UML Class Diagrams

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

Outcomes

After today’s lecture you will be able to:

- Understand the basic concepts of domain analysis
- Be capable of using domain modeling to model requirements
- Be capable of extracting UML class diagrams from requirements using textual analysis
- Understand the basic components of a class diagram

Inspiration

“Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away” – Antoine de Saint-Exupery

Design Phase: From Requirements to Code

Software Design

- **Design:** specifying the structure of how a software system will be written and function without actually writing the complete implementation
- A transition form “what” the system must do, to “how” the system will do it
  - What classes will we need to implement a system that meets out requirements?
  - What fields and methods will each class have?
  - How will the classes interact with each other?

How to Design Classes?

Identify classes and interactions from project requirements

- **Nouns** are potential classes, objects, and fields
- **Verbs** are potential methods or responsibilities of a class
- **Relationships** between nouns are potential interactions (containment, generalization, dependence, etc.)

How to Design Classes?

Identify classes and interactions from project requirements

- **Nouns** are potential classes, objects, and fields
- **Verbs** are potential methods or responsibilities of a class
- **Relationships** between nouns are potential interactions (containment, generalization, dependence, etc.)

- Which nouns in your project should be classes?
- Which ones are fields?
- What verbs should be methods?
- What are potential interactions between your classes?

Describing designs with UML

- Class Diagram (Today)
  - Shows classes and relationships among them.
  - A static view of the system, displaying what interacts but not what happens when they do interact.
- Sequence Diagram (next lecture)
  - A dynamic view of the system, describing how objects collaborate: what messages are sent and when.

Describing Designs with UML: an overview

What is UML?

- Pictures or views of an OO system
  - Programming languages are not abstract enough for OO design
  - UML is an open standard; lots of companies use it
- What is legal UML?
  - A descriptive language: rigid formal syntax (like programming)
  - A prescriptive language: shaped by usage and convention
  - It’s okay to omit things from UML diagrams if they aren’t needed by team/supervisor/instructor

UML: Unified Modeling Language

- Union of Many Languages
  - Use Case diagrams
  - Class diagrams
  - Object diagrams
  - Sequence diagrams
  - Collaboration diagrams
  - Statechart diagrams
  - Activity diagrams
  - ...

Uses for UML

- As a **sketch**: to communicate aspects of system
  - Forward design: doing UML before coding
  - Backward design: doing UML after coding as documentation
  - Often done on whiteboard or paper
  - Used to get rough selective ideas
- As a **blueprint**: a complete design to be implemented
  - Sometimes done with CASE tools
- As a **programming language**: with the right tools, code can be auto-generated and executed from UML
  - Only good if this is faster than coding in a “real” language

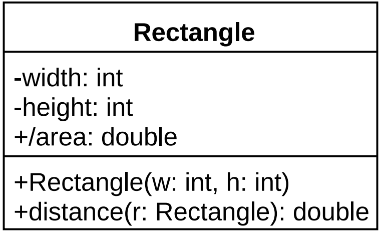
UML Class Diagrams

What is a UML Class Diagram?

- A UML class diagram is a picture of
  - the classes in an OO system
  - their fields and methods
  - connections between the classes that interact or inherit from each other
- Not represented in an UML class diagram:
  - details of how the classes interact with each other
  - algorithmic details: how a particular behavior is implemented

Diagram of a single class

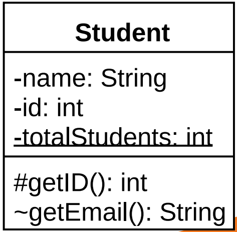
- Class name
  - write <<`interface`>> on top of interfaces' names
  - use italics for an abstract class name
- Attributes (optional)
  - fields of the class
- Operations / methods (optional)
  - may omit trivial (get/set) methods
  - but don't omit any methods from an interface!
  - should not include inherited methods



Rectangle

-width: int
-height: int
+/area: double

+Rectangle(w: int, h: int)
+distance(r: Rectangle): double



Student

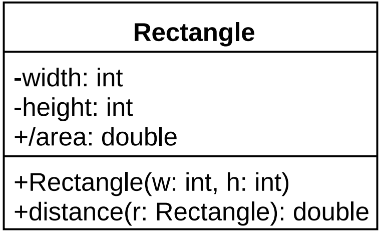
-name: String
-id: int
-totalStudents: int

