Objects, UML, and Java

Suleman Shahid
suleman.shahid@lums.edu.pk

Abdul Ali Bangash
abdulali@lums.edu.pk

Department of Computing Science

LUMS University

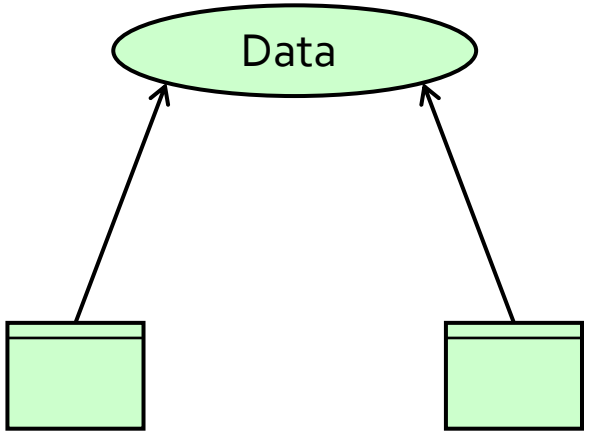
CMPUT 360 – Introduction to Software Engineering

Slides adapted from Henry Tang, Dr. Abram Hindle

Modeling Principles

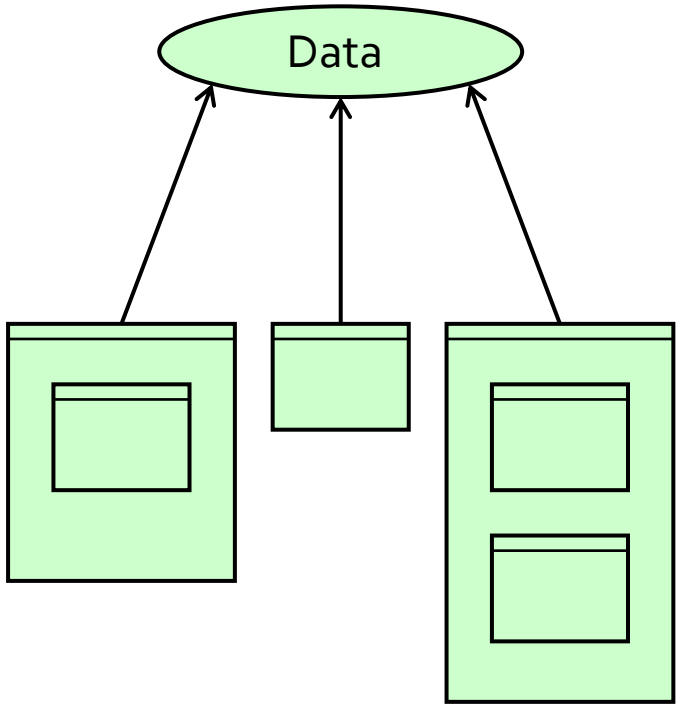
Language Evolution

- COBOL, Fortran:
  - Subprograms (subroutines) access global data
  - Break up system into subroutines



Language Evolution

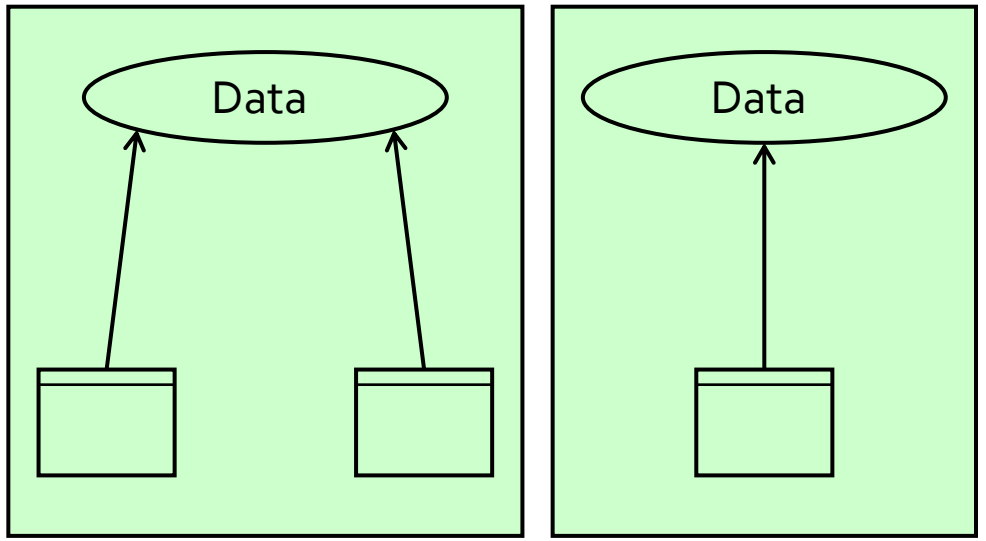
- Algol, Pascal:
  - (Nested) procedures with block structured scope
  - Break up system into nested procedures



