(List<Shape> sh) {
      shapes = sh;
   }

   int area() {
      int a = 0;
      for (Shape s : shapes)
         a = a + s.area();
      return a;
   }
}
```

A little history: programming languages

- Simula
  - Late 1960s
  - a language for developing simulations
  - encapsulation
- Smalltalk
  - 1970s
  - GUI concepts
  - Strictly object-oriented
  - untyped
  - Garbage collection
- C++
  - 1980s
  - Extensions to existing GPL
- Java
  - 1990s
  - typed
  - Not an extension of previous language
  - Features for web-based applications
- C#
  - 2000
  - Designed for service-based distributed applications

A little history: modelling

- Object-oriented modelling grew up in the 1980s
  - Grady Booch, 1982, “Object-Oriented Design”
  - Jim Rumbaugh, OMT (object modelling technique)
  - Ivor Jacbosen, Objectory Method
- UML
  - Booch & Rumbaugh combine in 1994, later joined by Jacobsen
  - The Three Amigos
  - Formed The Rational Corporation
    - Recently absorbed into IBM
- UML now managed as a standard by OMG
  - UML 1.0 in 1997
  - Now UML 2.0

Objects and Classes

- An object is
  - a computational entity which
    - provides services
    - with which other entities may interact
    - typically, the service consumer sends a message (requesting the service) to the provider object
  - possesses
    - State
    - Behaviour
    - Identity

Classes

- A class
  - Describes a set of equivalent objects
    - hence, operates like a data type for objects
    - typically, every object in a system belongs to a class
  - Provides a useful way of representing and implementing the shared state and behaviour of the objects that it describes
    - E.g., object creation method

Sommerville on Objects

- Object
  “is an entity that has a state and a set of operations on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects (clients) that request these services when some computation is required.” (section 14.1)

Why Objects?

- Encapsulation
  Objects can operate as software modules that organise data and behaviour into meaningful associations
- Message-based Invocation
  Helps to make code adaptable and reusable, since
  - receivers don't have to know about senders
  - senders need only know about message *protocol* for receivers
  - Hence message protocol is a well-defined *interface* between senders and receivers
- Data Hiding
  Data hiding helps to manage complexity, since programmer can control the data that objects are allowed to do manipulate

Data Hiding

- Object technology (e.g., object-oriented programming languages and databases) provide mechanisms for controlling access to attributes
- So, attributes can be private to the owning object or to some restricted set of objects
- An sender can query or modify an attribute of a receiver via appropriate messages, even if the attribute is private

Sommerville on Classes

- “Objects are created according to an object class definition. An object class definition is both a type specification and a template for creating objects. It includes declarations of all the attributes and operations that should be associated with an object of that class.” (section 14.1)

Why Classes?

- Inheritance
  - Provides a mechanism for sharing attributes and operations,
  - a form of reuse
- Polymorphism
  - Allows different effects of messages to be determined by the receiver, without the sender having to know about the differences
  - And, can add new (sub)classes of objects without having to change the interface between sender and receiver

Method Classification

- There are three different types of method
- ***constructor***
  - » It will be called when an object is created
  - » It provides guaranteed initialisation
- A ***transformer***
  - » which changes the internal state of the object
- An ***accessor***
  - » which uses the internal data without changing it

("method", "message" and "operation" mean the same thing)

Where do classes come from?

- once we have a set of use cases we begin a process of analysis to identify candidate data that participates in the use cases
- the types for this data will become a set of classes
- during design we will
  - refine these class definitions
  - add additional classes as necessary

Analysis vs Design

- In developing our understanding of the classes needed for a proposed system, there are two fundamental stages: analysis & design
- According to Larman (Applying UML & Patterns, section 1.4):
  - ***Analysis***: “During object-oriented analysis, there is an emphasis on finding and describing objects (or concepts) in the problem domain.
  - ***Design***: “During object-oriented design, there is an emphasis on defining software objects and how they collaborate to fulfill the requirements.”
- This lecture focusses on (early) analysis

Class Notation in UML

- A class is represented by a rectangle with three compartments.
  - The name of the class
  - The class attributes
  - The class operations.
- The attributes are listed in a language dependent way. UML is not fussy about the notation used.
- The operations are written in the same language style as the attributes.

Naming Classes and Instances

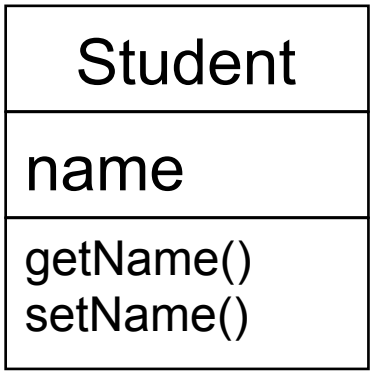
- AClass *a class*
- :AClass *an anonymous instance of AClass*
- a1:AClass *a named instance of AClass*
- ASuperClass::AClass

  refers to AClass which inherits from ASuperClass

Levels of Class Description

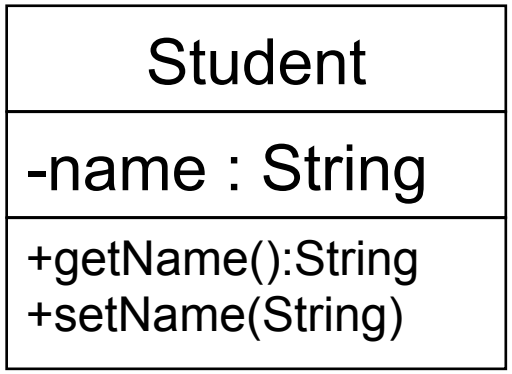
- Consider class that represents student information
- Analysis

  Student has a name attribute
- High level design



Levels of Class Description

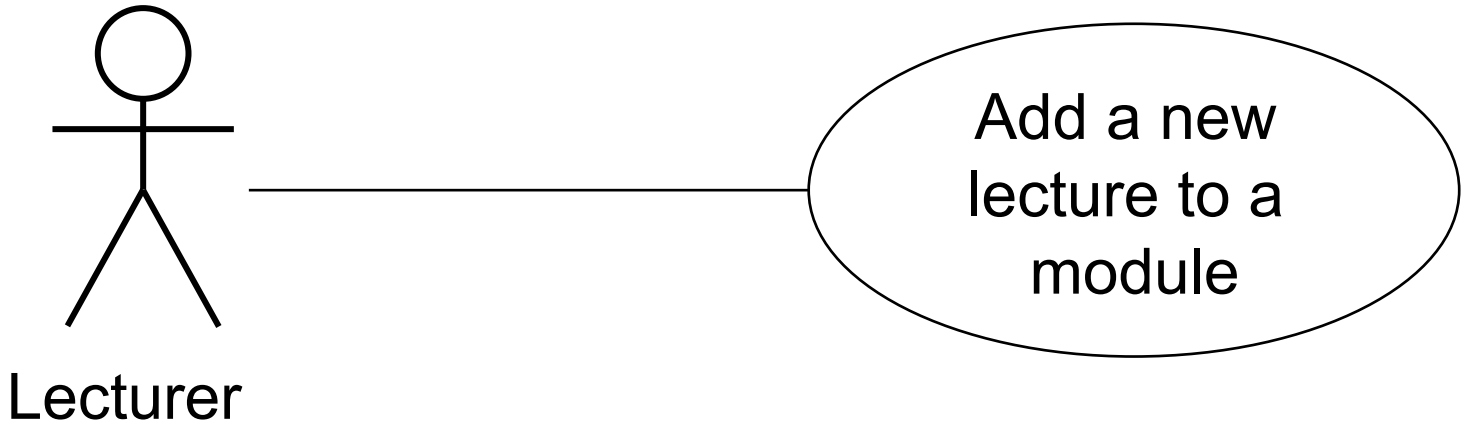
- Detailed design
  - Attribute held as private member?
