SE 350 - Object-Oriented Software Development

Software Life Cycle and Documentation

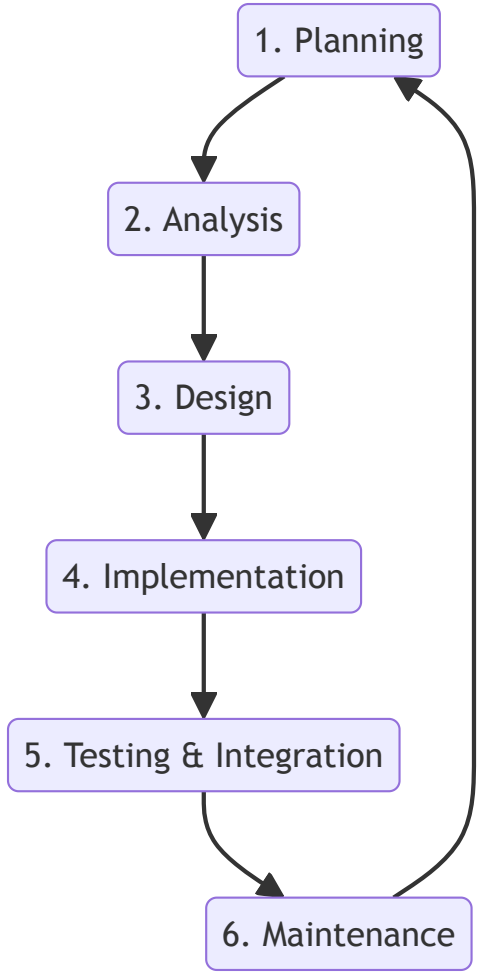
Instructor: Stefan Mitsch

Learning Objectives

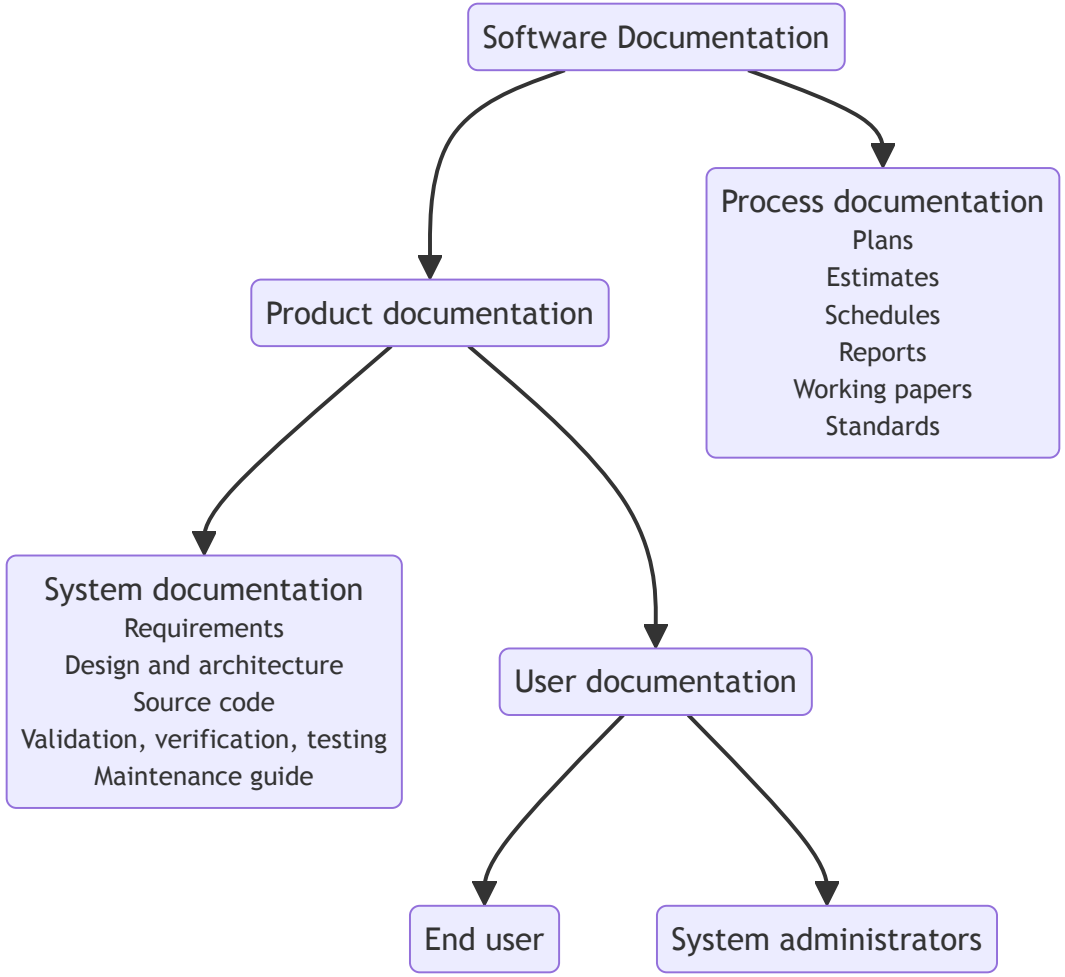
- Understand software development life cycles
- Understand documentation types
- Build UML modeling skills