Nested procedures

Language Evolution

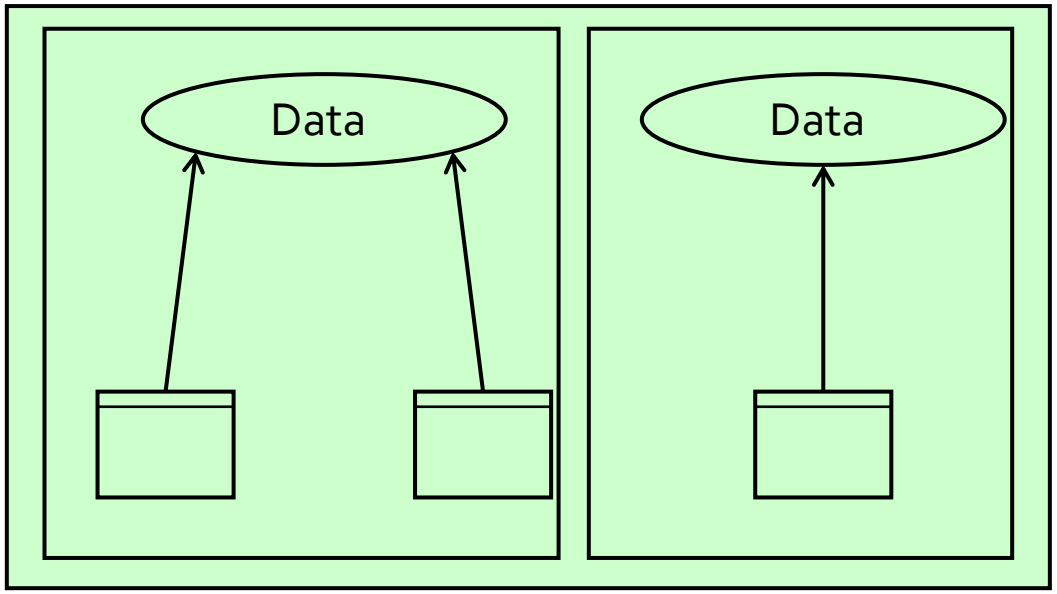
- Modula-2, C:
  - Modules (files) of related data and functions
  - Break up system into modules (e.g., abstract data types)



Modules

Language Evolution

- Smalltalk, C++, Java:
  - Classes with data and methods
  - Classes as “factories” for objects
  - Break up system into classes



Classes and packages

Discussion

- Question:
  - What software engineering design principles drove programming language evolution?
  - https://www.altexsoft.com/blog/pros-and-cons-of-java-programming/

Abstraction

- Simplifying to its essentials the description of a real-world entity or concept
  - Coping with complexity
  - “Selective ignorance”
- Modeling the problem space
  - E.g., a “Person” abstraction

Encapsulation

- Bundling data with access functions
  - Distinguishing “what” from “how”
  - “Need to know” restricted access
  - Maintaining integrity
- Information hiding criterion
  - Hide changeable internal details from the outside world, but reveal assumptions through interface
  - E.g., a “Person” abstract data type

Decomposition

- Dividing whole things into parts
  - Or composing whole things out of parts
  - “Separation of concerns”
- Data parts
  - Fixed or dynamic number
  - Sharing of parts
  - Lifetime of parts

Generalization

- From specific cases, looking for commonalities that can be factored out
  - Reusing common designs
  - Reducing redundant code
- Making systems flexible and extensible

Object-Oriented Models

Object-Oriented Models

- Implementing OO models:
  - OO programming languages
    - E.g., Java, C++
- Expressing OO models:
  - OO design notations
    - E.g., UML

Java

- Principal designer:
  - James Gosling, Sun Microsystems
- Language goals:
  - Simple, object-oriented
  - Robust, secure
  - Network and thread support
  - “Compile once, run anywhere”