  - Or implemented as instance of a new “Name” class?



- Implementation

```
Class Student {
        private String name;
        public String getName();
        public setName();
}
```

An Example Use Case: Add New Lecture to Module



An Example Use Case: Add New Lecture to Module

- The lecturer selects the required module from the relevant course and adds a new lecture to the existing list of lectures for that module. The details of the lecture are completed by the lecturer.

An Example Use Case: Add New Lecture to Module

- The ***lecturer*** selects the required **module** from the relevant **course** and adds a new **lecture** to the existing **list of lectures** for that **module**. The **details of the lecture** are completed by the ***lecturer***.

CRC: Finding Classes

- CRC=Class-Responsibility-Collaboration
- Technique for capturing initial class information
- Use index cards, one per candidate class
- Keep descriptions informal and simple

CRC Format

Class Name	
Responsibilities	*Collaborations*

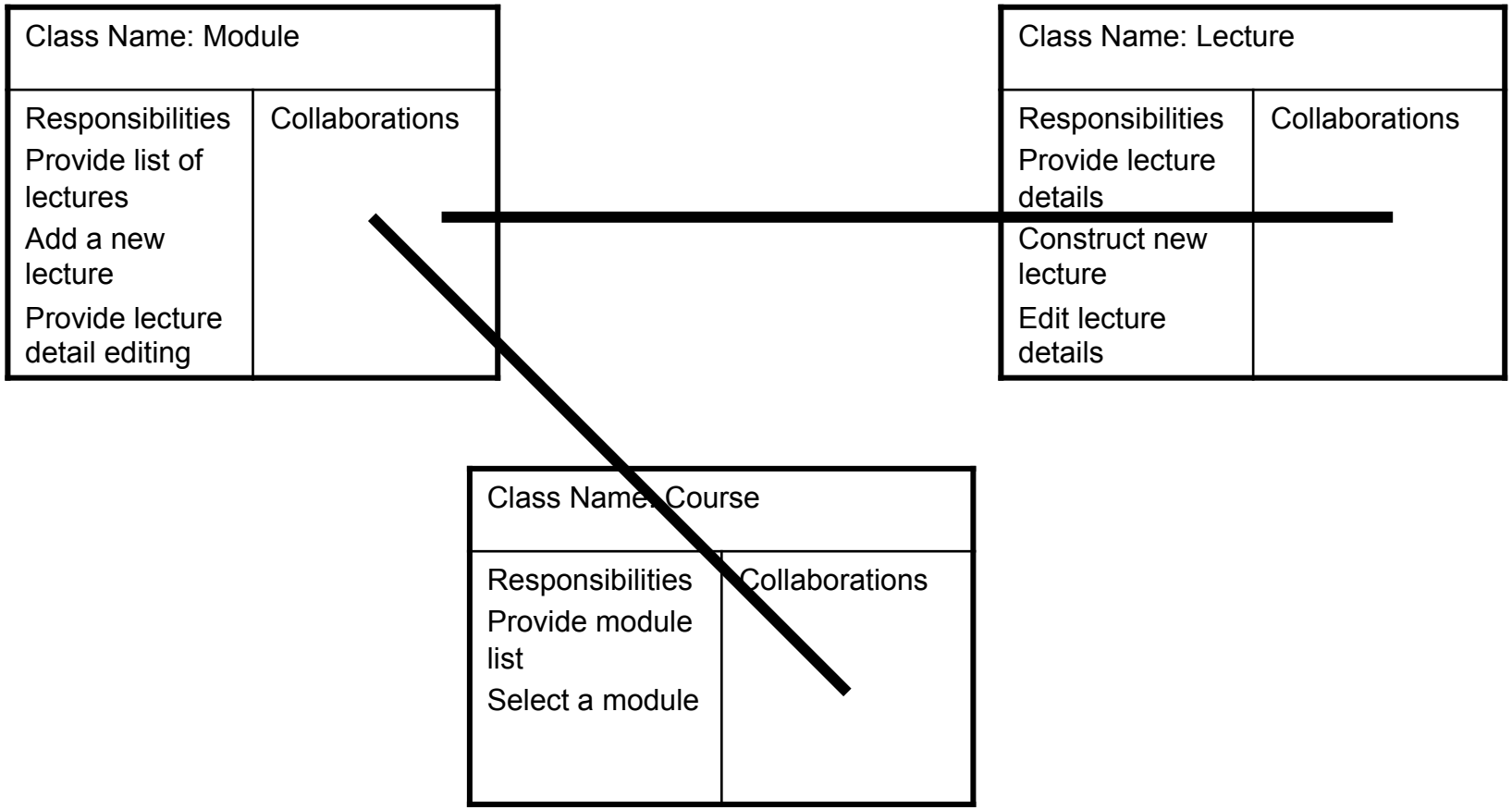
CRC for Campaign

Class Name: Module	
Responsibilities Provide list of lectures Add a new lecture	Collaborations Course Lecture

Class Responsibilities

- Doing
  - Doing something itself
  - Starting an action in another object
  - Coordinating other objects
- Knowing
  - Knowing private data
  - Knowing about related objects
