CS 417
Design Patterns

UML Modeling part 2

Dr. Chad Williams
Central Connecticut State University

Topics

- Principles and concepts
- Modeling relationships and structures
- Modeling requirements with use cases

Terminology

- Objects are **equal** if states are the same
- Objects are **identical** if refer to same object
- **Accessor** is a method that allows you to read the state of an object (does not change the state)
- **Mutator** is a method that changes the state of an object
- **Immutable object** is an object whose state may never be modified by any of its methods – i.e. state is constant

Visibility

- **Public** – Feature is accessible to any class
- **Protected** – Feature available in the class itself, all classes in its same package, and all its subclasses
- **Package** – Feature is accessible to the class and all classes in the same package
- **Private** – Feature is only accessible within the class itself

Principles

- Good OO design always focuses on the principles of:
  - Modularity
  - Abstraction
  - Encapsulation

Modularity

- Complex code should be highly cohesive but loosely coupled
  - –Cohesive – all entities within a module should be closely related in what is their functionality
    - Each module small and simple
  - –Coupling – interdependency of different models
    - Interaction between modules should be simple
- Result is testing an individual module is simple and testing interaction between modules is simple, thus making testing of the entire application much easier

Module example

- 2 modules
  - Banking account
    - Functionality – get balance, make withdrawal...
  - Bill pay
    - Functionality – get bill balance, make payment
- Testing each module would be simple, and interface between the two modules would be simple as well
- General principle is if something is complex decompose it to its lower level simple parts
- A module is also a hierarchical concept where one module may be made up of other sub modules

Abstraction

- Principle: keep things simple
  - Reduce number of visible interfaces/methods and/or exposed attributes to only the essentials
  - Exposed methods act as *contractual interface* – specify what provided not how
    - Well defined set of expectations of passed parameters
    - Well defined set of output
    - Calling component is unaware/unaffected of any complexities that happen internally

Abstraction example

- Interacting with phone
  - Same basic interface whether analog, digital, encrypted, or cell phone
  - Simple interface to module, easy to understand and interact with
  - End users do not need to care about complexities behind the scenes

Encapsulation

- Implementation separated from contractual interface
- Purpose is to reduce interdependency of coupling modules
- Information hiding – the way a task is implemented is hidden from calling components
  - Hide method implementation
  - Hide variables used to support functionality

Encapsulation cont.

- With phone example not only does end user not need to know complexity, they don't need to know implementation being used
- One way of accomplishing this is to separate the interface used from the class implementing the interface

Polymorphism

- Ability to use several different service providers for same contractual interface
- Behavior of service can change dynamically without requiring any changes by components using the common interface
- Ex. a dual band cell phone switching from GSM to CDMA completely without ever impacting the user

Modeling relationships and structures

- Last time looked at creating classes in UML
- Here we look at depicting static structures and relationships between classes
- The combination of these help for the *class diagrams*, which describe the dependencies between classes (and thus objects once created)

Class diagrams

- The class diagram uses UML to describe:
  - Classes and interfaces
  - Relationships between classes and interfaces
    - Inheritance (extension and implementation)
    - Association (aggregation and composition)
    - Dependencies

Inheritance

- Defines a relationship among classes and interfaces
  - Extension between two classes
  - Extension between two interfaces
  - Implementation of an interface with a class

Class extension

- Conceptually extension means one class (the child class) is a subtype of another class (the parent class)
  - Ex. a dog is a subtype of mammal
- With UML this relationship is referred to as:
  - The child class *specializing* the parent
  - The parent *generalizing* the child

Class extension cont.

- Fundamentally the child class has all of the same features (attribute and methods) as the parent but differs in one or more of the following:
  - What the attributes mean
  - The behavior of the methods of the parent
  - Additional attributes
  - Additional methods
- However because it has all of the same features as the parent the child class can be used anywhere the parent without impact to the code

Extension cont.

- When classes extend classes, unless *overridden* any functionality or *implementation* of the parent is also reflected in the child class
  - Reuse of code without duplication
  - Fix once in parent class also fixes all child classes
  - By making methods of the parent private it allows complexity to be hidden from child classes

