CS-AD 209 - Spring 2013
Software Engineering
Lecture 2

Professor Michael Paik
mpaik@cs.nyu.edu

Attendance

Agenda

- Administrivia
- Object Oriented Design
- Composition
- Modular Dependency Diagrams

Agenda

- Administrivia
- Object Oriented Design
- Composition
- Modular Dependency Diagrams

Syllabus Update

New syllabus, including topic coverage, is posted on newclasses

Exams

Moving from
1 Midterm + Final to
2 Midterms + Final

Rescheduling

Need to reschedule
25 February and 8 April
classes – let me know your
availability

Lecture Notes

Drawing from my experience teaching Intro, providing complete lecture notes makes people not come to office hours and creates a false sense of 'It's fine, I'll look at it later'

As such, I will not necessarily be providing answers in the slide decks for the various examples

Agenda

- Administrivia
- Object Oriented Design
- Composition
- Modular Dependency Diagrams

Object Oriented Design

Object orientation has become a standard set of tools to design and implement software artifacts

(Good!)

Object Oriented Design

The toolbox of classes, subclasses, interfaces and abstract classes has provided a useful standard set of semantics

(Good!)

Object Oriented Design

Along the way, however, particularly with beginner programmers, an idea has arisen of the One Design To Rule Them All™ that is notionally perfect for your program

(Bad!)

Refresher

A *class*, or *type*, as you will recall, is a construct that has zero or more *members* and zero or more *methods*

Members are collections of variables contained within a class or *instance* of a class (a la C `structs`)

Methods are functions that (usually) operate on the members of a class or instance and are defined within the class

Refresher

An *instance* is a concrete copy in the mold of a class definition (e.g. CS-AD 101 and CS-AD 209 and their attendant information might be a instances of the class `NYUADClass`) that has appropriate members and methods

Refresher

An *interface* is an implementation-free specification of methods (also members, but there's a nuance) that an *implementing* class must provide code for

Interfaces are typically used as a workaround in languages that don't support multiple inheritance, so those that do support it may be missing this language feature (e.g. Python)

Refresher

An *abstract* class is like a normal class, but methods can optionally omit implementation

It is like an interface in purpose insofar as it's meant to specify behavior to be implemented by another class, but unlike in that it can contain code and be inherited from like any other class

Abstract classes cannot be instantiated

Refresher

A *subclass* is a class that extends or inherits members and/or methods from a (potentially abstract) parent class

What Does What?

A class, conceptually, is usually a ***noun*** – it represents a class of persons, places, or (usually) things

An instance is a particular member of the class – e.g. Abu Dhabi is a particular member of the class of things known as Cities

What Does What?

An interface, conceptually, is usually an ***adjective*** – it represents a set of properties and attendant capabilities, and can be applied to various appropriate classes

A common example in Java, for instance is `Comparable`, which specifies a method `.compareTo()`; any class may implement this interface to provide comparison mechanics

What Does What?

An *abstract class*, conceptually, is usually just an overly broad class/set of nouns represented by classes

Abstract classes will have properties and attendant implementation common to the inheriting classes, but not enough information to provide a concrete instance, e.g. a class `AbstractDataStore` might provide code for saving and loading from somewhere, but this is abstract without an implementation that makes it realizable

Interfaces vs. Abstract Classes

Remember, interfaces are *adjectives*, and classes (abstract or otherwise) are *nouns* in our grammar

If you have a behavior or property that is likely to be commonly used for *heterogeneous* classes (different implementations), use an interface

If you have a behavior or property that is specific to a particular conceptual class of nouns (same implementation), use an abstract class

Interfaces vs. Abstract Classes

Language enforcement of these constructs will vary (e.g. Python's Duck Typing provides no explicit support for interfaces and ugly support for actual abstract classes) but you can still build along these lines

Exercise

Create a set of objects characterizing a playing card game

Which elements are classes, which interfaces, and which abstract?

How would the program be different with a non-object oriented approach (think bash shell script)

Rule of Thumb

Object oriented design is meant to reduce the amount of code through reuse and simplify comprehension

If you are adding more elements to the detriment of this purpose, you are doing it wrong

In particular, design to be *just* general enough to support conceivable future feature additions, and no more

Multiple Inheritance

Only certain languages (e.g. Python) support multiple inheritance, with strict method MRO (method resolution order) rules to prevent method name collision from multiple superclasses

In general try to use these as you would interfaces and keep them logically distinct; if the MRO has to be invoked because multiple superclasses have the same method name, you are probably doing something wrong

Agenda

- Administrivia
- Object Oriented Design
- Composition
- Modular Dependency Diagrams

Composition

Another orthogonal way of organizing classes is *composition*

In composition, we move from the 'is a' grammar of subclasses to 'has a'

Composition

This is different from the normal use of members because it is an extraction of members and methods that would otherwise be inherited

Example

How would you characterize things that can fly?