  - Knowing how to derive a value

Finding Collaborations



Elaborating the Classes

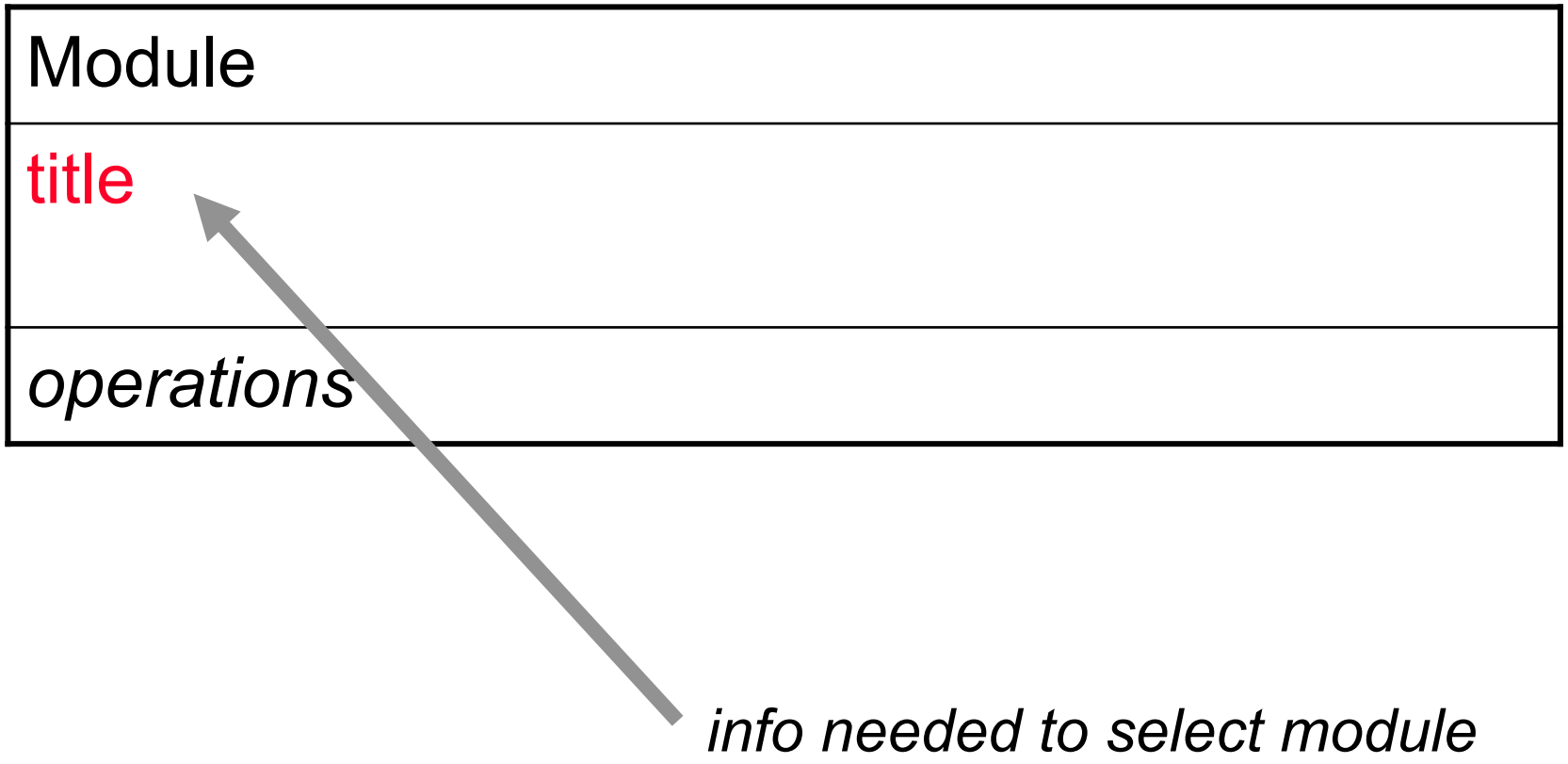
- Each class has
  - A set of attributes
  - A set of operations
  - A set of associations
- We represent this information in a class diagram

Class Diagram Structure

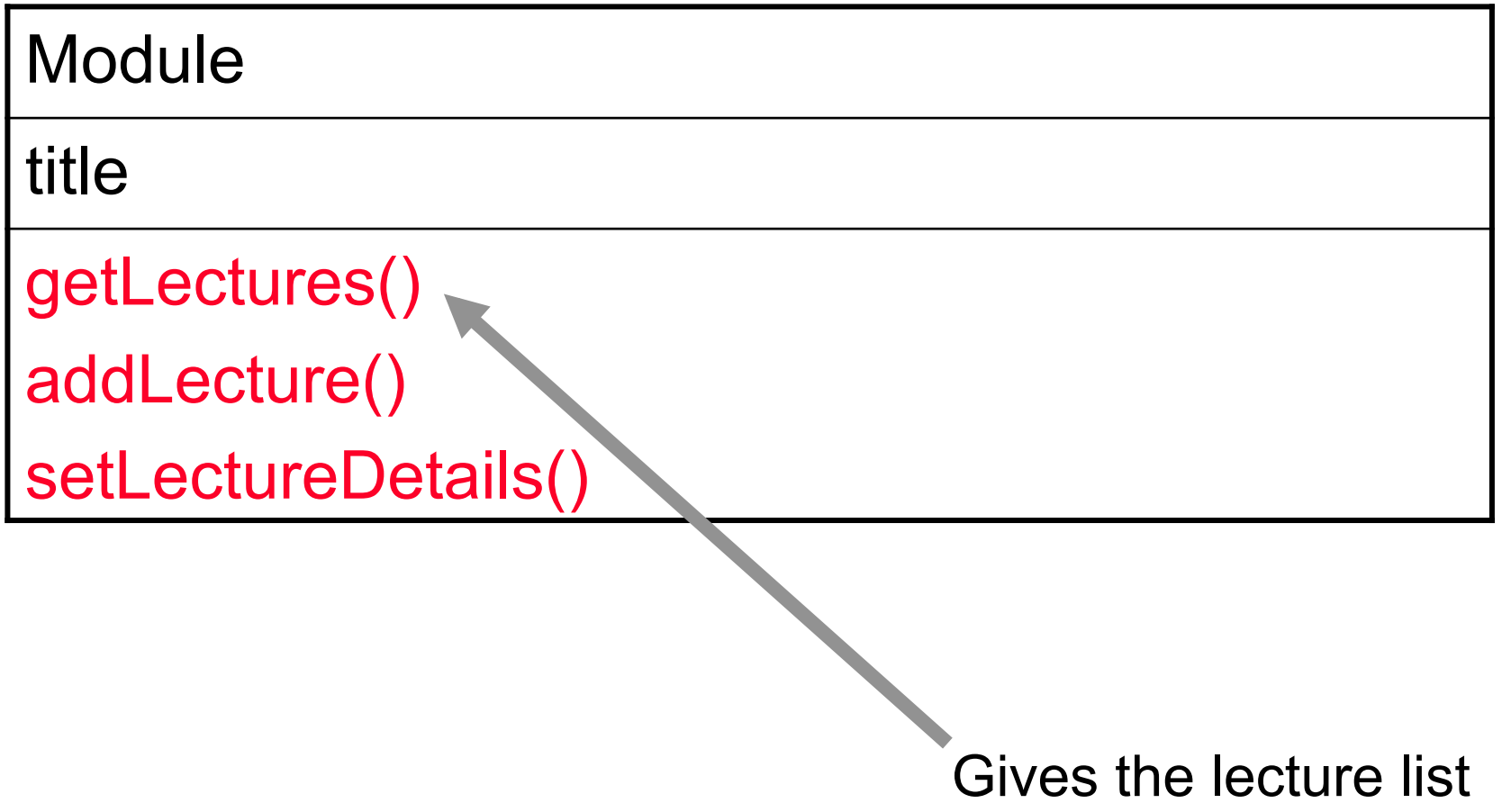
Naming the Class

Module
attributes
operations

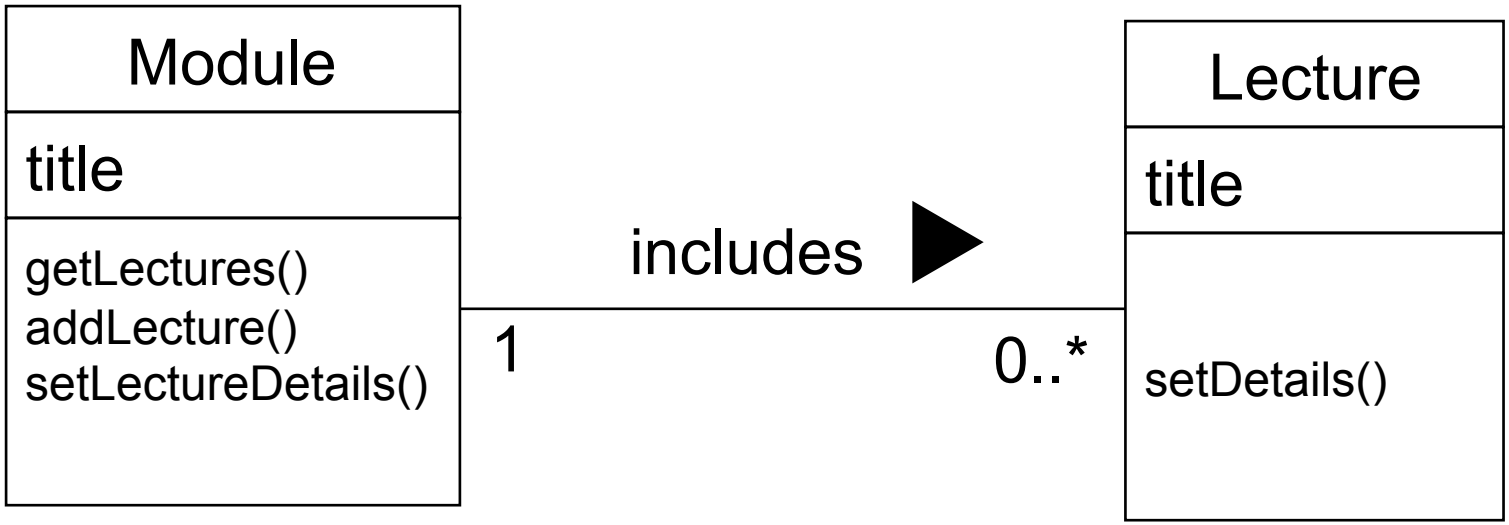
Attributes



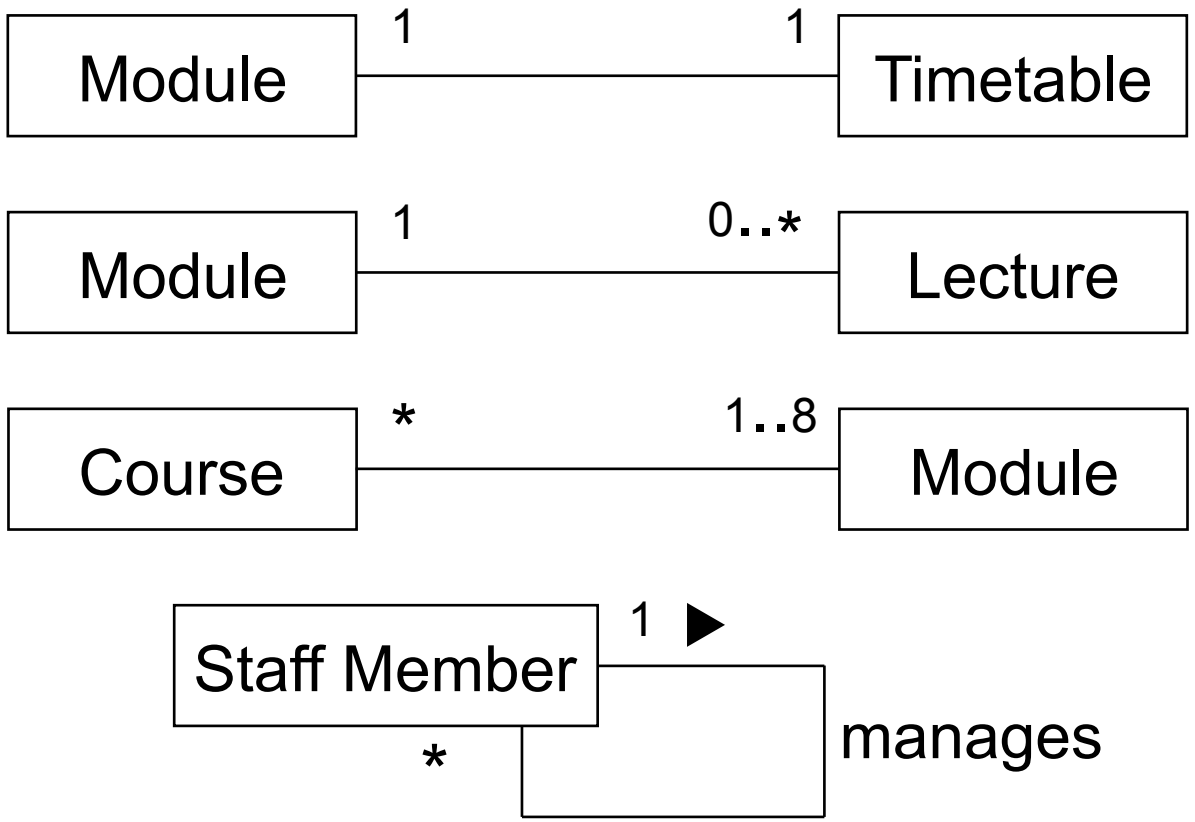
Operations


Module
title
getLectures()
addLecture()
setLectureDetails()
Gives the lecture list

Associations



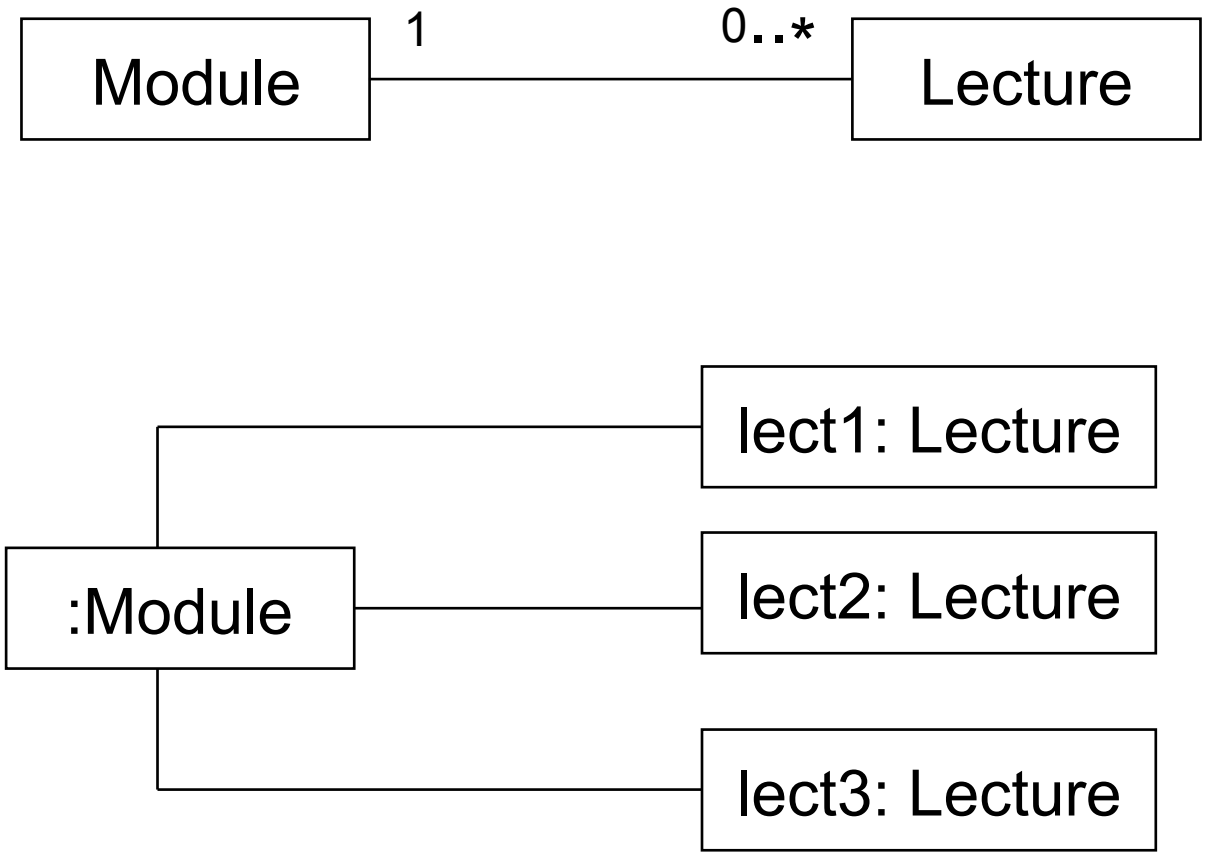
Kinds of Associations