Java

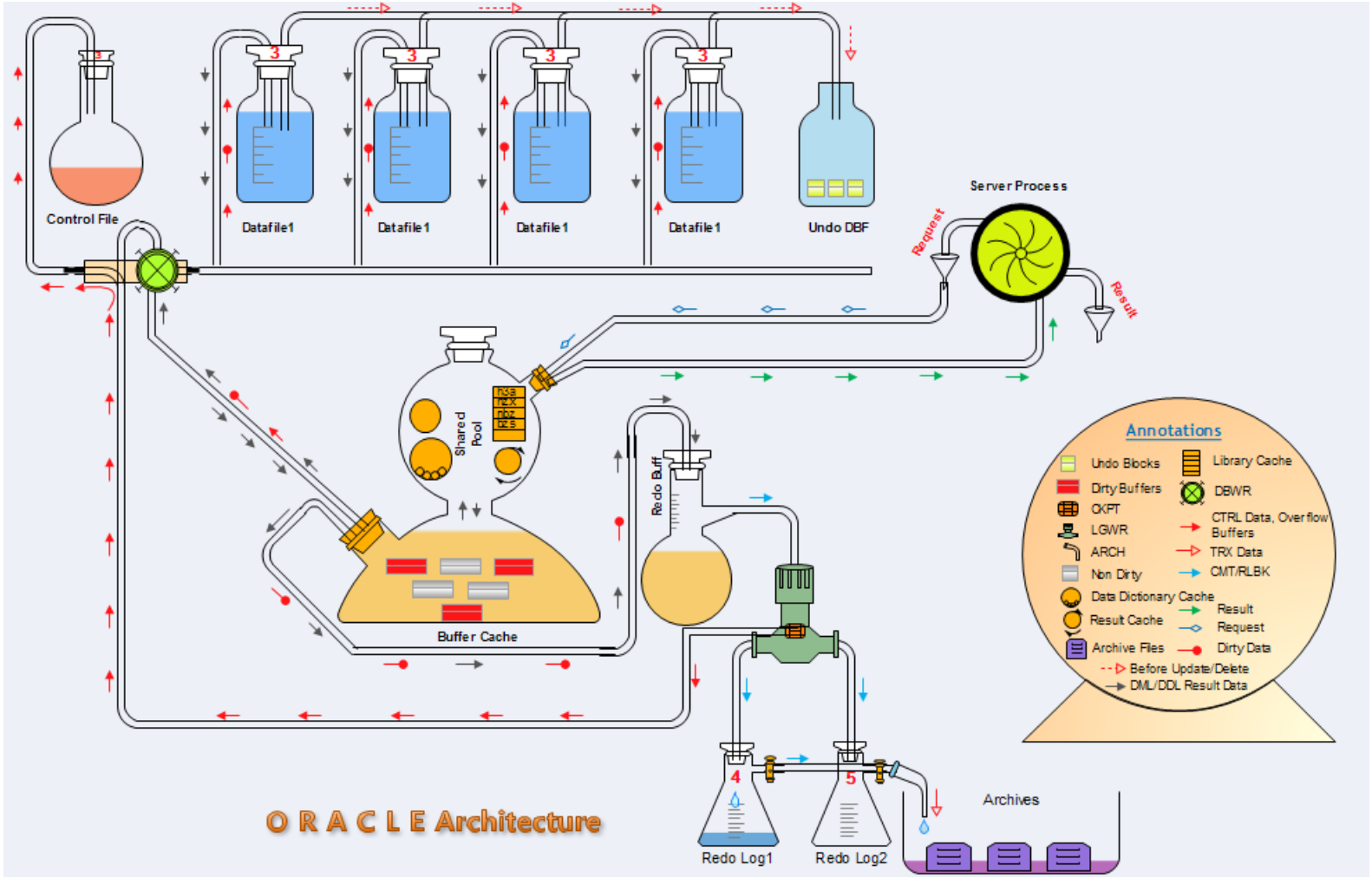
- Language design inspired by:

Language	Feature
Lisp	Garbage collection, reflection
Simula-67, C++	Classes
Algol-68	Overloading
Pascal, Modula-2	Strong type checking
C	Syntax
Ada	Exceptions
Objective C, Eiffel	Interfaces
Modula-3	Threads

Unified Modeling Language (UML)

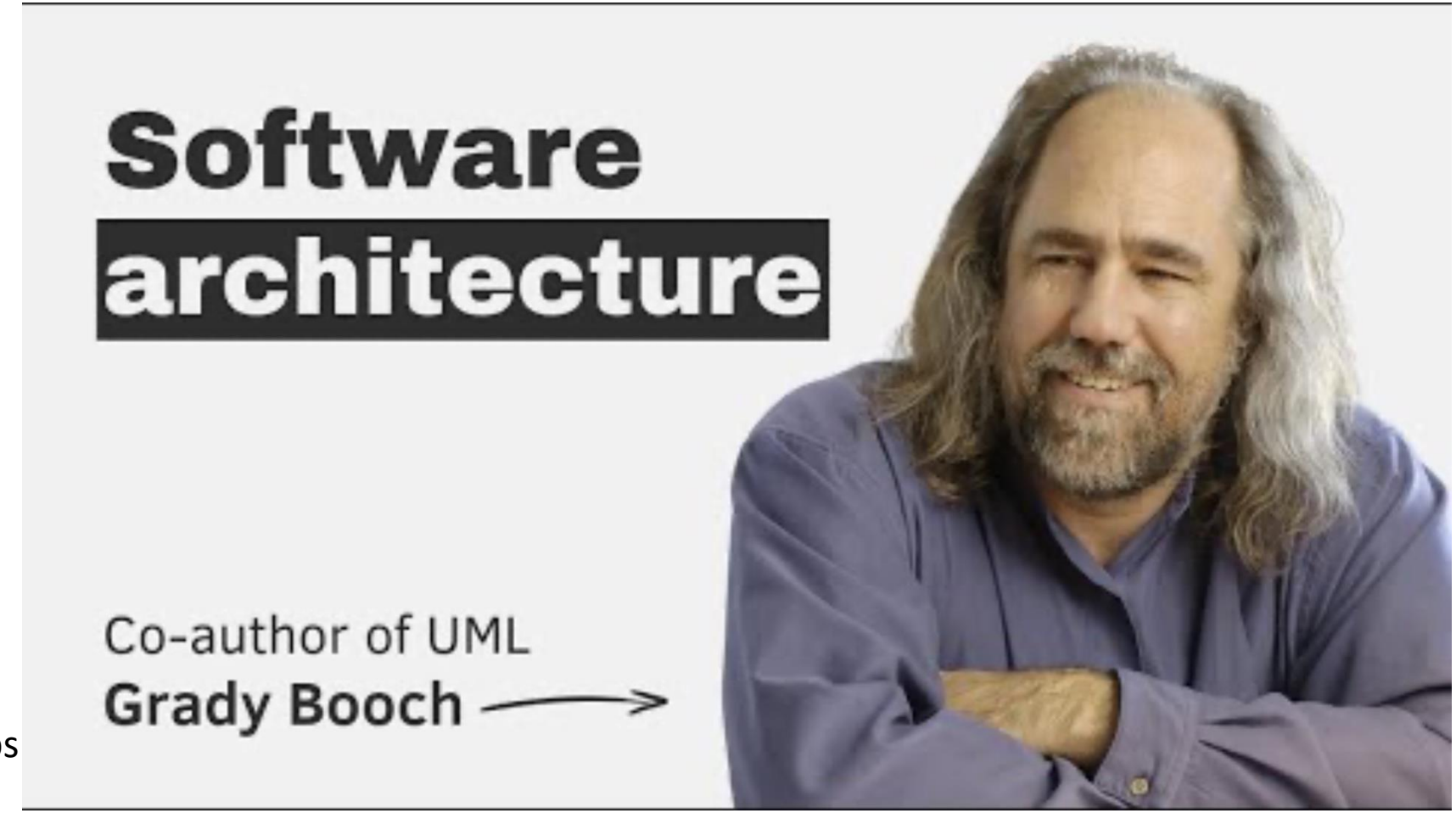
- Principal designers:
  - Grady Booch, Ivar Jacobson, James Rumbaugh
- Language goals:
  - Express object-oriented designs visually
  - Programming language independent
  - Communicate, evaluate, and reuse designs
  - Make design intent more explicit
- Allows thought about design before coding