#getID(): int
~getEmail(): String

Class Attributes

```
visibility name : type [count] = default_value
```

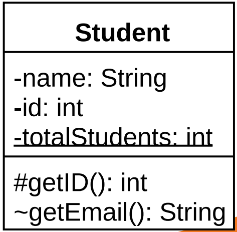
- **visibility**
  - + public
  - # protected
  - - private
  - ~ package (default)
  - / derived
- **derived attribute**: not stored, but can be computed from other values
- underline static attributes



Rectangle

-width: int
-height: int
+/area: double

+Rectangle(w: int, h: int)
+distance(r: Rectangle): double



Student

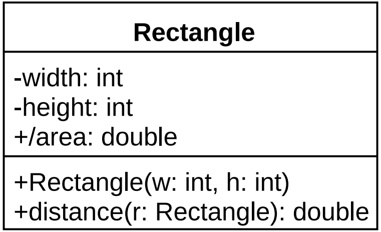
-name: String
-id: int
-totalStudents: int

#getID(): int
~getEmail(): String

Class Operations

`visibility name(parameters) : return_type`

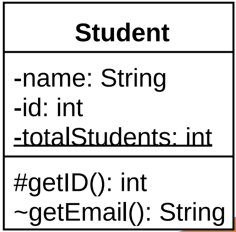
- **visibility**
  - + public
  - # protected
  - - private
  - ~ package (default)
- **parameters** listed as `name : type`
- underline static methods
- omit `return_type` on constructors and when return type is `void`



Rectangle

-width: int
-height: int
+/area: double

+Rectangle(w: int, h: int)
+distance(r: Rectangle): double



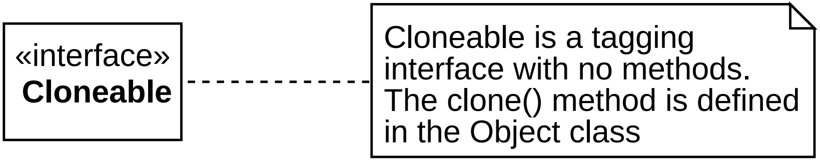
Student

-name: String
-id: int
-totalStudents: int

#getID(): int
~getEmail(): String

Comments

Represented as a folded note, attached to the appropriate class/method/etc. by a dashed line

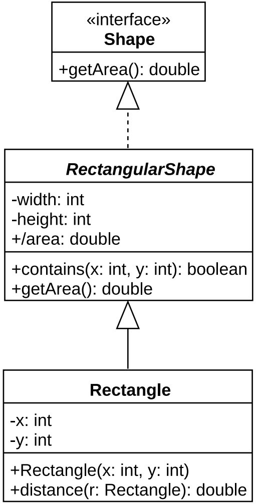

«interface»
Cloneable
Cloneable is a tagging interface with no methods. The clone() method is defined in the Object class

Class Relationships

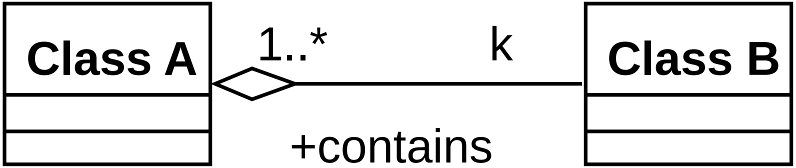
- **Generalization**: an inheritance relationship
  - inheritance between classes
  - interface implementation
- **Association**: a usage relationship
  - dependency
  - aggregation
  - composition

Generalizations

- Hierarchies drawn top-down
- Arrows point upward to parent
- Line/arrow styles indicate if parent is a(n):
  - **class**: solid line, black arrow
  - **abstract class**: solid line, white arrow
  - **interface**: dashed line, white arrow
- Often omit trivial / obvious generalization relationships, such as drawing the Object class as a parent

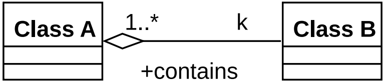


Associations



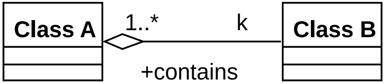
Associations

- Multiplicity (how many are used)
  - `*` (zero or more)
  - `1` (exactly one)
  - `2..4` (between 2 and 4, inclusive)
  - `3..*` (3 or more, `*` may be omitted)



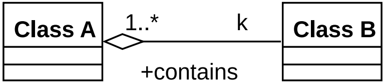
Associations

- Multiplicity (how many are used)
  - `*` (zero or more)
  - `1` (exactly one)
  - `2..4` (between 2 and 4, inclusive)
  - `3..*` (3 or more, `*` may be omitted)
- Name (what relationship the objects have)



