CSCI E-97
Lecture 2
Introduction to the UML

January 31, 2017

Outline

- Announcements
- Assignment 1
- Comments on Writing
- Modeling
  - Cover basic concepts of object models
  - Introduce notation from the Unified Modeling Language (UML)

Announcements

- Starting sometime this week (2/1), course videos and the course website will be password-protected. Both will be available only to registered students (use your Harvard ID and PIN)
- There is a course discussion forum on Piazza. Make sure to participate.

  https://piazza.com/class/iy7pgel326w1kg?cid=8

- Please add your bio and picture to the people tab of the course web site.

- Reading: chapters 4 and 5 of UML Distilled

Assignment 1

- Assignment 1 is due next week, Monday (2/6) at midnight.
- Submit questions to discussion forum on Piazza.
- The Assignment 1 grade sheet is posted with the assignment 1 materials.
- Include class and method level JavaDoc and inline comments to explain logic where appropriate.
- Consider and implement exception handling.
- Please check to make sure that your project can be compiled and run from the command line.
- Submit your zip file to the assignment 1 dropbox on class web site.

Notes on Writing Technical Documents

- Why do we write design documents?
  - to communicate
  - to improve our own understanding
  - to improve the quality of our software
- The three questions we have to ask ourselves are
  - Who is the intended audience?
  - What is the topic?
  - What is it that I know that I can say?
- Once you have these squared away, you can begin to **design** your document
- Fortunately, there are principles and patterns that can help with your writing

Elementary Principles of Composition – 1

- These follow the advice from Chapter II of Strunk & White (2000), *The Elements of Style*, 4th edition, Longman

12. Choose a suitable design and hold to it.

- It is amazing how closely **the process of constructing a system matches the process of constructing a document** to describe that system
- In a free-form world, you can **take the approach that you'll answer the three questions on the previous slide** and then design your document. In reality we are often required to follow a 'standard' template.
- The authors say "...in most cases, **planning must be a *deliberate* prelude to writing**." and refer to this rule as the first principle of composition

Elementary Principles of Composition – 2

13. Make the paragraph the unit of composition.

- "As a rule, **begin each paragraph either with a sentence that suggests the topic or with a sentence that helps the transition**. If a paragraph forms part of a larger composition, its relation to what precedes, or its function as a part of the whole, may need to be expressed."
- **Apply this recursively**; that is, apply this to a chapter by having an initial paragraph introduce the topic or topics of the chapter, and apply this to an entire document by having an introductory chapter that introduces the topics to be covered and describes how the rest of the document is structured

Elementary Principles of Composition – 3

- 14, 15, and 16 require us to make a commitment to being clear about the details.

14. Use the active voice.

- It's much clearer, and more accurate, to write

"When the method detects an error, it throws an XYZ exception that the client uses to display a message to the user."

than to write

"When an error is detected a message is shown to the user."

15. Put statements in a positive form.

- In our kind of writing, use simple declarative sentences.

16. Use definite, specific, concrete language.

Avoid vague expressions. Imagine a use case where the writer begs the question of what happens by writing

"The user creates a profile"

with no details about what goes into the profile

Modeling

- What kind of model are we trying to achieve?
- If one believes that **design follows from requirements** (and redesign follows from requirement changes) then one view is to
  - start with a model of the problem domain – this is referred to as domain modeling, and Fowler uses the term ***conceptual perspective***
  - then create a model for the *software* itself – Fowler uses the term ***software perspective***
- In domain modeling, one finds the things (entities) that are of interest to the business user (e.g., a Customer, an Order) and captures constraints on the classes and their relationships
- In software modeling, one identifies the implementation elements that support the domain (e.g., a Factory for creating objects, a Mediator for coordinating the interactions of multiple other objects)

Simplicity in Modeling: Occam's Razor

- **Occam's Razor** (also spelled **Ockham's Razor**), is a principle attributed to the 14th-century English logician and Franciscan friar, William of Ockham. It forms the basis of **methodological reductionism**, also called the principle of parsimony or law of economy.
- In its simplest form, Occam's Razor states that one should make no more assumptions than needed. Put into not quite everyday language, it says

  Given two equally predictive theories, choose the simpler.

- In engineering, people tend to apply the variant that says "Entities should not be multiplied beyond necessity“ (really!), which is especially fitting for modeling software systems
- “Keep things as simple as possible, but no simpler”, Albert Einstein
- Less is More… Keep it Simple

What Does an Object Model Comprise?

- As defined by the Object Management Group, **a core object model needs to address the following set of concepts**:

  - **Objects, operations, types, and subtyping**. An object can model any kind of entity, for example a person, a ship, a document, a department, a tuple, a file, a window manager, or a lexical scanner.
  - **A basic characteristic of an object is its distinct identity**, which is immutable, persists for as long as the object exists, and is independent of the object's properties or behavior.

- Three things are implicit in this description:
  - There's a distinction between a type and an instance of a type (an object)
  - An object has properties (state) and behavior
  - Some things are objects and some things aren't

So What Does an Abstract Object Model Address?