Why UML?


Control File
Datafile1
Datafile1
Datafile1
Datafile1
Undo DBF
Server Process
Request
Result
Shared Pool
Buffer Cache
Redo Buff
Annotations
Undo Blocks
Dirty Buffers
CKPT
LGWR
ARCH
Non Dirty
Data Dictionary Cache
Result Cache
Archive Files
Library Cache
DBWR
CTRL Data, Overflow Buffers
TRX Data
CMT/RLBK
Result
Request
Dirty Data
Before Update/Delete
DML/DDL Result Data
ORACLE Architecture
Redo Log1
Redo Log2
Archives

But do we need UML?

https



Abstraction

- Object:
  - An entity with specific attribute values (state), behavior, and identity
  - Typically instantiated from a class
- Class:
  - Associated type of an object
  - Defines attributes and methods

Java and UML Class

```
public class Frame { // version 0
    // represent a 'window'
    /* body of class definition goes here */
}
```

UML class notation

Encapsulation

- Class:
  - Access control for attributes and methods
    - E.g., public or private
  - Access is not the same as visibility
  - “Design by contract”
    - Public interface represents a contract between the developer who implements the class and the developer who uses the class

Java Class

```
• public class Frame { // version 1
      // private implementation

      private datatype variablename;

      // public interface

      public Frame( arguments ) {
          // implementation of constructor
      }

      public returntype methodname( arguments ) {
          // implementation of method
      }
  }
```

Java Class

```
public class Frame { // version 2
    private int x;
    private int y;
    …
    public Frame ( String name,
        int x, int y, int height, int width ) { … }

    public void resize(
        int newHeight, int newWidth ) { … }

    public void moveTo(
        int newX, int newY ) { … }

    public void drawText( String text,
        int x, int y ) { … }
}
```

```
Frame f = new Frame( "Untitled", 0, 0, 480, 640 );
```

UML Class

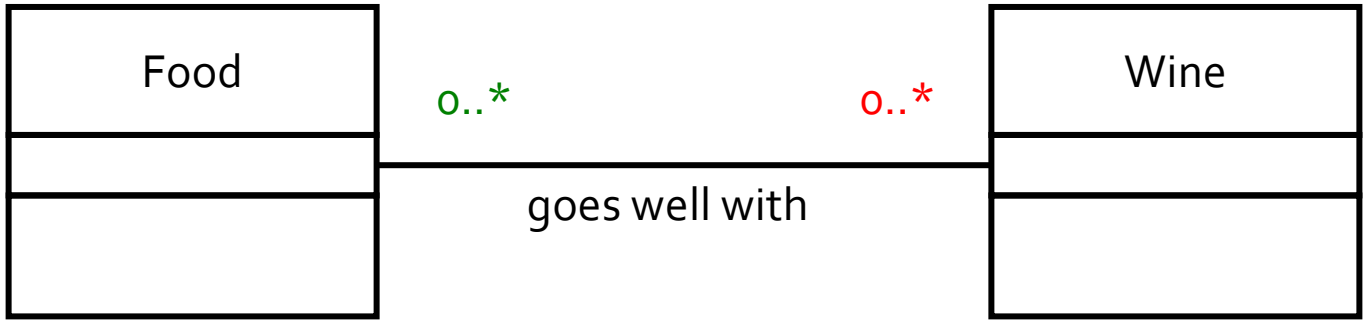
Frame
-x : Integer -y : Integer
+Frame(name: String, x: Integer, y: Integer, height: Integer, width: Integer) +resize(height: Integer, width: Integer): void +moveTo(x: Integer, y: Integer): void +drawText(text: String, x: Integer, y: Integer): void