Software Development Life Cycle

Stage	Activity	Key Roles	Deliverables
Planning	Preliminary requirements, analysis, research, vision and scope	Customer, sales, analyst	Understanding the customer needs, basic project roadmap and technical recommendations
Analysis	Identify project goals, functionality, specifications, requirements	Customer, analyst, tech experts, project managers	Complete functional and design specification, work breakdown structure, initial cost estimate
Design	Define system architecture, UI design	Architect, UI/UX, project manager	Draft system architecture, product design, refined cost estimate
Implementation	Software development	Software engineer, architect, project manager	Functioning software product
Testing	Quality assurance	Software engineer, test engineer, project manager, customer	Finalized software product
Maintenance	Deployment, support, updates	Software engineer, project manager	Updated product



Types of Software Documentation



Design Documentation: UML

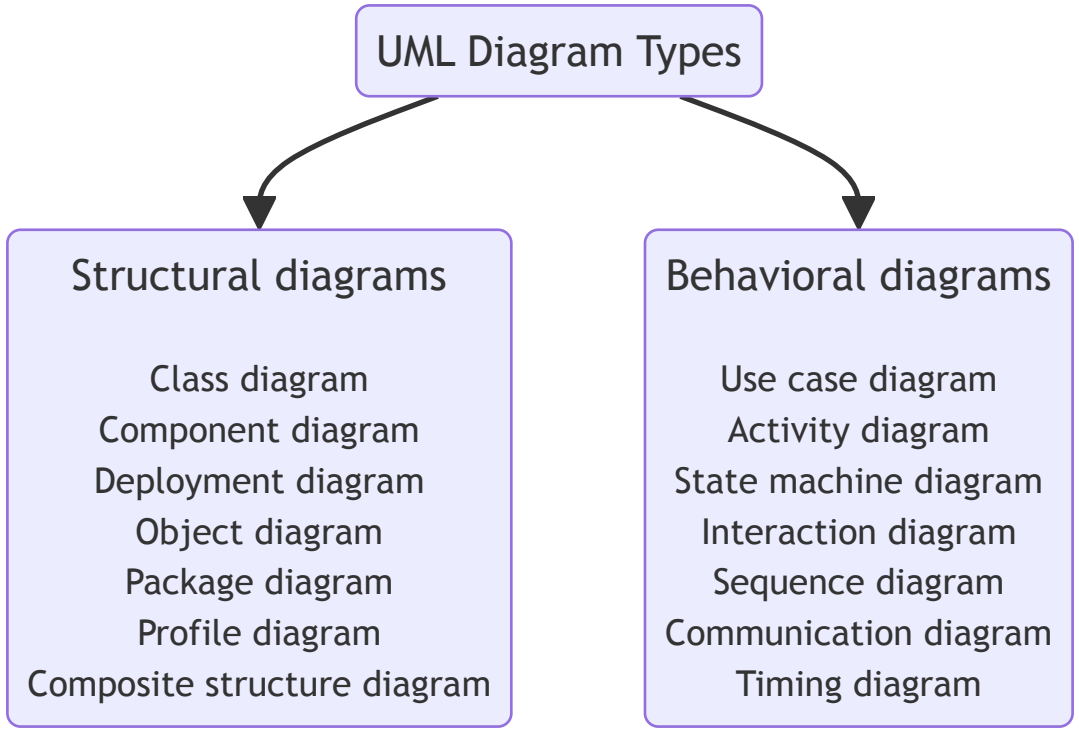
- **U**nified **M**odeling **L**anguage
  - visual modeling
  - specify, visualize, document
- History
  - Before 1995: fragmented methods
  - 1995: unification (Grady Booch, Ivar Jacobson, James Rumbaugh)
  - 1997: UML 1.0 (HP, IBM, Microsoft, Oracle, ...)
  - 1998: standardization
  - 1999: industrialization
  - Today: UML 2.5.1