- What is an object (state and behavior works for most people)
- Are there non-object data types (e.g., integers, reals, strings, booleans)
- What is a type?
- What is a class?
- What's the difference between a type and a class?
- How are operations on a type defined?
- Is an attribute a first-class object?
- Is an association a first-class object?
- Are inheritance and polymorphism supported?
- What kind of inheritance is supported?

Some Short Answers

- An object is an encapsulation of state and behavior
- Almost all object systems (except SmallTalk) differentiate between things that are objects and things that are not (e.g., integers, reals, strings, booleans)
- A type is a predicate specifying membership (an object is a member of a type or not – an object can be a member of many types, because of inheritance)
- A class is an implementation of a type. A type can have many different implementations. This latter statement is especially true in distributed systems
- Every object system defines a signature for operations (aka methods or class functions)
- Generally, attributes and associations are not treated as first-class objects in any O-O language.
- The kind of inheritance supported varies widely

The Unified Modeling Language (UML)

- For many years, **Rumbaugh**'s Object Modeling Technique(OMT), **Booch**'s O-O methodology, and **Jacobsen**'s Objectory methodology were the three primary, but competing, O-O methodologies
- The UML combines the concept and the notation from these three into one unified object model
- The UML has been adopted as a standard for Object-Oriented Analysis and Design by the Object Management Group (OMG)
- Note: The most recent UML specification is available at www.omg.org

How Does One DO Modeling?

- One begins by understanding the requirements, however they are expressed (some might argue that domain modeling is part of the elaboration of the requirements)
- Some things pop out immediately from reading the requirements, and become things in the object model – some as types, some as properties of types
- A sense of behavior begins to take shape – the important step is assignment of behavior to objects
- There are three principles that can be applied from the very beginning
  - Divide and Conquer
  - Separation of Concerns
  - Abstraction

Three Enabling Principles

- ***Abstraction*** is about finding 'the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries relative to the perspective of the viewer' [Booch, Object Oriented Analysis and Design with Applications, 1994 – see also Buschmann et al, Pattern-Oriented Software Architecture, 1995]
- ***Divide and Conquer*** is about breaking a large complex problem into smaller problems that are more easily addressed
- ***Separation of Concerns*** is about separating different responsibilities among different types. For instance, presentation is not a responsibility of a domain object; to be more specific, a Customer object does not know how to present itself – that responsibility is left to a presentation object. Similarly, a User object does not know how to read and write itself to a persistent store (a file or a database)

How Do the People Who Use the UML Use the UML?

- Fowler captures the range of uses quite well on page 2
- Use as a sketch
  - very informal, lots of scribbling on whiteboards and paper, maybe even some sketches using a tool
- Use as a blueprint
  - (forward engineering) Create a complete model and pass it along as the basis for subsequent implementation, perhaps even generating a skeletal implementation of all the classes
  - (reverse engineering) Take a snapshot of existing source code and create a model from it
- Use as a programming language
  - This is the dream of the Model-Driven Architecture folks
  - One can 'compile the spec' by using very sophisticated tools

Really Basic Concepts in the UML

- There are two terms from the *metamodel* of UML that surface from time to time

1. Classifier – this represents a group of things in the abstract model that have common properties (a *Class*, an *Activity*, etc)
2. Adornment – Classifiers can have different kinds of additional information attached to them. Two of these are *constraints* and *stereotypes*

Properties: A Full Description

- The most complete form of definition of any property allows specification of the following (only the name is required)
  - visibility
  - name
  - type
  - multiplicity (the number of occurrences of the feature)
  - default value (the value to use if one is not specified at construction)
  - {property string} (e.g., {read-only})

Naming Conventions for Classes and Features in This Course

- We will follow the same naming conventions for classes and features in the UML as we do for classes, data members, and methods in Java.
- Class names begin with a capital letter, and subsequent words in the name have initial capital letters (e.g., `NetworkNode`, `TextFileParser`, `FileReader`)
- Features of a class have names that begin with a lower-case letter; subsequent words within the name have initial capitals; for instance,

  `attributeA`

  `getAttributeA`(),

  `void setAttributeA(Type t)`
- Parameter names and variables follow the same rule as features of a class (e.g., `numberOfElements`)
- Primitive (non-class) types usually have lower-case names (e.g., `string`, `real`, `integer`, `boolean`, `float`, `double, date`)

Visibility

- The UML has a notation for specifying which features have public, private, protected, or package scope (names are preceded by a '+', '-', or '~', respectively)
- In most of our use of the UML, we will leave these marks off based on two assumptions:
  - that the operations are assumed to be public
  - that attributes are assumed to be private, and public accessors (get/set) are implicitly defined [if an attribute is read-only, only the get accessor is defined]
- We discuss other conventions we will follow in the class a bit later

Multiplicity for Attributes

- For attributes, the multiplicity is shown in square brackets
- For instance, we could write

  ```
  separatorCharacter : char [1] = "\t"
  ```

  for a single-valued attribute of a class
- The notation for multiplicity shouldn't be confused with the size of the attribute – here we just mean that there is exactly one value for `separatorCharacter`
- The example shows how we would write out an attribute in detail – the same fields of an association are defined differently in the UML, mostly through annotations of association links.