Representing Associations between Instances and between Classes

- Sometimes it's useful, especially in early analysis stages, to examine associations at the level of object instances
- Later, as the classes are being identified and specified, associations are represented at the class level

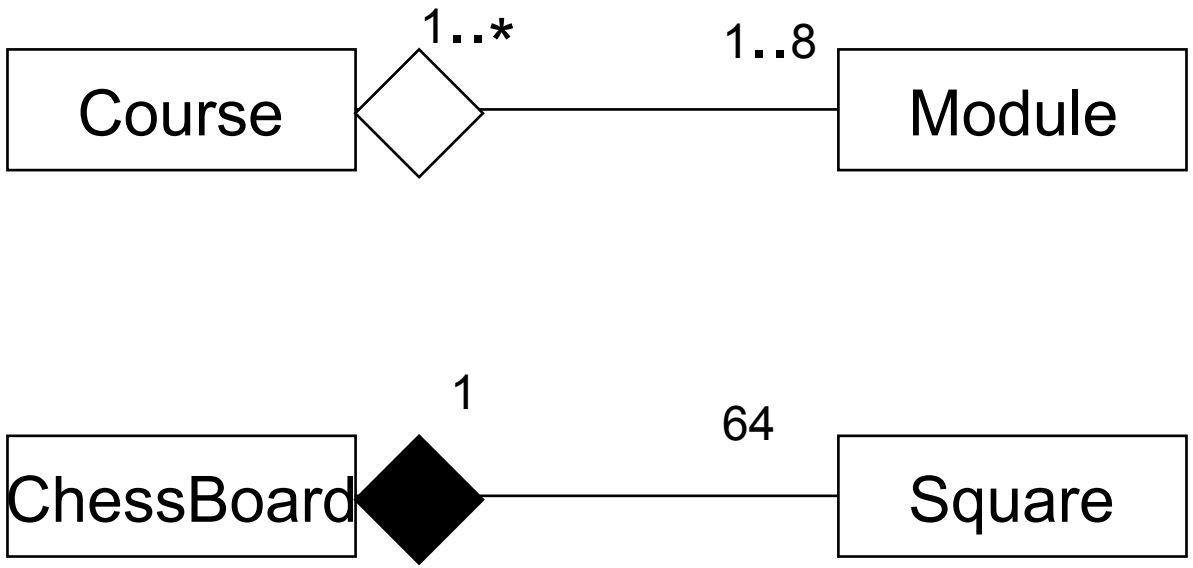
Representing Associations between Instances and between Classes



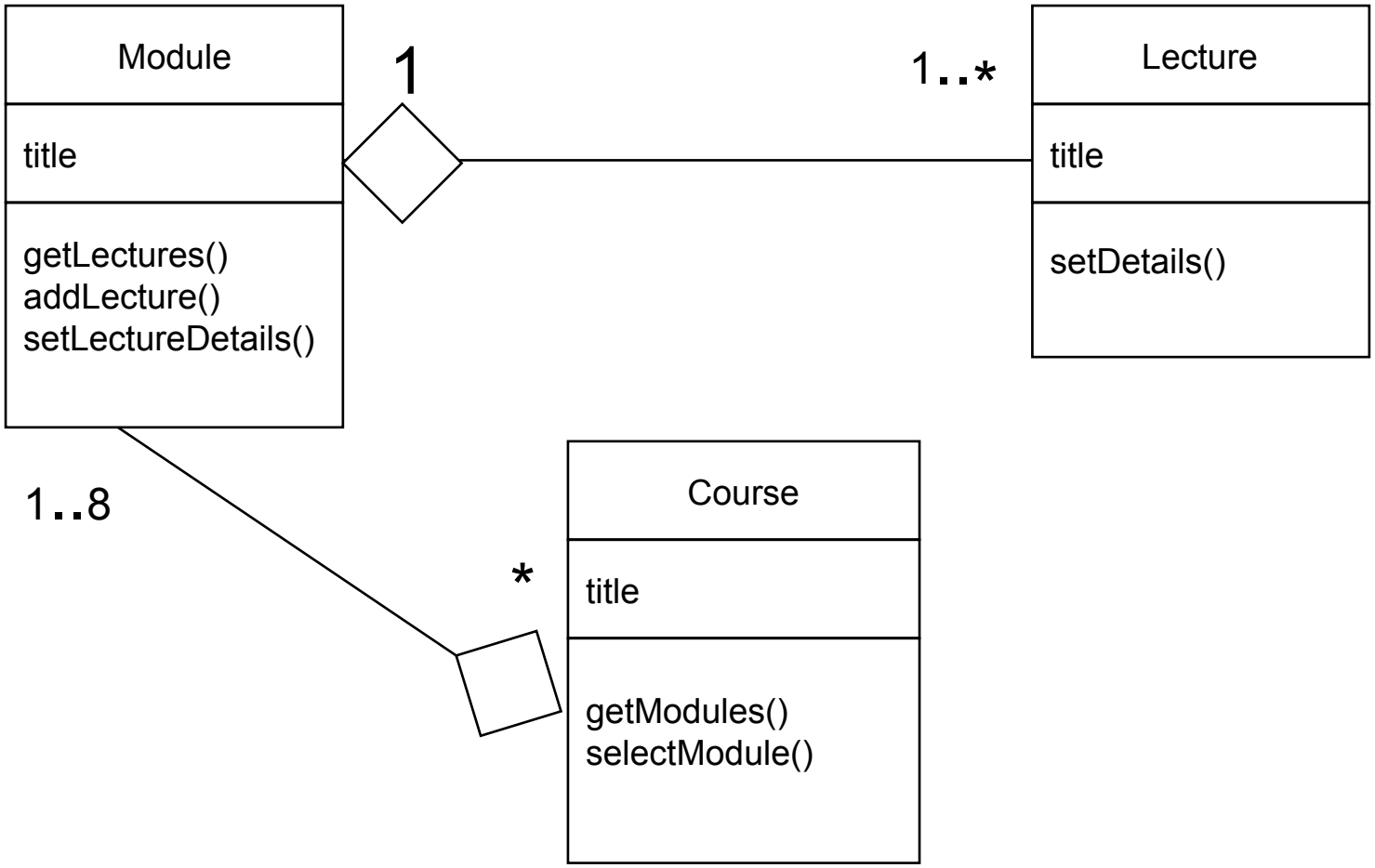
Aggregates and Composites

- Aggregate
  - an association in which one object is a part of another
  - Shown by an open diamond at aggregation end
- Composition
  - a strong form of aggregation
  - If the composite is deleted or copied, so must its parts be