- – Private
- \+ Public

Decomposition

- Association relationship:
  - "Some" relationship between classes
    - E.g., between Book and Patron

UML Association



- Read class diagram using "objects"
  - A Food *object* goes well with a Wine *object*
  - A Food *object* is associated with o or more Wine *objects*
  - A Wine *object* is associated with o or more Food *objects*

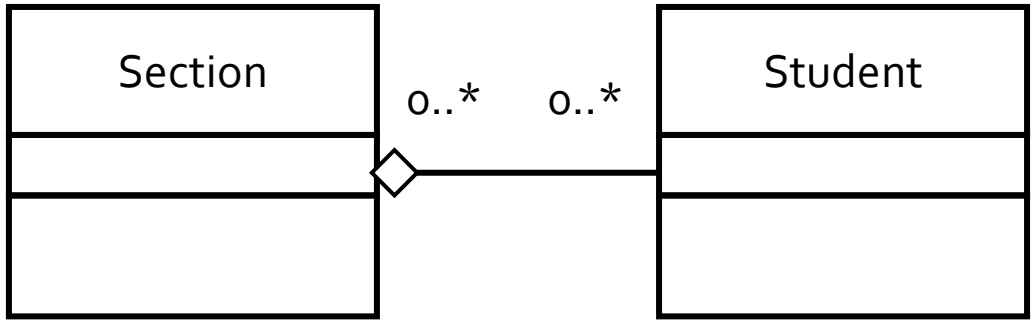
Decomposition

- Aggregation relationship:
  - Weak “has-a” relationship
  - Whole “has-a” part
  - A part may belong to (be shared with) other wholes
    - E.g., a Section and a Student

Java and UML Aggregation

- Dynamic number of aggregated objects:

```
public class Section {
    private ArrayList<Student> roster;
    …

    public Section() {
        roster = new ArrayList<Student>();
        …
    }
    public void add( Student s ) { … }
}
```



Java and UML Aggregation

- Fixed number of aggregated objects:

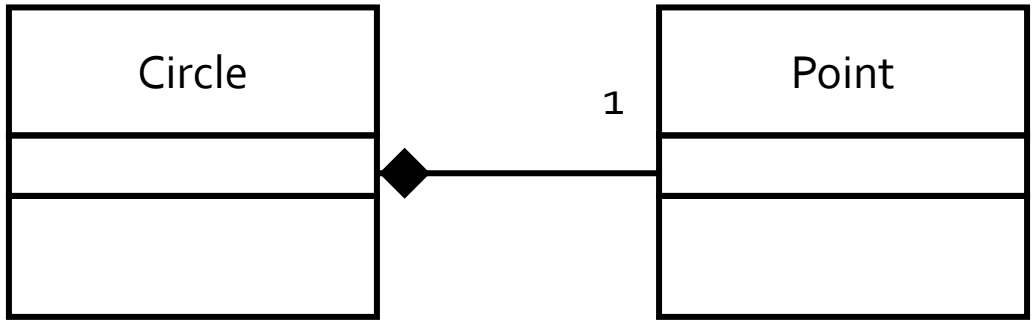
```
public class Frame {

    private Location defaultLocation; // shared
    private Size defaultSize; // shared
    …
}
```

Decomposition

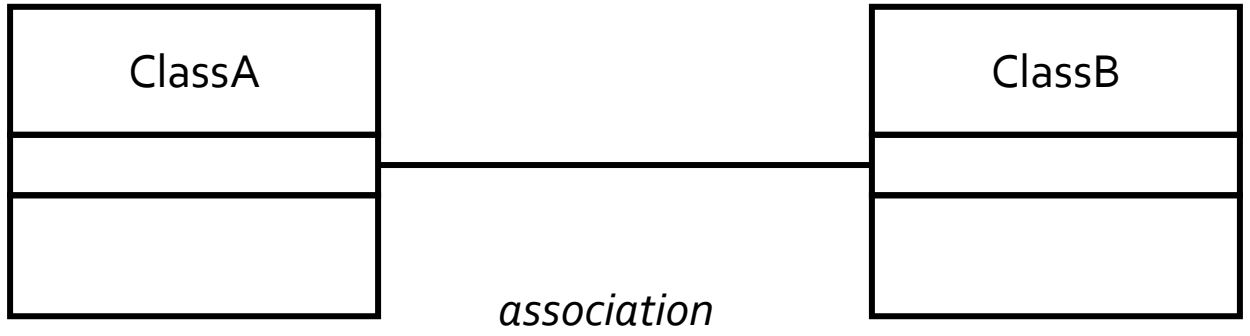
- Composition relationship:
  - Strong “has-a” relationship
  - Exclusive containment of parts
  - Related object lifetimes
    - The whole cannot exist without having the parts; if the whole is destroyed, the parts should also be destroyed
  - Often access the parts through the whole