Defining Operations

- An operation defines a service that an object provides in response to a request
- The most complete form of definition of an operation allows specification of the following (only name is required)
  - visibility (public, private, protected, package)
  - name
  - parameter-list
  - return type
  - {property string} (e.g., {query})
- Generally, this could be expressed as

```
visibility name (parameter-list): return-type
  {property string}
```

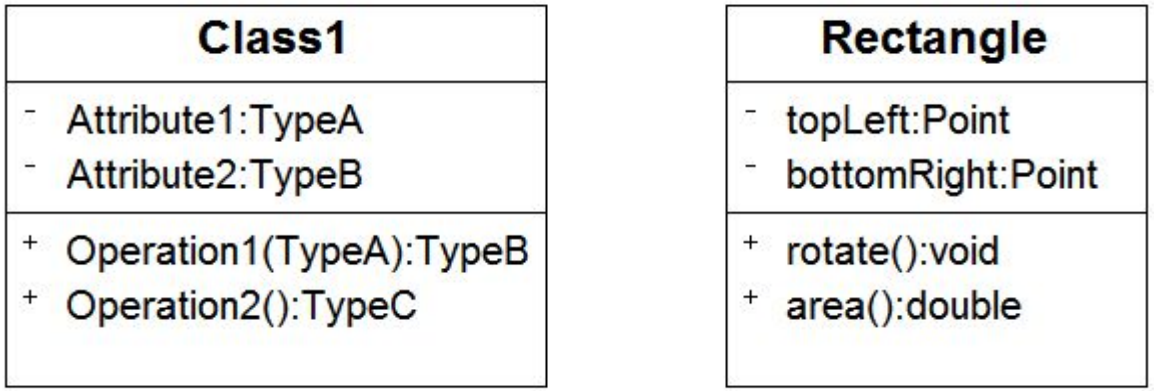
- For instance, we could write

```
+ getSeparatorCharacter() : char
```

  where the property string is not present

UML Notation for the Class Construct

- A class defines a set of properties (which express the state of the class) and operations (which express the behavior of the class)



- The diagram above shows each of the following in a separate box or compartment:
  - the name of the class (in bold)
  - the attributes
  - the operations

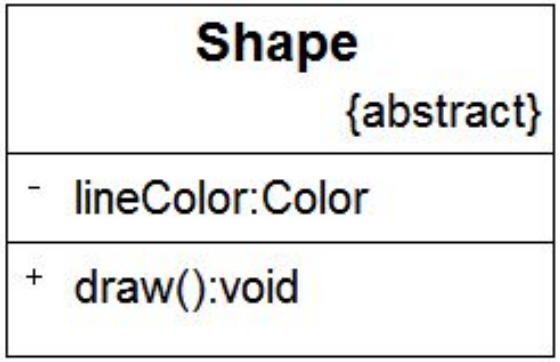
Our Conventions for Attributes

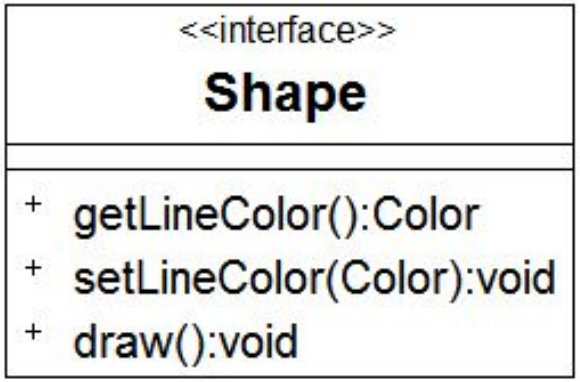
- We assume that attributes are private and have "getter" and "setter" accessors that are public. Hence we don't bother to list the accessors in a class diagram
- In the case of class `Rectangle` on the previous slide, there are two implicitly defined operations for the attribute `topLeft` that are not shown in the diagram:

```
public Point getTopLeft();
public setTopLeft(Point p);
```

Declaring Abstract Classes and Interfaces

- Often, we want to specify that a class is abstract, or that we have an interface rather than a class
- An abstract class is a class that cannot be instantiated directly.
- In the first case we specify the UML property of 'abstract' and in the latter we add a stereotype for an interface





Relationships Between Classes

- There are four kinds of relationships between classes that the UML addresses:

  Generalization – this is how inheritance gets defined

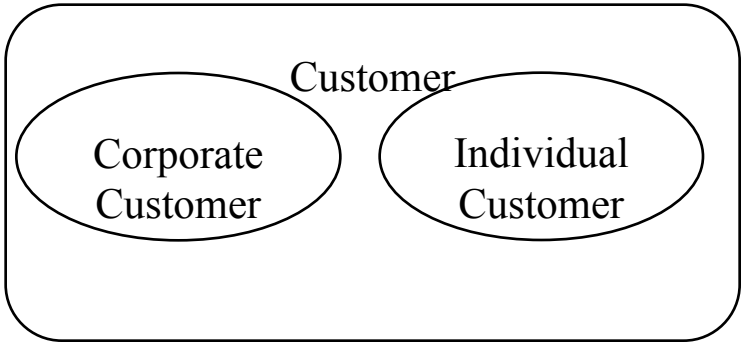
  Associations – this is for structural relationships

