Lecture Notes
October 10, 2017

Requirements and Wireframes team assignment due Thursday

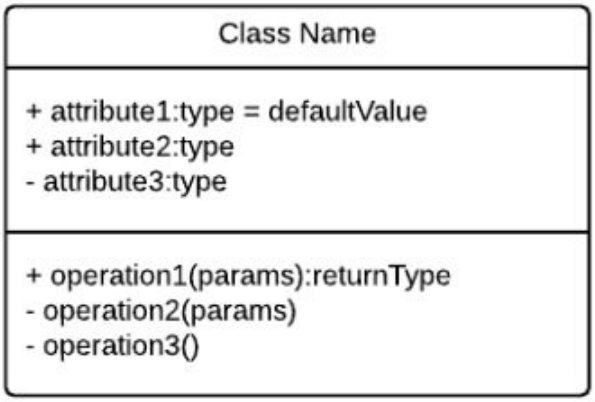
Design team assignment due next Thursday October 19

Full set of first iteration team assignments posted

Class Diagrams

Goal of class diagrams is to convey information about *static structure* of your application domain - there are other kinds of UML diagrams for dynamic behavior (e.g., activity and sequencing)

Useful to communicate, visualize and/or analyze



A class can be a type (that can be instantiated many times) or a singleton (no more than one), applies to non-OO languages, however data types and singleton data entities are defined

Attributes: What describes it? Properties of the object - may be primitive or references to collaborators

Operations: What does it do? Behaviors correspond to the responsibilities

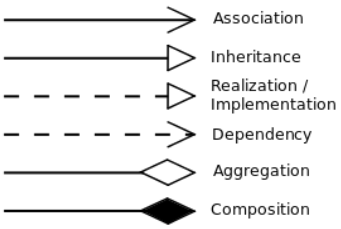
Usually omit CRUD: constructors, getters, setters, finalizers

Signatures (type, defaultValue, params, returnType) often omitted

Some class diagrams indicate visibility (+ public, - private, etc.)

Some class diagrams show the structure of a single class, with its attributes and operations, while others focus on the *relationships* among classes - only class name within class box, main text is on the association lines/arrows, some mix/match

On CRC cards, *any* kind of relationship is fine for “collaborators”, but in class diagrams, we refine the kind of relationship



Association is most general, instances of one class know about or communicate with instances of another class (or the same class)

Usually not transient, an attribute in instances of one class references instance(s) of the other class (the name of the attribute will also appear as the name of the association line), not just that some method of a class receives parameters that are instances of the other class

unidirectional represented as line with arrow

bidirectional represented as just plain line

Open and closed aggregation are special cases of association

Open aggregation (or weak aggregation) corresponds to HAS-A, no lifecycle relationship

Line with open diamond at container class

Multiplicity at contained class end of line indicates number of instances

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	0 or more
1..* | *	1 or more
n	Only n (where n > 1)
0..n	Zero to n (where n >1)
1..n	One to n (where n > 1)

Closed aggregation (composition) corresponds to CONSISTS-OF rather than HAS-A, strong lifecycle relationship

> *Strong lifecycle relationship* means deleting parent automatically deletes all the children, or alternatively that all the children must have already been deleted to delete the parent, children can belong to only one parent
>
> Line with closed/filled diamond at container class
>
> Multiplicity at contained class end of line: N..M, 0..*, 1..256, 3

Besides instance level relationships, also class level relationships:

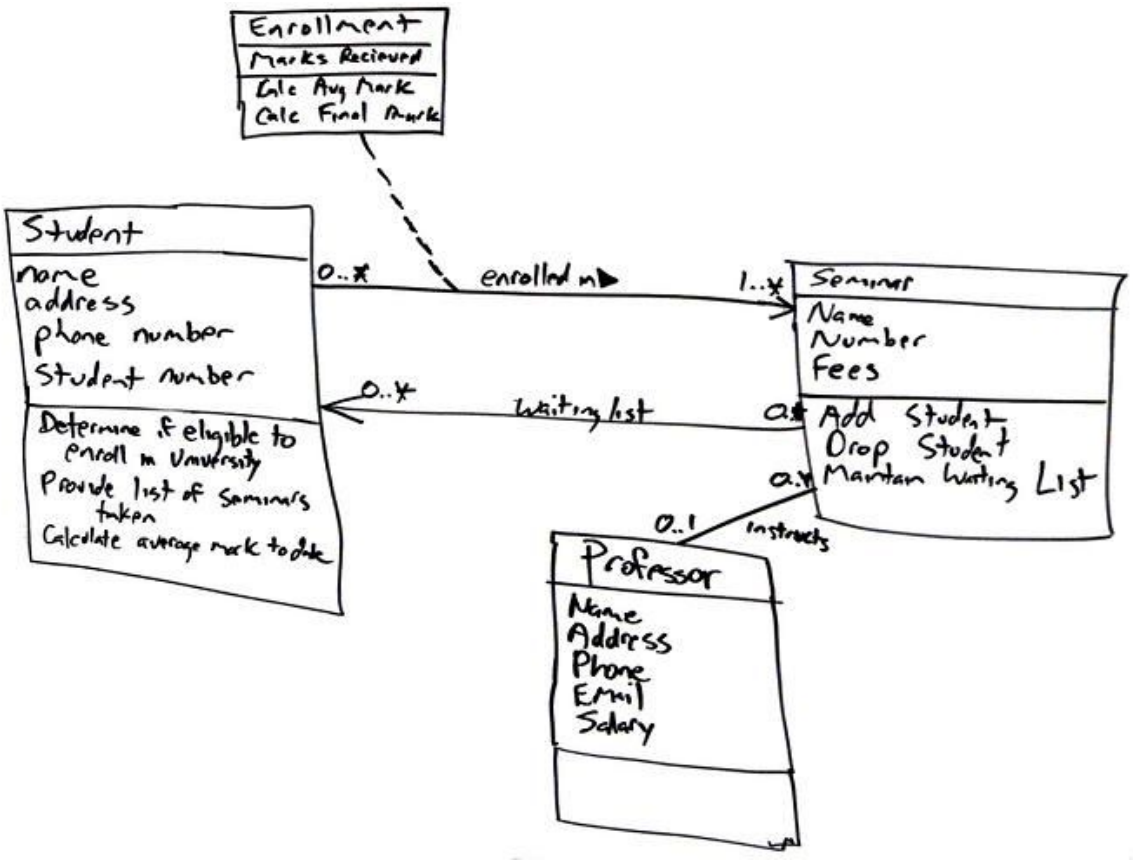
Generalization or Inheritance from superclass (abstraction, base class) to subclass (specialization, extension, derived class)