UML Building Blocks

- Things
  - Structural: class, object, interface, use case, actor, component, node
  - Behavioral: state machine, activity, interaction
  - Grouping: package
  - Annotational: note
- Relationships
  - Dependency, association
  - Generalization, realization

UML Diagram Types

- Structure
  - Classes, objects
  - Documents *static* aspect of software
- Behavior
  - Interaction between software elements and users
  - (Information) flow between components
  - Documents *dynamic* aspect of software

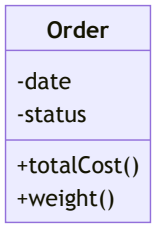


UML Class Diagram

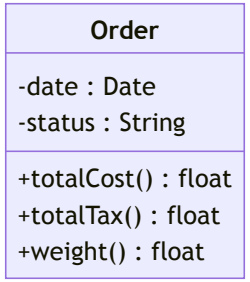
- **Static view** of software: classes and their relationships between them
- **Documents** analysis and design, responsibilites
- Builds a **dictionary** for other diagrams
- **Communication** with non-development stakeholders

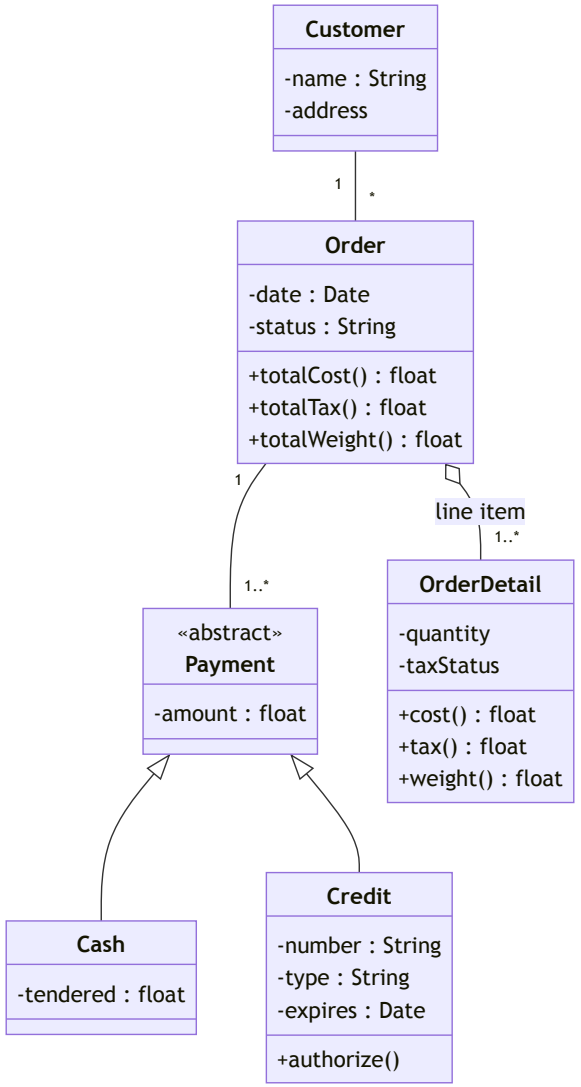
Analysis

Design



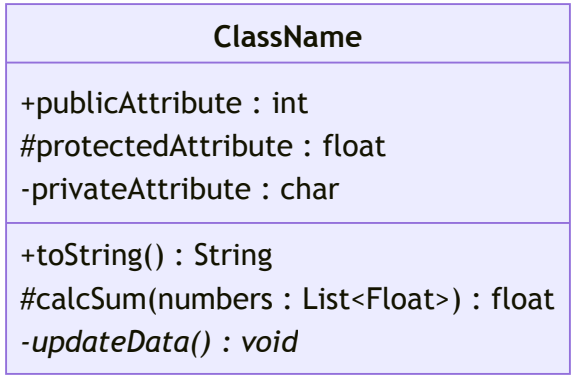
Implement





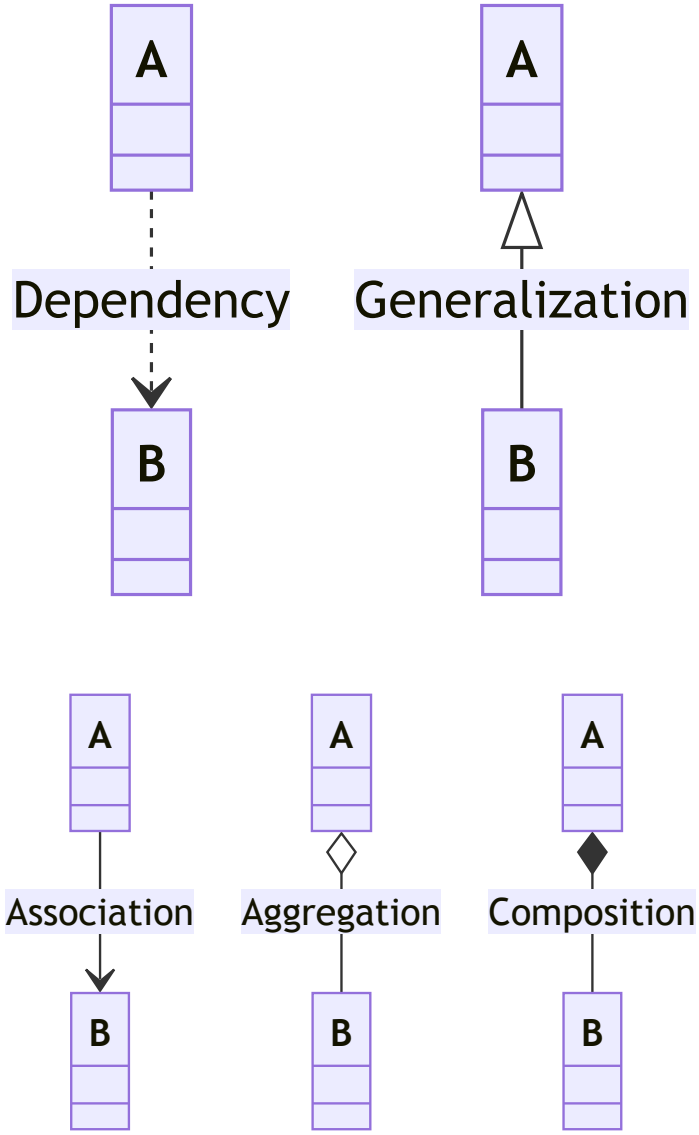
UML Class Diagram Components

- Top section: Class name
  - Abstract class: *italics* or `<<abstract>>`
- Middle section: Attributes
  - Visibility: `+` public, `#` protected, `–` private
  - Type in `attributeName : type` notation
- Lower section: Methods
  - Visibility `+` public, `#` protected, `–` private
  - Argument types and return type in notation
    `methodName(arg1 : Type1, ...) : Type`
  - Abstract method: *italics*