Composition

Composition is useful because it can be easier to understand in some situations, and performs slightly better under some MRO regimes (traversing class hierarchies takes time)

However, the performance gain is rarely significant, so if you don't find this cognitively useful in simplifying your object model, don't use it

Agenda

- Administrivia
- Object Oriented Design
- Composition
- Modular Dependency Diagrams

The 100-Foot View

Now that we have our full gamut of object oriented primitives, we can start thinking one level up from implementation to design

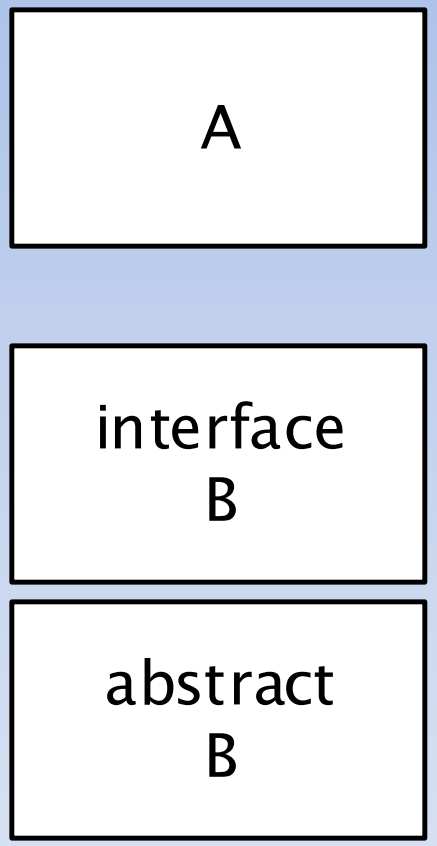
To make this easier, we should speak a common language

Module Dependency Diagrams

A useful standard tool for visualizing the relationships between these components is the *Module Dependency Diagram* (*MDD*)

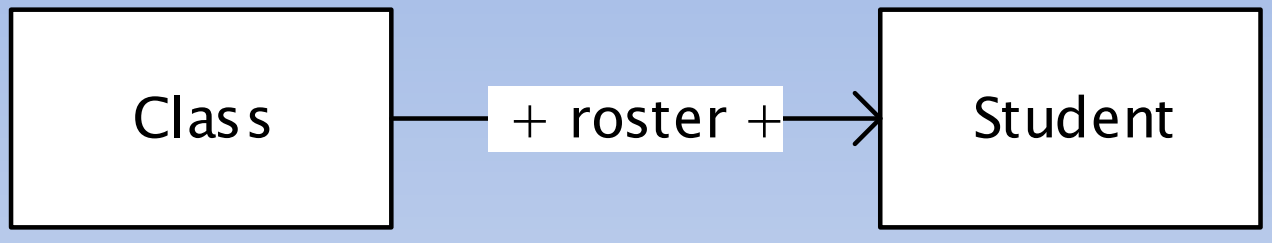
This diagram indicates the various relationships between classes/interfaces and their quantities (and is very useful when you're building data models for persistence later)

MDD Notation



- A box represents a class, its name in normal font
- An interface or abstract class looks the same except it specifies 'interface' or 'abstract'

MDD Notation

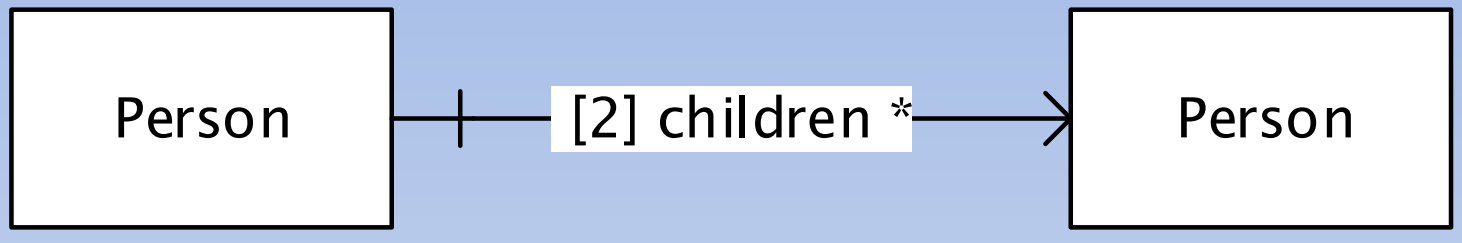


- An arrow with a v-shaped head indicates membership, and such arrows are labeled with the field name and multiplicities