UML Composition

- Contained *objects* are exclusive to the container

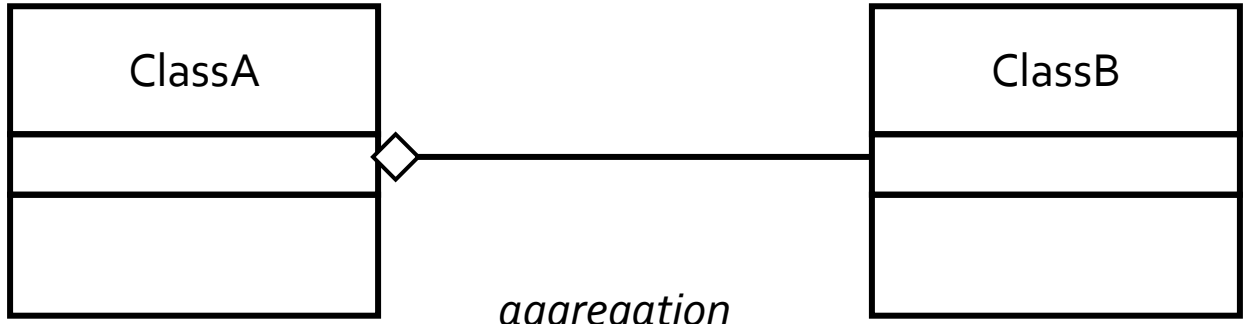


- A Circle object has a Point object that is exclusive to it (however, other objects may contain Point objects, just not this one)