  Dependency – this shows how classes might depend on other classes

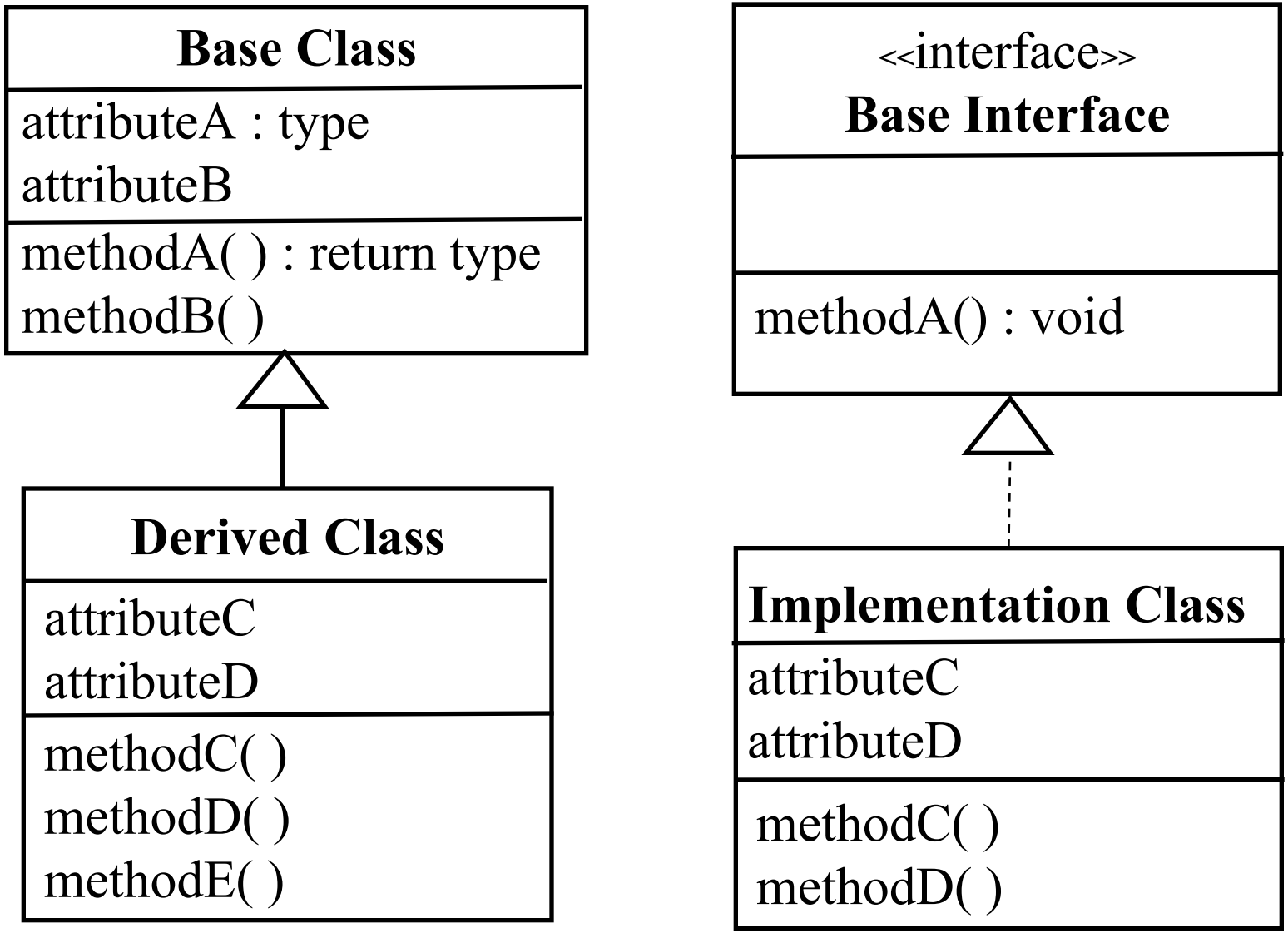
  Abstraction and Realization – this is for looking at refinements of definitions (e.g., an analysis view of a class and an implementation view).