MDD Notation

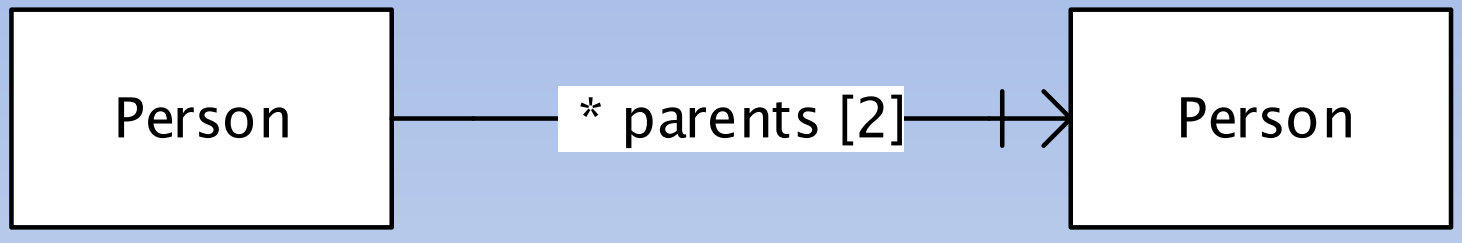
- *, the Kleene star, represents 0 or more
- + represents 1 or more
- ? represents 0 or 1
- ! represents exactly 1
- [*m*...*n*] represents a value between *m* and *n*, inclusive

MDD Notation



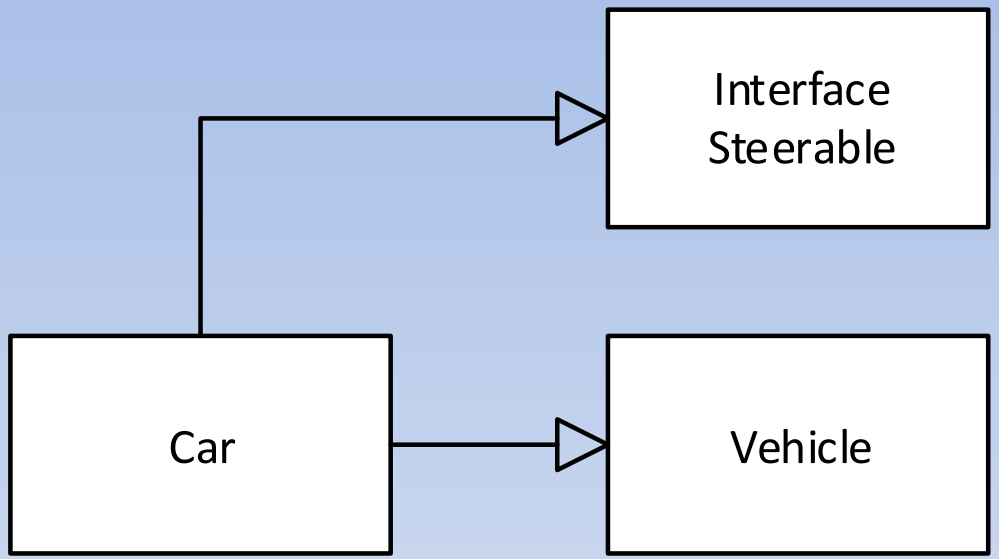
- A vertical line at the source means that the relationship is source-static, that is, the set of instances of the source mapped to a given target doesn't change over a target's lifetime

MDD Notation



- A vertical line at the target represents the converse: that the relationship is target-static, that is, the set of instances of the target mapped to a given source doesn't change over a target's lifetime

MDD Notation



- An arrow with a triangle-shaped head indicates inheritance or implementation (in this case `Car` implements `Steerable` and inherits from `Vehicle`)

Exercise

Let's revisit our playing card example

Draw a modular dependency diagram for our earlier solution, including quantities

Alternatives in Practice

MDDs as specified here are lightweight and easy to use, but have expressive limitations

More modern alternatives like UML etc. exist that have greater expressive power, but at the cost of more complex grammars

Homework

Using the Visio stencils provided (courtesy of MIT's 6.170 class) on newclasses, create a module dependency diagram of a program of sufficient complexity to require subclassing, interfaces, and abstract classes, and specify quantities using the given notation (you'll find a key on newclasses)

In addition to your diagram, provide one alternate design

If you don't have Visio, you can use alternative open-source software like LibreOffice Draw, Dia, etc. I believe ITS can provide a Visio installation for you if you request it

Reading for Next Class

Liskov Paper (on newclasses)

(~22pp)