  - Shown with a filled diamond at the composite end

Examples



Class Diagram from Example Use Case



Finding Attributes

- Attributes should be atomic or simple
  - boolean, data, number, string, time
- A proposed attribute may turn out to be an association with another object
- Derive attributes from operations
  - If you have a "provide name" operation, then you will need a name attribute or an association with a name

Finding Attributes in Our Example

- Consider setLectureDetails
- The use case as it stands does not specify the attributes that make up the lecture details
- Thus, the analysis may raise questions about the completeness or clarity of the use case description
- E.g., perhaps need to go back to client or to domain stakeholders to determine what is needed as lecture attributes

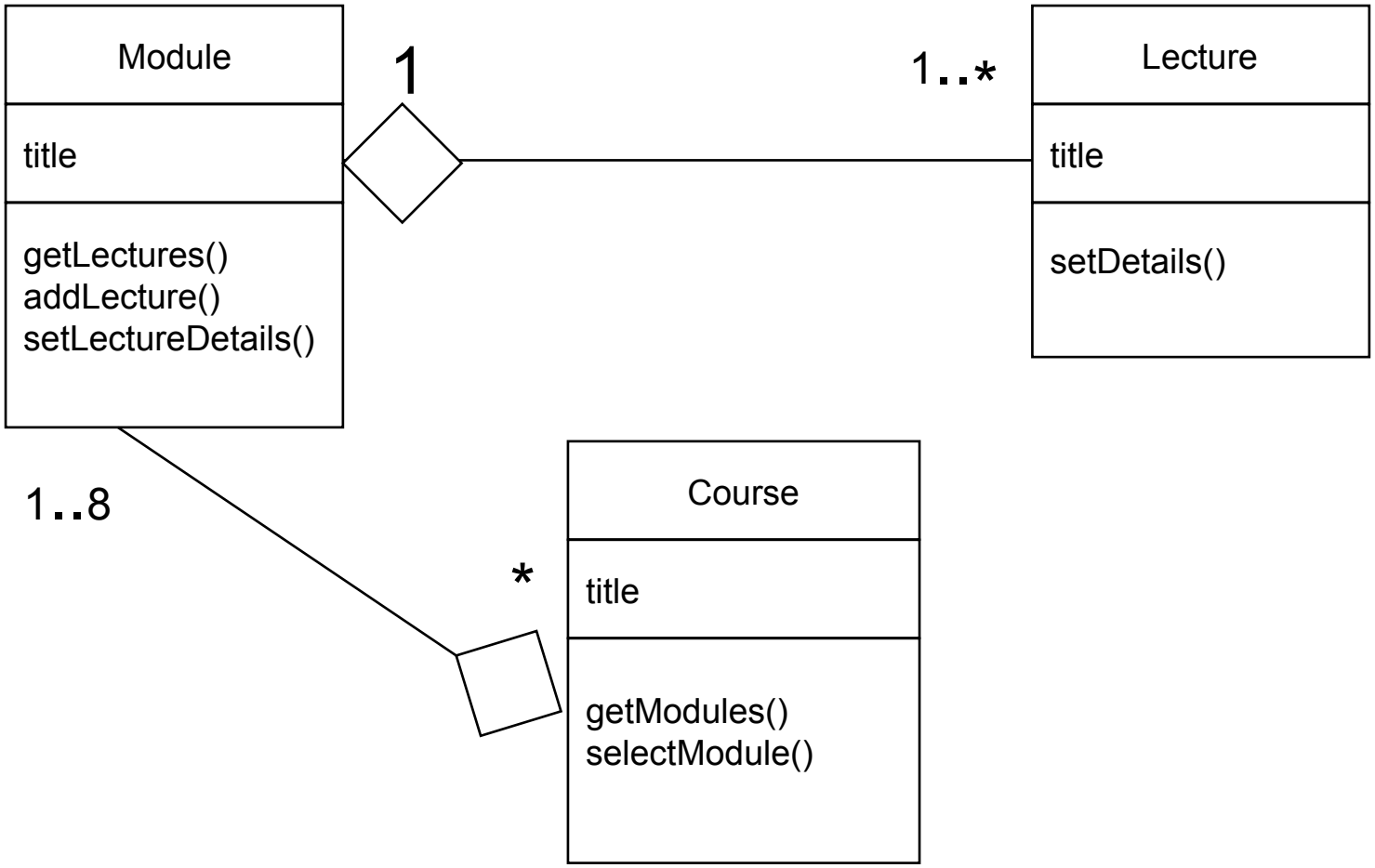
Referring to associated objects

- At the analysis stage, you do not need to
  - » indicate how an association will be implemented
  - » decide how to represent a collection
- That will be determined later in the design process

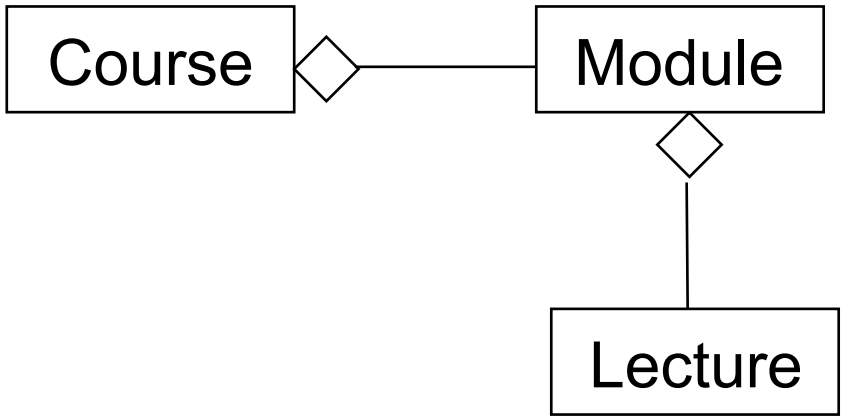
Another Example: Assign Lecturer to a Module

Actor Action	System Response
	Lists titles of all courses
Selects course	Lists titles of all modules for that course
Selects module	Displays all staff members not already allocated to a module
Selects staff member to be assigned to this module	Displays confirmation