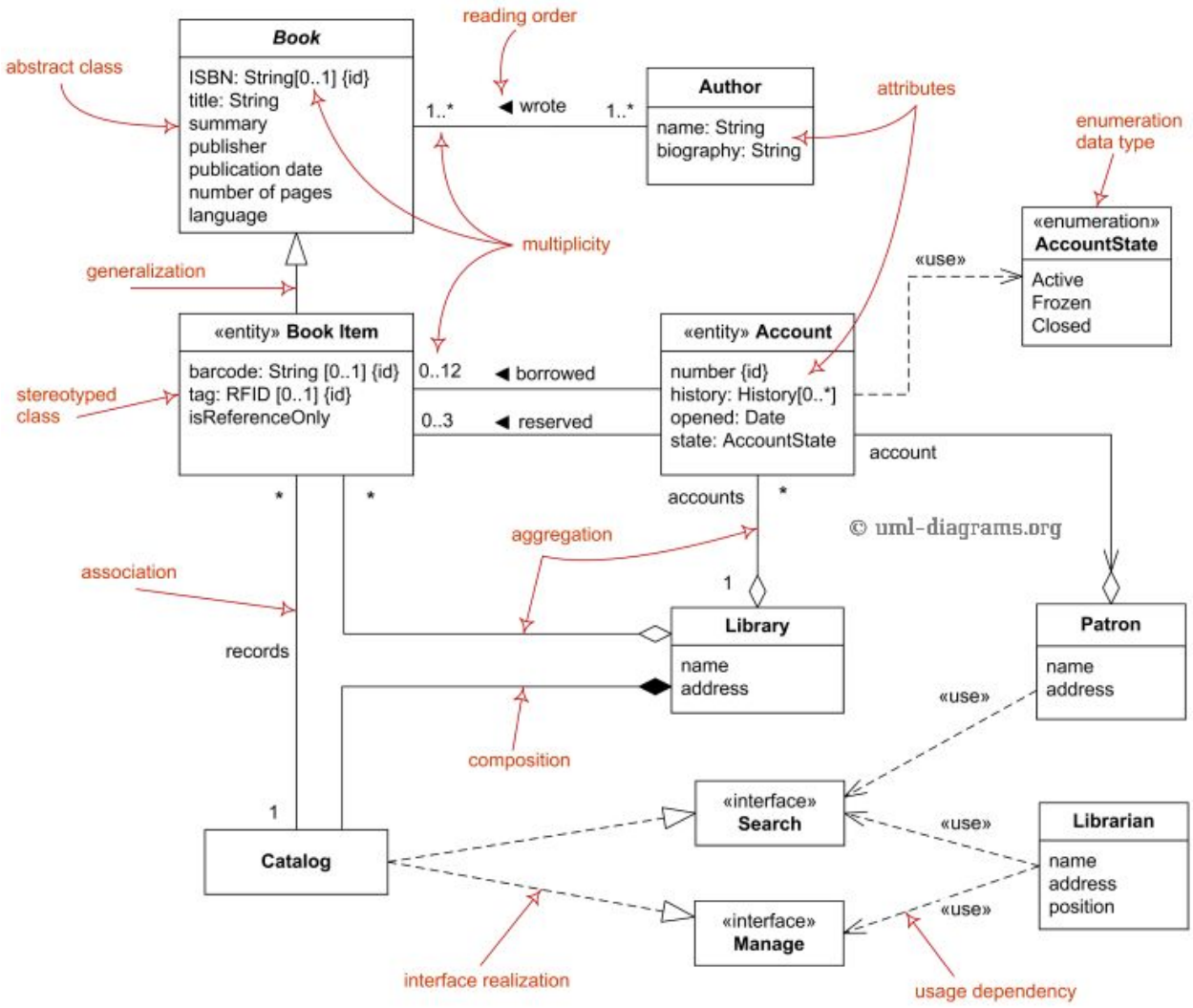
> Applies even to non-OO languages, you can have generalized types and specialized types associated with each other conceptually even though there is no inheritance, polymorphism, etc.
>
> Solid line with open (unfilled) arrowhead from subclass to superclass