Generalization

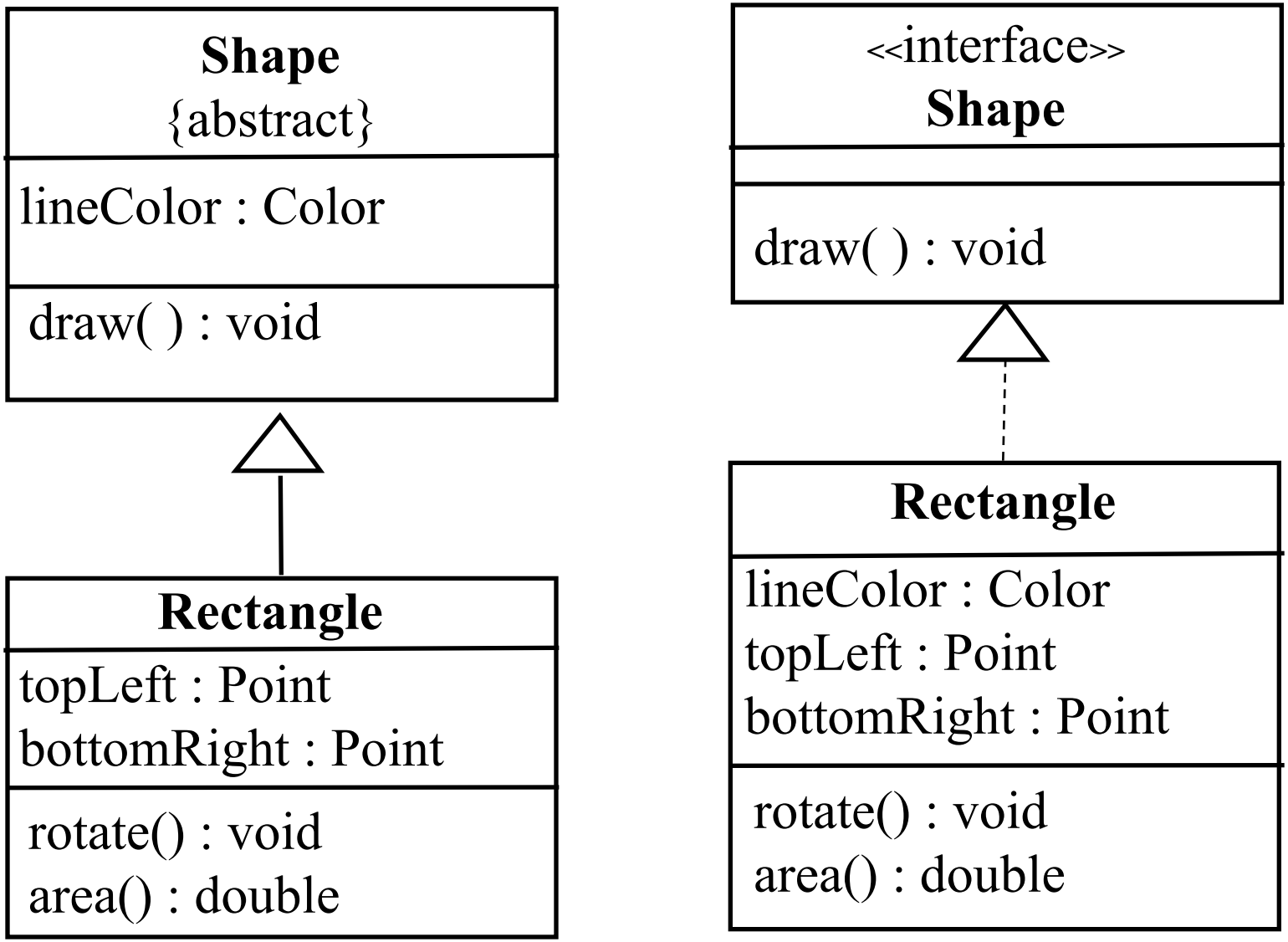
- A type is a predicate that allows identification of the set of all instances of that type
- It's possible that the set of all the instances of one type is a subset of the set of all instances of another. In this case we say that the second type is a *generalization* of the first
- Here's a specific example: A banking system has two kinds of customers – `CorporateCustomer` and `IndividualCustomer`. These have many features in common, but have some differences (for instance, the bank policies that apply are different)
- We define a generalization of the two called class `Customer`



UML Notation for Generalization



Examples of Generalization/Specialization



Inheritance

- Inheritance is sometimes referred to as the ‘Is-A’ relationship (although Fowler doesn't like to – his objection seems to be that people use the expression *x isa y* carelessly – it should be used as a short-hand in "Class X is a class Y", which expanded means "every instance of Class X is an instance of Class Y")
- You should use inheritance only when an instance of the derived class can be substituted everywhere that the base class appears
- In C++, public derivation is the means of implementing inheritance relationships. By contrast, the Java programming language allows one to declare an interface explicitly, and a class can declare that it implements multiple interfaces, while it can extend only one other class
- Inheritance, as described here, is sometimes qualified as ‘interface inheritance’, which means the incremental definition of an interface by including other interfaces

A Side Note on Polymorphism

- Polymorphism is an add-on to inheritance, and is a capability of an object system by which a method on an object is bound to the implementation of the run-time type of the object rather than the implementation of the declared type
- For instance, suppose we have the following Java code fragments

```
// File: Rectangle.java
public class Rectangle extends Shape {…}

// File: ShapeDriver.java
. . .
Shape s = new Rectangle ();
s.draw();
```

- The 'right thing' should happen here. What is it?