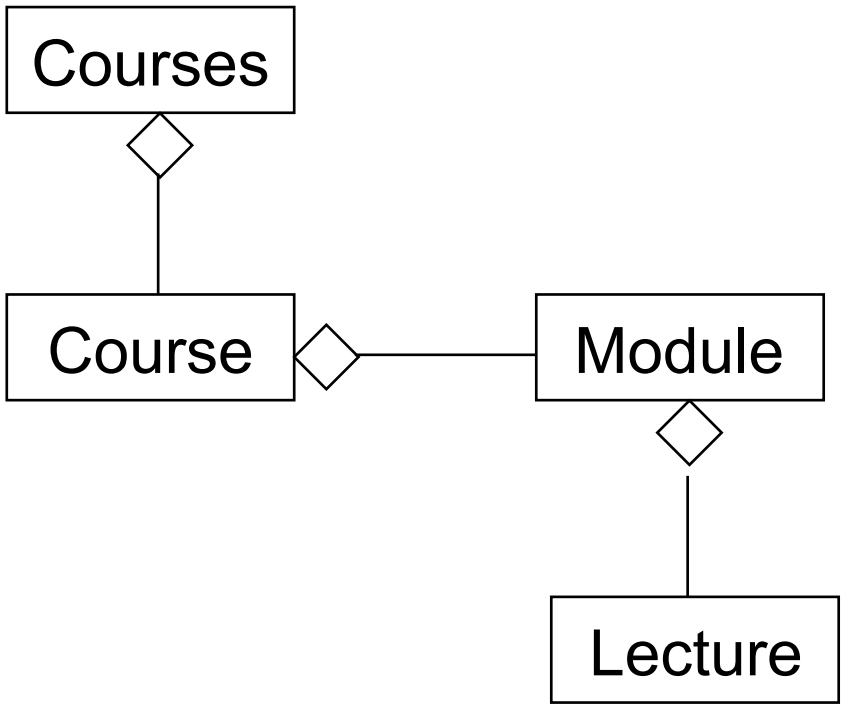
Class Diagram from First Use Case



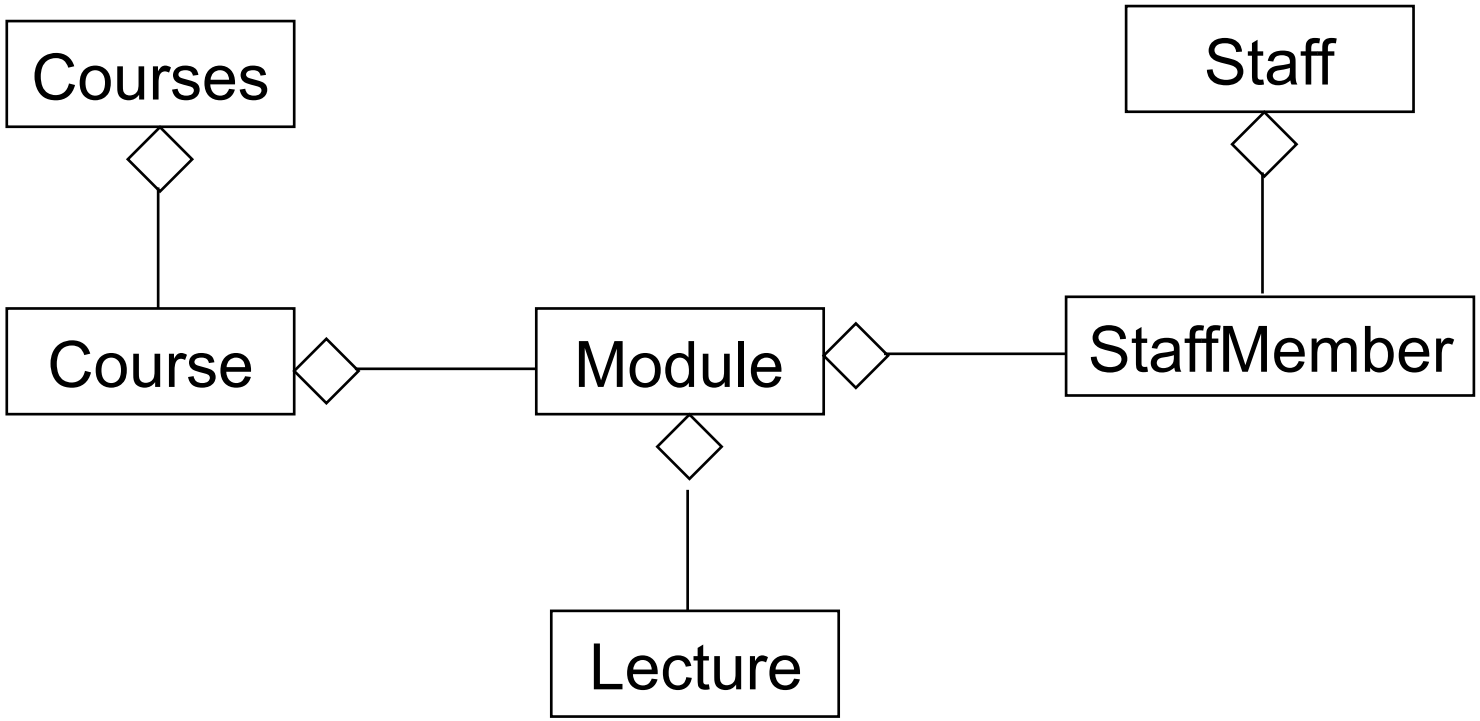
Refining the Class Diagram (1)



Refining the Class Diagram (2)



Refining the Class Diagram (4)



Visibility

- Visibility is related to the data hiding features of class members
  - + public (external code can use)
  - # protected (only child classes can use)
  - - private (internal to class)
- Private is normally only useful at the implementation level
- Protected is related to inheritance and will be discussed later

Visibility

- Something to think about:

- How is visibility handled in Java?
- Compare it to the visibility feature in UML