association

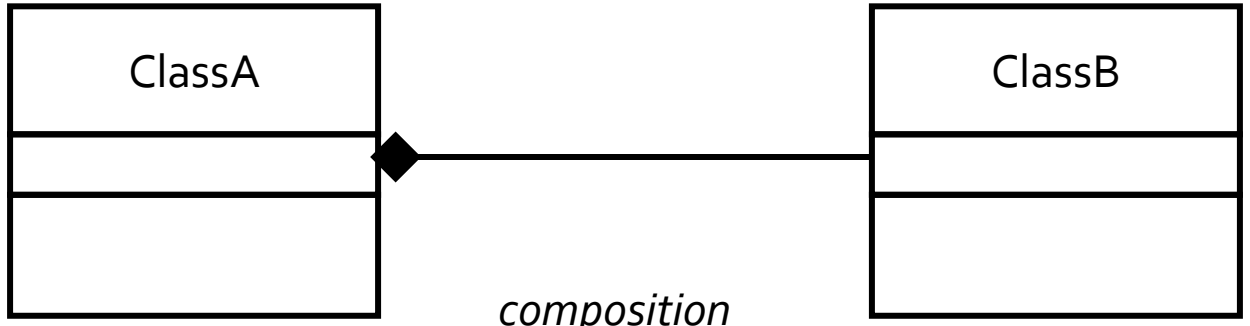
whole *part*



aggregation

ClassA and ClassB can created/destroyed independently

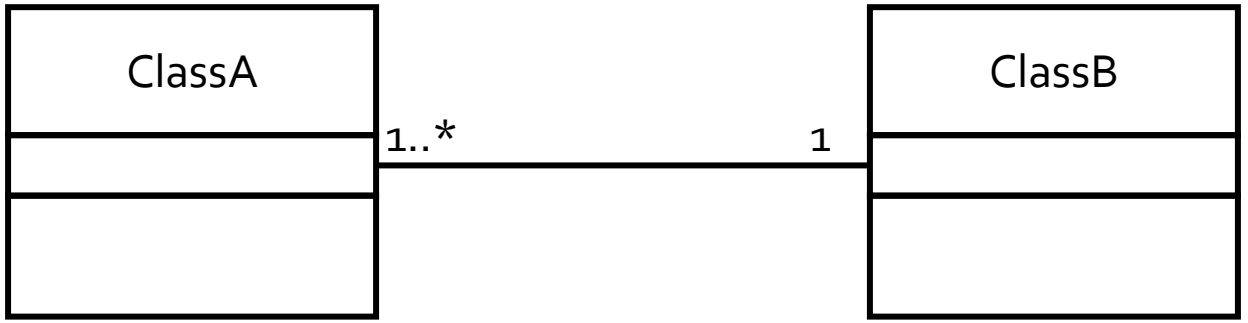
whole *part*



composition

If ClassA instance is deleted, all of its ClassB instances get deleted too

Navigability

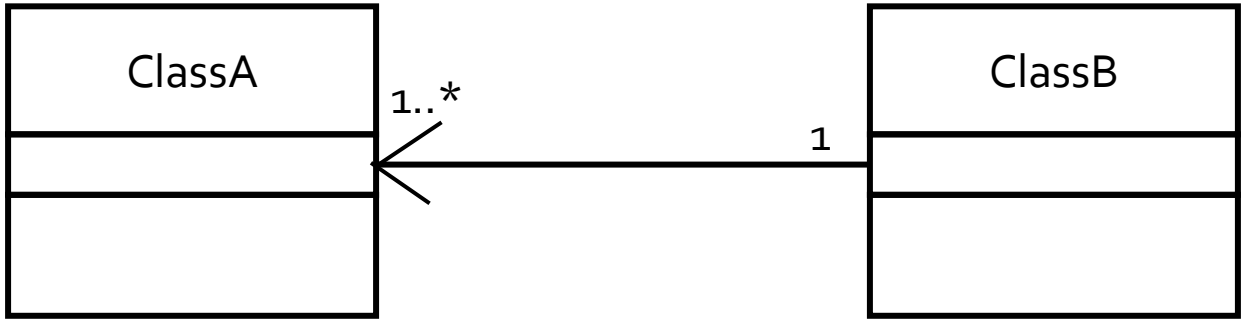


Missing navigability

Usually implies:

A instances have references to one B

B instances have references to one or more A

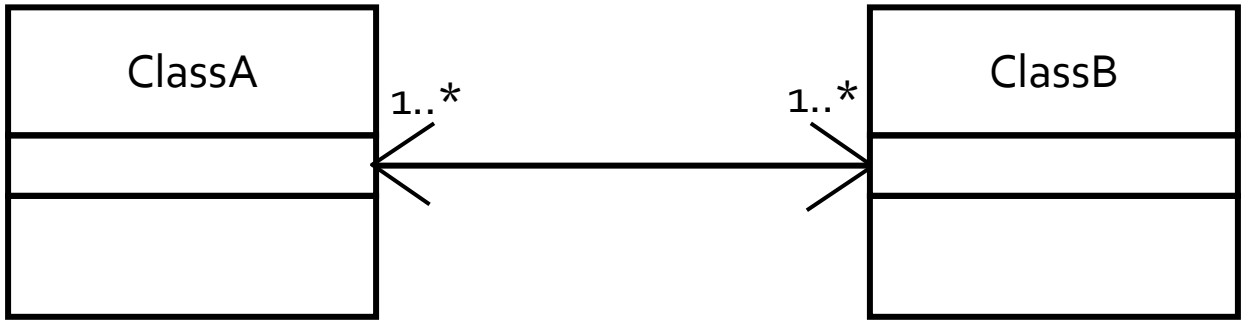


Navigability

← Arrow

B instances have references to one or more A

A instances DO NOT have reference(s) to B



Explicit two-way navigability

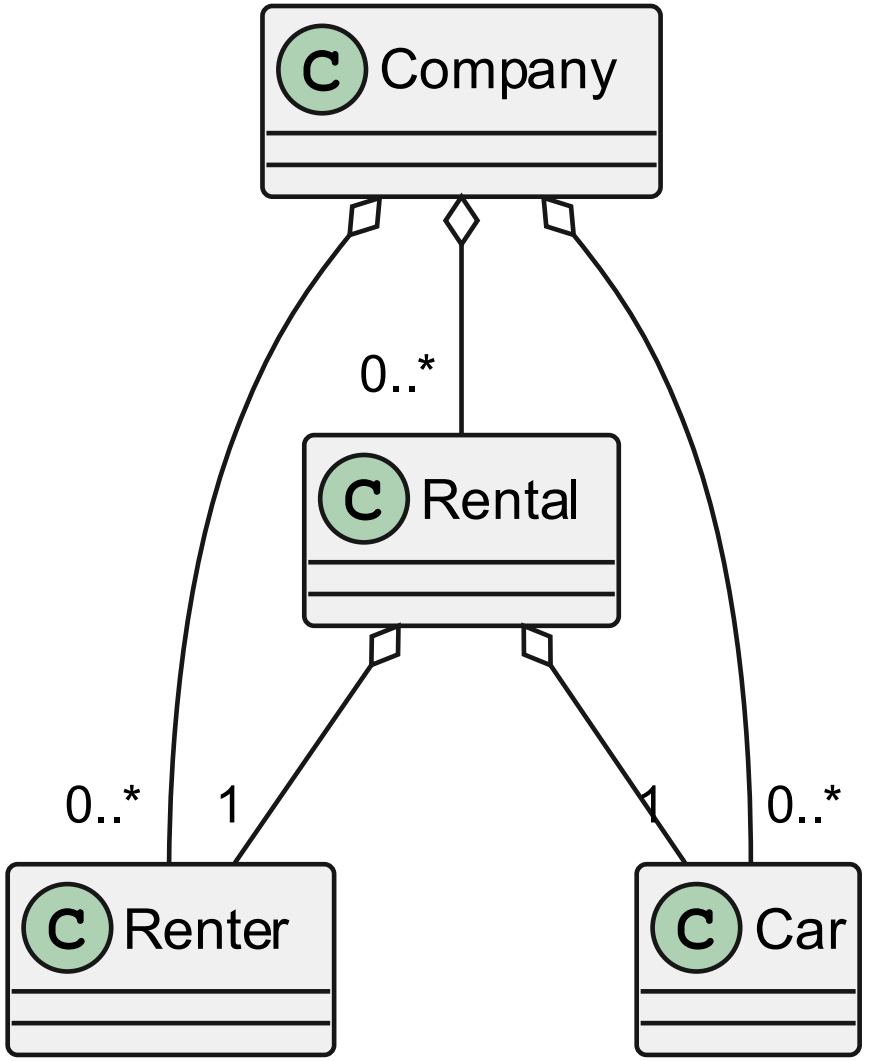
(rare)