Compile-time vs Run-time Binding

- Binding is about an object system finding the right code to run when an object invokes a method on another object
- This binding can happen at compile time or at run time
- The advantage of compile-time binding is type safety
- The advantage of run-time binding is flexibility

```
// this requires run-time binding
Shape s = new Rectangle ();
s.draw();
```

```
// this allows compile-time binding
Rectangle r = new Rectangle ();
r.draw();
```

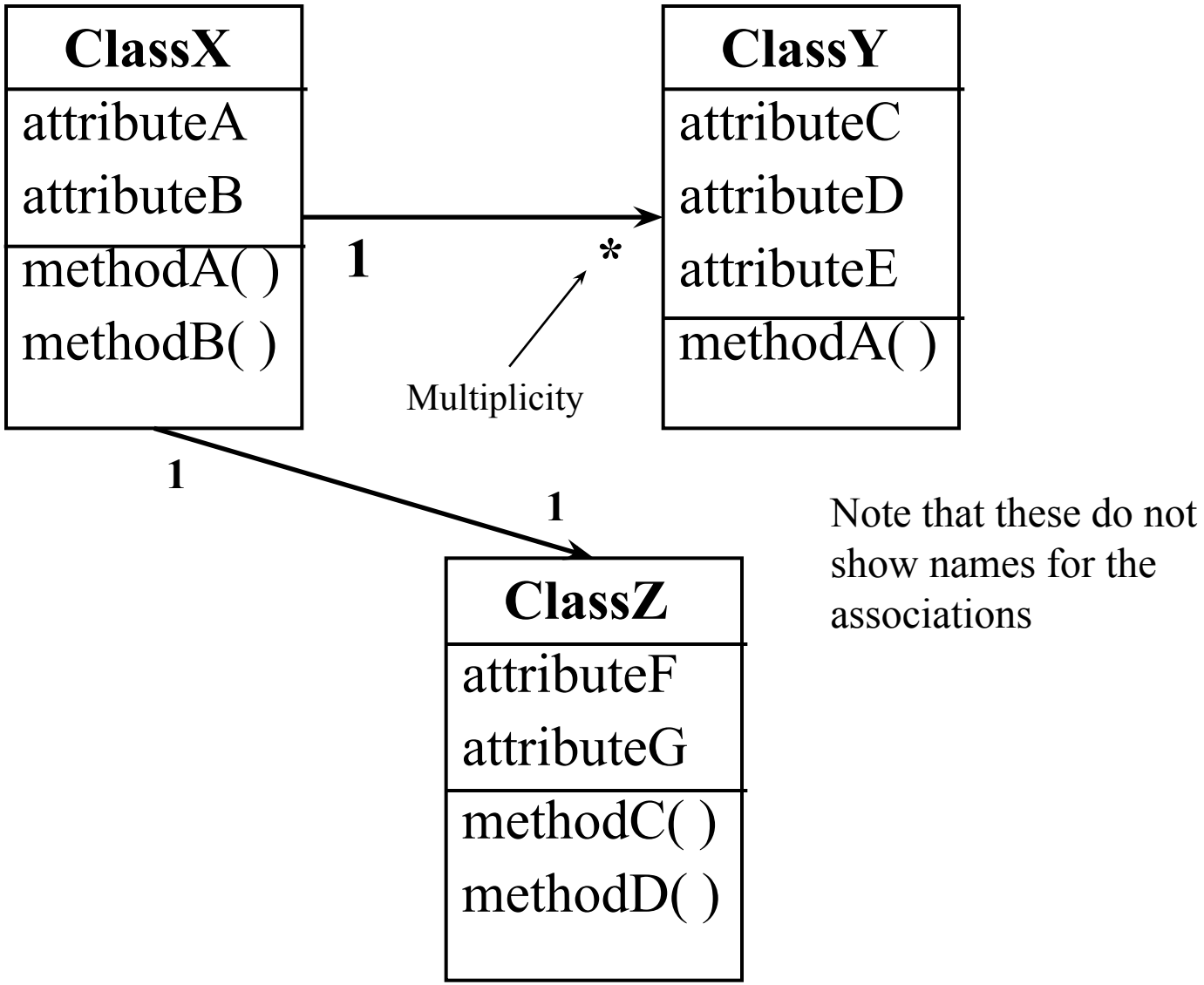
Associations

- Associations are the twin of attributes in that one could use one form to define the same features as the other
- In general use, however, associations are used to indicate a structural relationship of one class to another
- In a programming language, an association usually translates into a data member of one class holding a reference or collection of references to instances of another class
- Associations have all the fields that attributes do