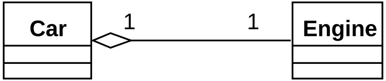
Associations

- Multiplicity (how many are used)
  - `*` (zero or more)
  - `1` (exactly one)
  - `2..4` (between 2 and 4, inclusive)
  - `3..*` (3 or more, `*` may be omitted)
- Name (what relationship the objects have)
- Navigability (direction)

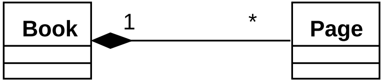


Multiplicities

- **One-to-one**
  - Each car has exactly one engine.
  - Each engine belongs to exactly one car.

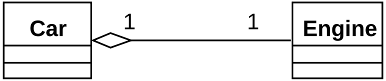


- **One-to-many**
  - Each book has many pages
  - Each page belongs to exactly one book



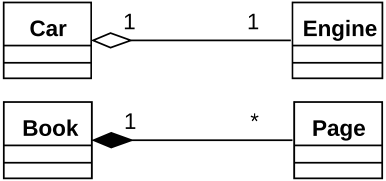
Association Types

- **Aggregation**: “is part of”
  - symbolized by a clear white diamond



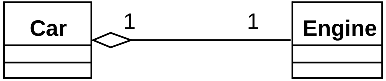
Association Types

- **Aggregation**: “is part of”
  - symbolized by a clear white diamond
- **Composition**: “is entirely made of”
  - stronger version of aggregation
  - the parts live and die with the whole
  - symbolized by a black diamond

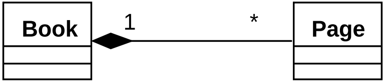


Association Types

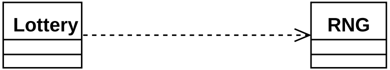
- **Aggregation**: “is part of”
  - symbolized by a clear white diamond



- **Composition**: “is entirely made of”
  - stronger version of aggregation
  - the parts live and die with the whole
  - symbolized by a black diamond

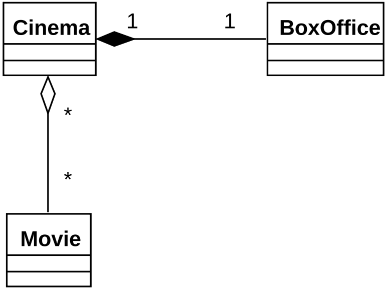


- **Dependency**: “uses temporarily”
  - symbolized by dotted line
  - often is an implementation detail, not an intrinsic part of the object’s state