A instances have references to one or more B

B instances have references to one or more A

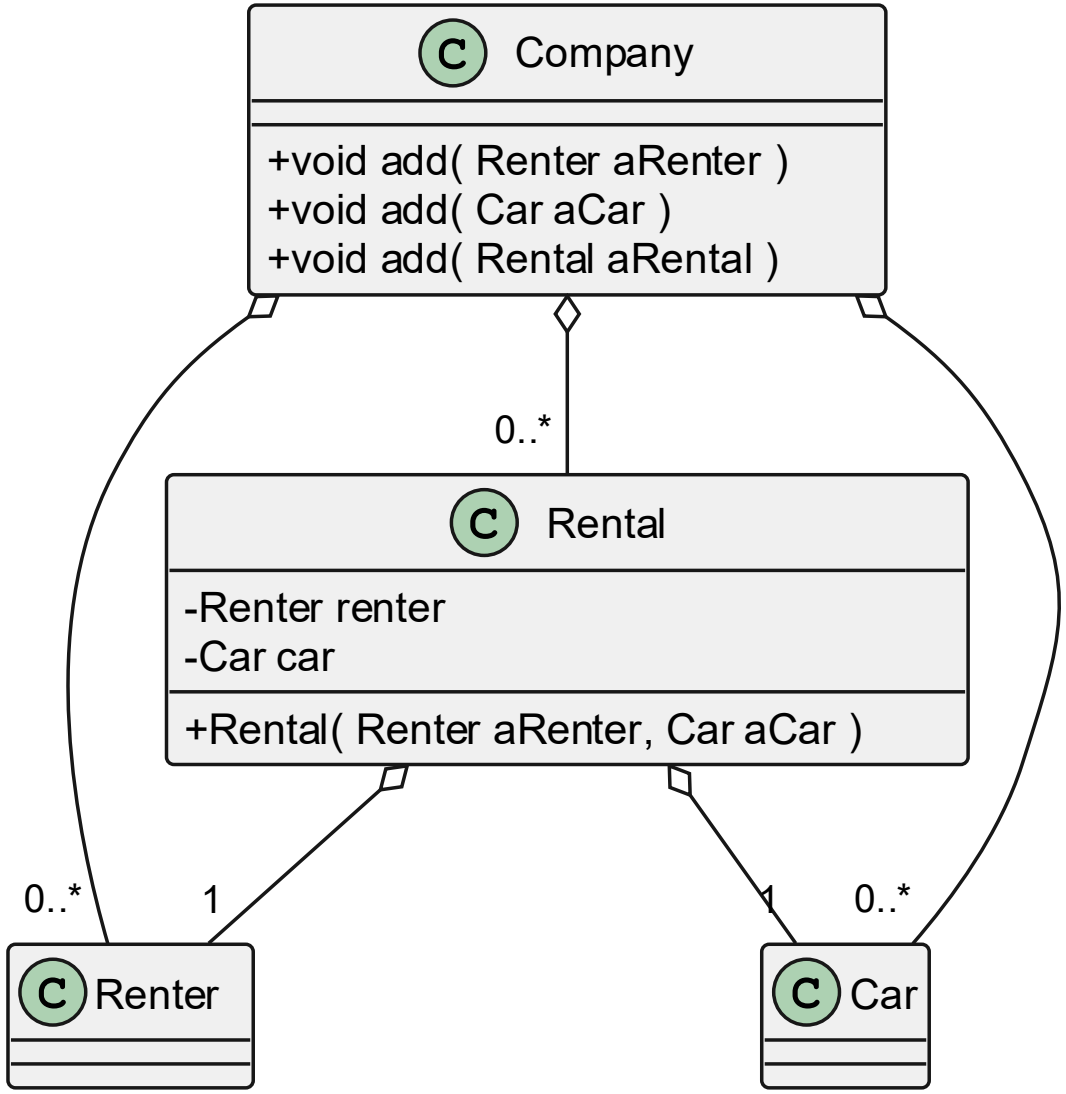
Exercise

- Analyze a UML class model for a car rental company that keeps track of cars, renters, and renters renting cars.


Company
0..*
Rental
0..*
1
1
0..*
Renter
Car

Exercise

- Implement the corresponding Java code.



More Information

- Link:
  - UML Quick Reference
    - http://www.holub.com/uml/
- Books:
  - Book: Developing applications with Java and UML
  - Book: Java Design: Objects, UML and Process
  - Book: Object-oriented design with UML and Java