Realization or Implementation of an interface

> Applies even to languages without concept of “interface”, which might exist only conceptually not in the code
>
> Dashed line with open arrowhead (triangle) from implementing class to interface
>
> The interface is represented in UML like a class, adding special marker <<interface>>

Tools for creating class diagrams: any drawing program will work, consider the “free EDU” accounts at Lucid Chart https://www.lucidchart.com/pages/usecase/education or open-source UMLet http://www.umlet.com/ (web version at http://www.umlet.com/umletino/)


Enrollment
Marks Recieved
Calc Avg Mark
Calc Final Mark
Student
name
address
phone number
Student number
Determine if eligible to enroll in University
Provide list of Seminars taken
Calculate average mark to date
0..*
enrolled in
1..*
Seminar
Name
Number
Fees
Add Student
Drop Student
Maintain Waiting List
0..*
Waiting list
0..1
instructs
Professor
Name
Address
Phone
Email
Salary


Book
ISBN: String[0..1] {id}
title: String
summary
publisher
publication date
number of pages
language
abstract class
reading order
wrote
1..*
1..*
Author
name: String
biography: String
attributes
enumeration
data type
«enumeration»
AccountState
Active
Frozen
Closed
«use»
multiplicity
generalization
«entity» Book Item
barcode: String [0..1] {id}
tag: RFID [0..1] {id}
isReferenceOnly
stereotyped
class
0..12
borrowed
0..3
reserved
«entity» Account
number {id}
history: History[0..*]
opened: Date
state: AccountState
account
accounts
*
*
*
1
© uml-diagrams.org
aggregation
association
records
Library
name
address
Patron
name
address
«use»
composition
1
«interface»
Search
«use»
Librarian
name
address
position
Catalog
«interface»
Manage
«use»
interface realization
usage dependency

Design Principles

A design principle is a technique that can be applied to designing or writing code to make that code more maintainable, flexible, and extensible

You Aren't Gonna Need It (YAGNI): do the simplest thing that could possibly work, a tenet of most agile processes

Single Responsibility Principle (SRP): Every object should have a single responsibility, and all its tasks or services should be focused on that responsibility (same as cohesion)

Seemingly contradicts the multiple responsibilities of CRC cards, but there may refer to tasks or services that are all part of the same focused single responsibility

How to tell when you're violating SRP: Does it make sense to say "The <class name> <method name> itself" for each method? (after fixing grammar)

Only one reason to change: When the requirements of the project change, the modifications will be visible through the alteration of the responsibilities of the classes. If a class has several responsibilities, then it will have more reasons to change.

Break up the functionality into multiple classes, where each individual class does only one thing (may lead to more smaller classes)

Class may start out reasonable but new functionality is added during changes, so refactor the class then

Don’t Repeat Yourself principle (DRY): Avoid duplicate code and redundant work, instead abstract out common code and put in a single location

Do not copy/paste - if you find code that you would like to copy/paste (from your own codebase), remove from its original location and put in a new method, then both places should call that method

When similar code was written independently, detect and combine during refactoring
(some refactoring tools automate “code clone” detection)

Make sure that you only implement each feature and requirement once

Each piece of information and behavior in your system in a single, sensible place - applies to requirements and design as well as code

DRY is about putting a piece of functionality in one place,
SRP is about making sure a class does only one thing

Open-closed principle (OCP): Classes should be open for extension and closed for modification, allows change but without modifying any existing code

Once you have a class that works, and is being used, you really don't want to make changes to it unless you have to

But since requirements change, we need to be able to change behavior - one way to change the behavior of an existing class is to derive a new subclass

Combination of encapsulation and abstraction - find the behavior that stays the same, and abstract that behavior away into a base class

Another way to change existing behavior is to add new methods that invoke the original methods in different ways

Liskov Substitution Principle (LSP): Extends OCP to require that subtypes must be substitutable for their base types (polymorphism)

Do inherited methods make sense for instances of the subclass?

Do overridden methods take the same parameter types and return the same types?

If not, better to delegate or compose/aggregate for reuse rather than inherit

Code already using references to the base classes should be able to use instances of the derived classes without being aware of this

"Design by Contract" - The contract of a method informs the author of a class about the behaviors that he can safely rely on. The contract could be specified by declaring preconditions and postconditions for each method. The preconditions must be true in order for the method to execute. On completion, after executing the method, the method guarantees that the postconditions are true. Derived classes need to fulfill the same contract.

Where are these examples from? Head First Object Oriented Analysis and Design