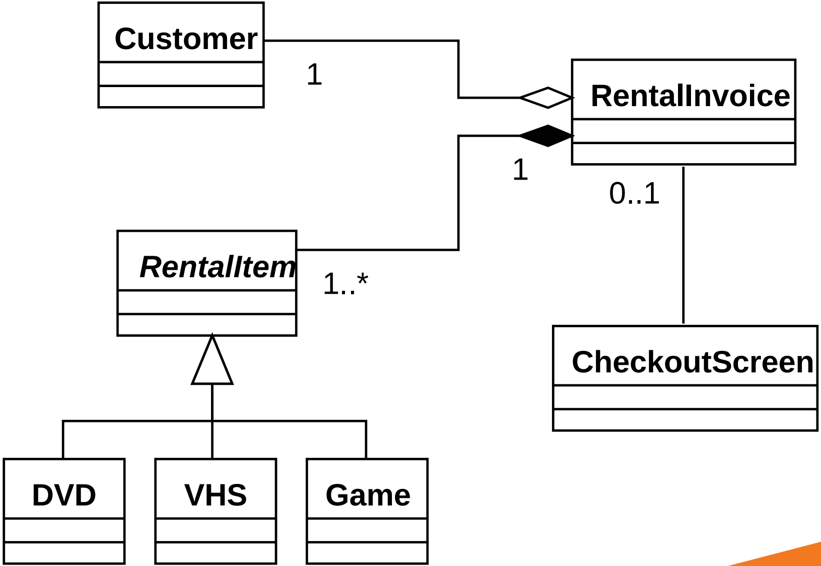


Aggregation/Composition Example

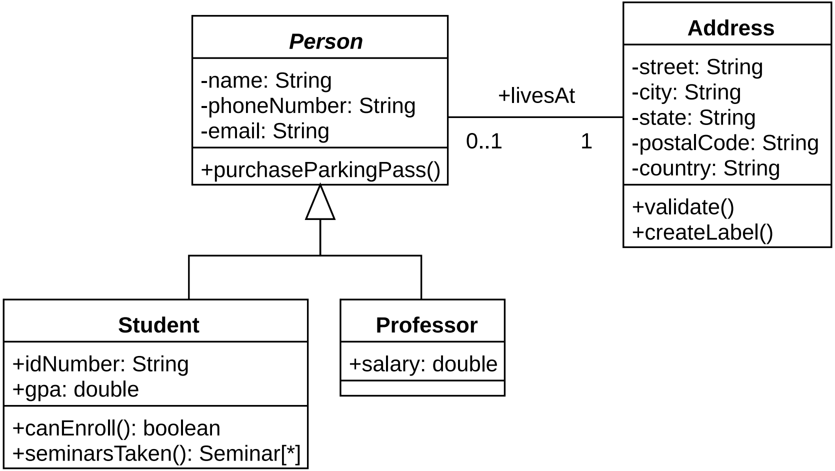
- If the cinema goes away
  - so does the box office: composition
  - but movies may still exist: aggregation



Example: Video Store



Example: People



Person

-name: String
-phoneNumber: String
-email: String

+purchaseParkingPass()

+livesAt

0..1

1

Address

-street: String
-city: String
-state: String
-postalCode: String
-country: String

+validate()
+createLabel()

Student

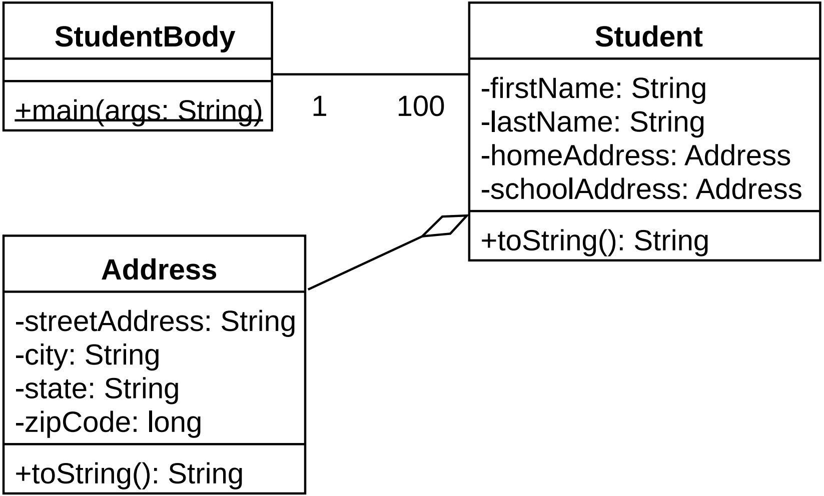
+idNumber: String
+gpa: double

+canEnroll(): boolean
+seminarsTaken(): Seminar[*]

Professor

+salary: double

Example: Student



StudentBody

+main(args: String)

1

100

Student

-firstName: String
-lastName: String
-homeAddress: Address
-schoolAddress: Address

+toString(): String

Address

-streetAddress: String
-city: String
-state: String
-zipCode: long

+toString(): String

Tools

- Violet UML (free)
  - http://horstmann.com/violet/
- StarUML (semi-free)
  - http://staruml.io
- LucidCharts (free for student use)
  - http://lucidcharts.com
- Rational Rose
  - http://www.rational.com/
- There are many others, but most are commercial

When to Use

- Class diagrams are great for:
  - discovering related data and attributes
  - getting a quick picture of the important entities in a system
  - seeing whether you have too few/many classes
  - seeing whether the relationships between objects are too complex, too many in number, simple enough, etc.
  - spotting dependencies between one class/object and another

When Not to Use

- Class diagrams are not so great for:
  - discovering algorithmic (not data-driven) behavior
  - finding the flow of steps for objects to solve a given problem
  - understanding the app’s overall control flow (event-driven? web-based? sequential? etc.)

Summary

- A design specifies the structure of how a software system will be written and function
- UML is a language for describing various aspects of software designs
- UML class diagrams present a static view of the system, displaying classes and relationships between them.

Project Iteration 2

The details of what you need to do for Iteration 2 will be discussed.

Are there any questions?