Multiplicity for Associations

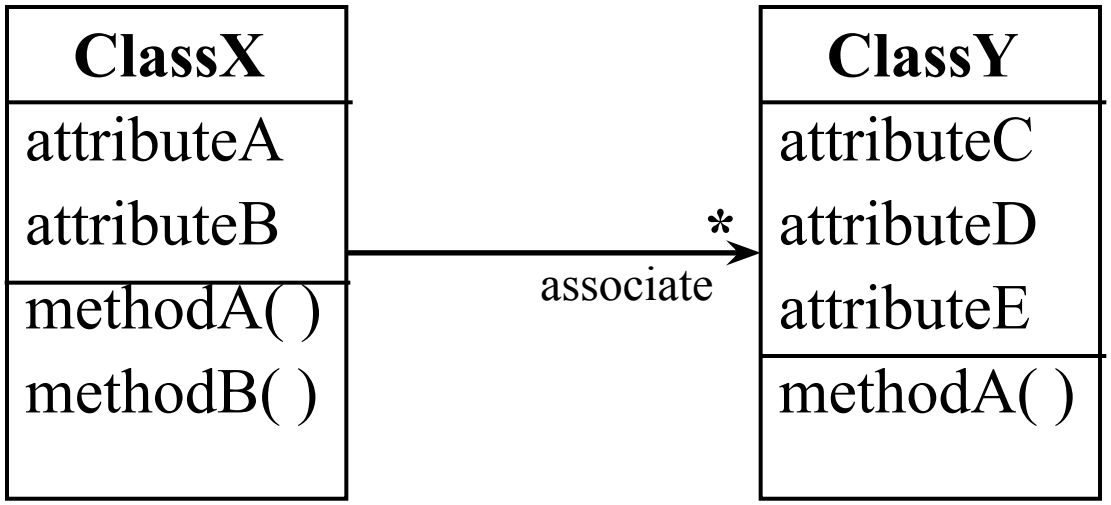
- For associations, the multiplicity is shown on a connector line near the target end
- The multiplicity of an association is usually a specific value or a range
- The default multiplicity is 1
- Typical multiplicity values are:
  - 1
  - 0..1
  - *
- One could have a case where there are one or two possible association values, so the range would be 1..2
- Associations indicate a structural dependency of one class on another. Concretely, an association usually translates into a data member of one class holding a reference or collection of references to instances of another class.

Association Example



Note that these do not show names for the associations

Conventions for Associations

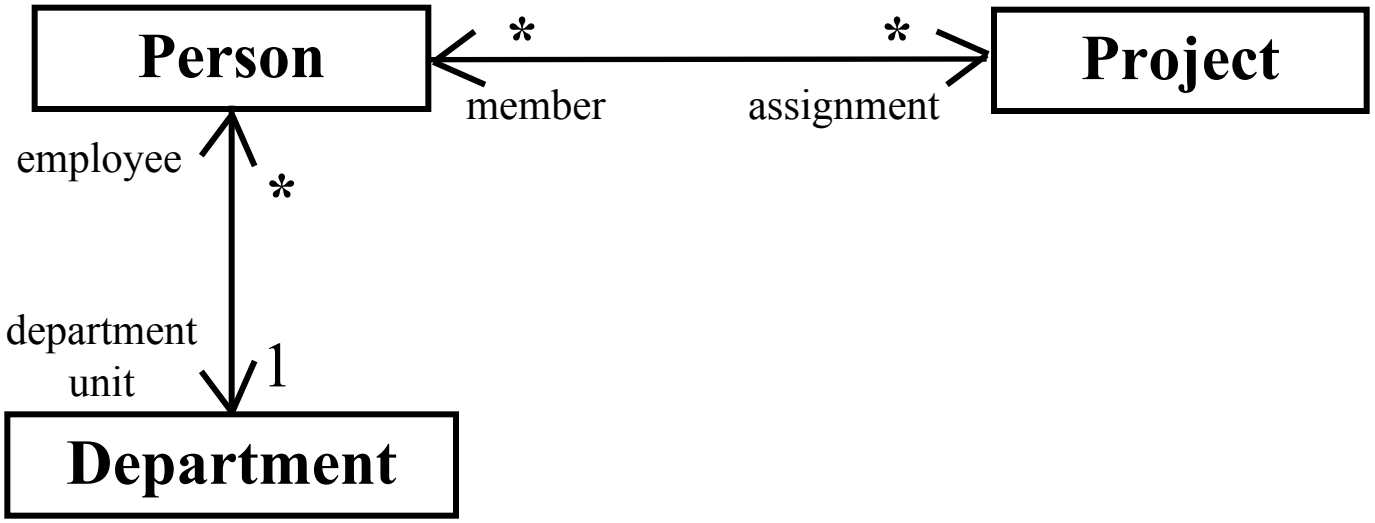


- Associations can have names, just as attributes do. Here we see a generic name ‘associate’ at the target end of the association
- For attributes, we expect two public accessors (get/set)
- For associations, there is a need for mechanisms to modify the association and to retrieve values from the association. Typically there are three public methods on `ClassX` we would expect to be defined implicitly

```
void add(ClassY cy)
void delete (ClassY cy)
Iterator list ()
```