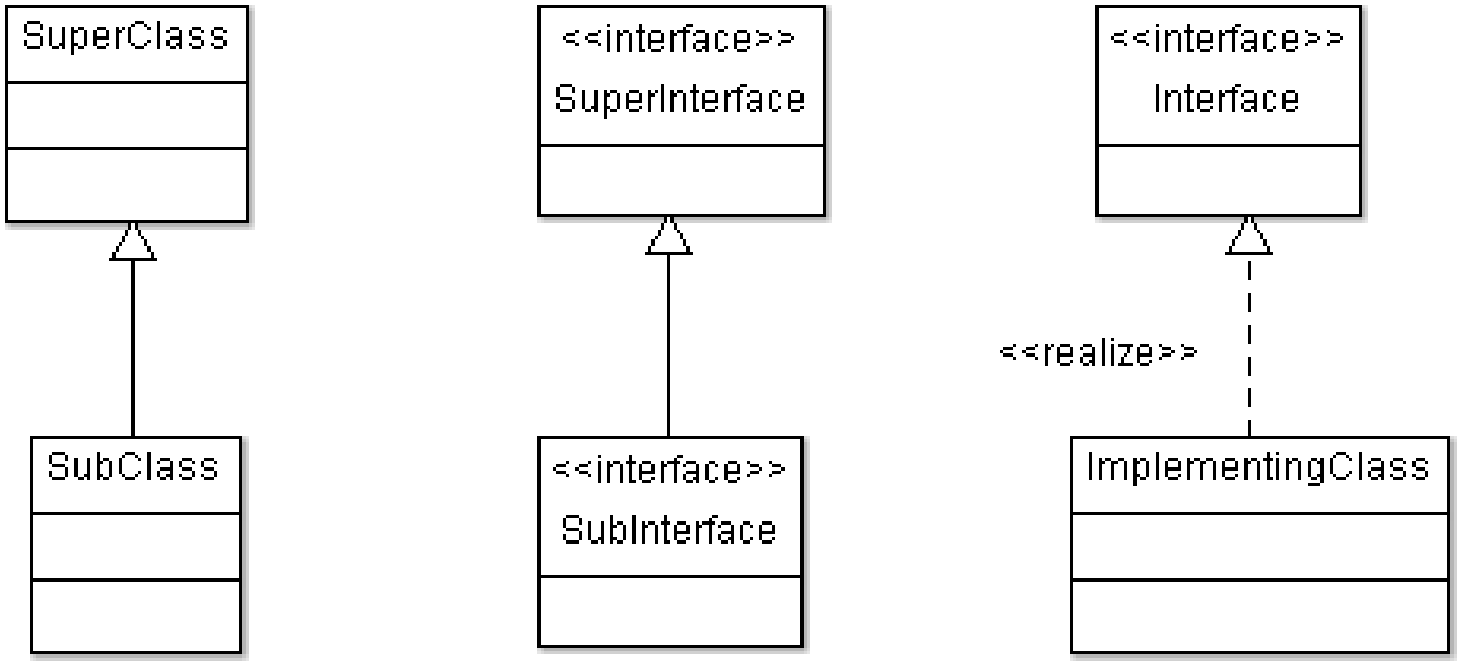
Interface extension

- Interfaces do not contain any implementation
- Like classes, interfaces can also extend one another
- Same terminology used (child, parent, specializes, generalizes)
- With interfaces the child interface contains the same *method signatures* as parent and new method signatures are added
  - ex. `int makeDeposit(int amount)`

Implementation

- When classes extend interfaces this is referred to in UML as *realization*
  - Indicates that the class provides an implementation of that interface
  - It is common for an interface to expose even fewer methods than the implementation to further help with abstraction
- Note: You can **not** have a interface extend a class

Inheritance in UML



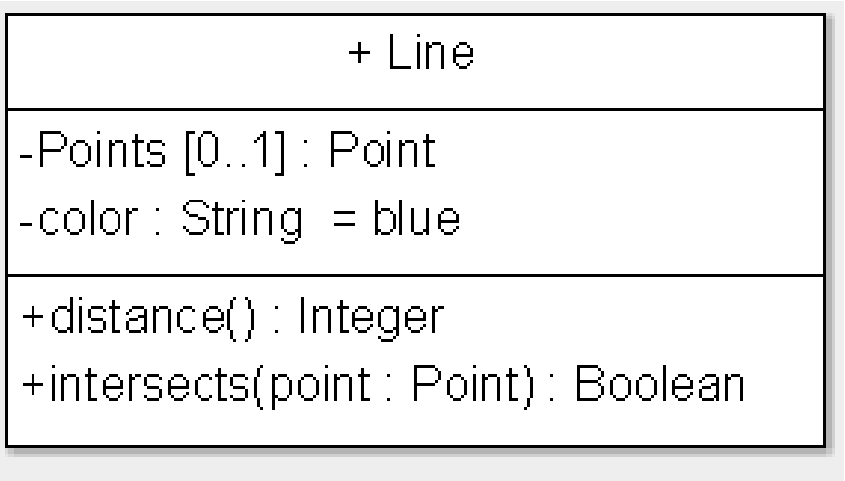
Group work (from previous)

- Split into groups 3 to 4 people
- Make up 3 classes for 2D geometric shapes. Draw UML for them, particularly think of attributes and methods that would require different visibilities – think moving, reshaping/manipulation, calculations, checks

Visibility	Java syntax	UML syntax
public	public	+
protected	protected	#
package		~
private	private	-

Group work

- Create a “Canvas” class, the class should have method(s) for you to place and then draw any of your shapes at a specific point(x,y) on the canvas. In order for a shape to be placed at a location it must not overlap with any shape already on the canvas. You should also be able to manipulate existing shapes (size, angle of triangle for example) with same rules. Refactor your initial model as necessary to make this as easy to maintain and add to as possible.



Group work

- Now revise your model to allow 3D shapes and placement of shapes in 3 dimensions on your canvas

What did you learn?

- Was your model easy to adapt to the change?
- Based on your final revisions looking back would there have been a better way to model your 2D? (with no knowledge that it would move to 3D)