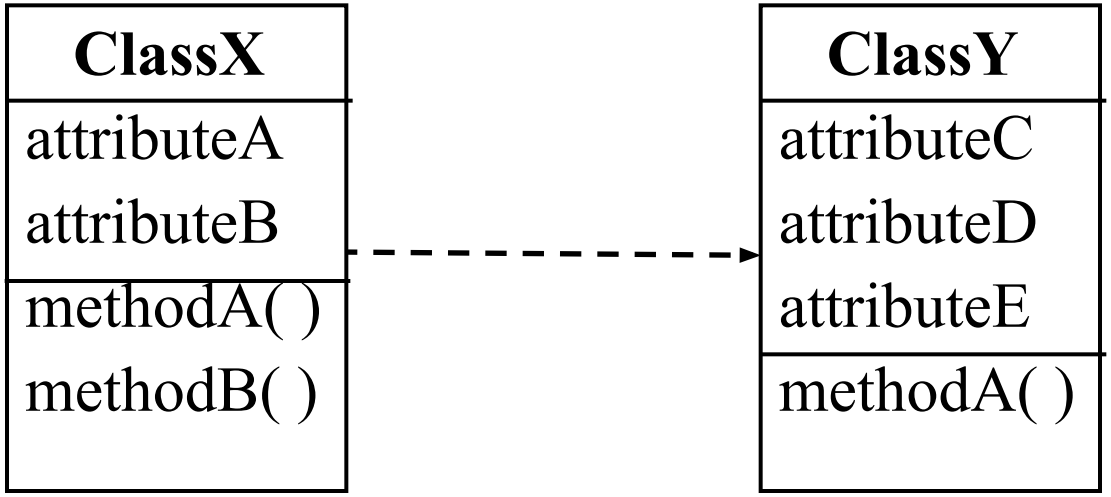
Example Associations

- The diagram below models the associations described in the statement "a person works in one department and a person can be assigned to multiple projects, while projects have many persons assigned"



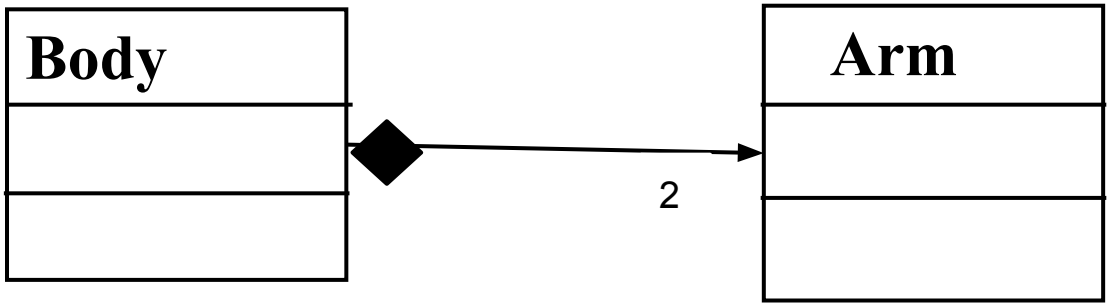
Class Dependencies

- One class depends on another if a change to the second can cause a change to the first. The simplest case of this is if the first class invokes an operation defined by the second
- One can denote a dependency with a dashed line, but it is not necessary to do so – in some cases, such as inheritance, there is a dependency that doesn't need the extra arrow. In the diagram, `ClassX` depends on `ClassY`



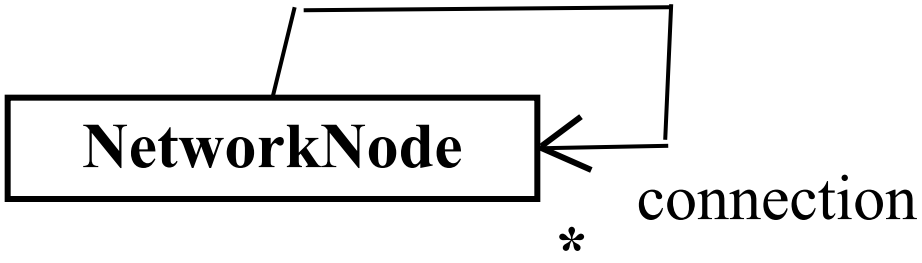
Composition

- Composition indicates that the one class is composed of another class.
- Shaded diamond is used to indicate composition
- Indicates strong ownership of associated class.
- Life cycle of associated class is usually restricted by owning class life cycle.
- In this partial class diagram for a robot, Arm is a component of Body, with a cardinality of 2.



Example: Recursive Associations

- Suppose we have a class `NetworkNode` that models a server in a network, and that each `NetworkNode` can be have direct connections to many others
- In this case, there is a recursive association of the class to itself



Attributes vs. Associations

- Suppose that in the early stages of modeling an ATM system we have already identified the need for an `Account` class and a `Customer` class. Then we encounter a requirement that states the following: "For each Account, the system will record all transactions (debits, credits, and transfers) with the date and time, the amount, and the customer"
- We decide to create a `Transaction` class, so now there are three classes

- Would we use an attribute or an association to hook up the `Transaction` to the `Account` and the `Transaction` to the `Customer`?
- Will the date and time be an attribute of `Transaction` or an association with `Transaction`?

How Are These Concepts Evident in Assignment 1?