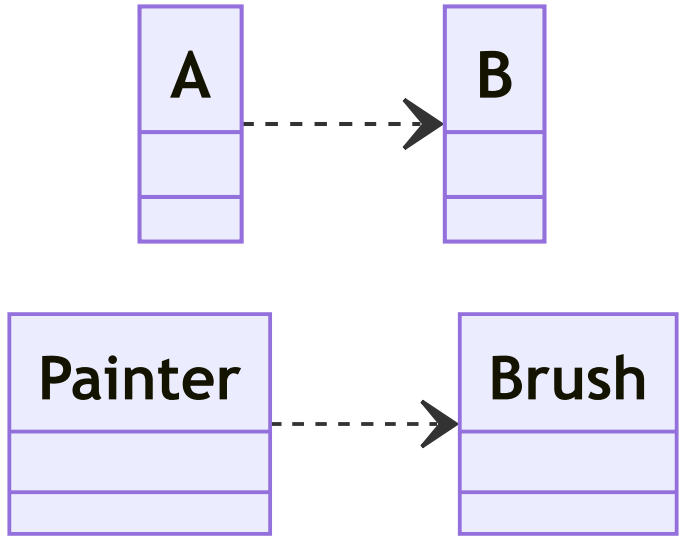
Class Diagram Relationships

- Dependency
  - One class depends on another class
  - `uses-a` relationship
- Generalization
  - Inheritance
  - `is-a` relationship
- Association
  - weak `has-a`
  - Aggregation: `has-a`, `is-part-of`
  - Composition
    - strong `has-a`, `belongs-to`
    - Deallocated together



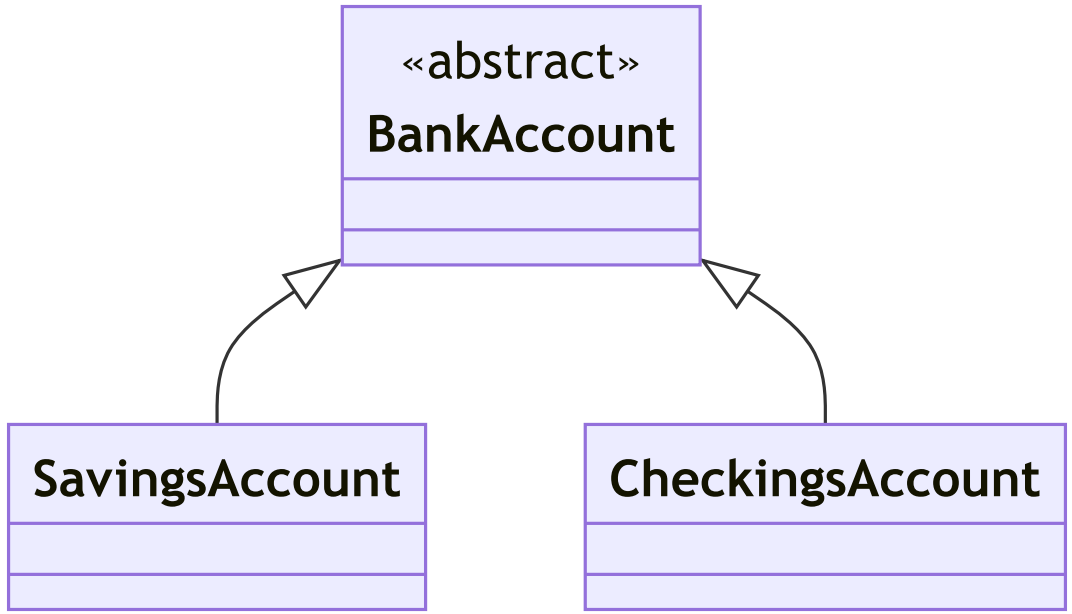
Dependency

- Denotes a weak dependency between classes
- **Temporary**, e.g., as argument to a method
- `uses–a` relationship (always directed)
- Example: `A uses–a B` method in a class temporarily uses object of another class
- Example: `A uses–a B` change in `B` may affect `A`



Generalization

- Inheritance between classes
- `is-a` relationship
- Generalization: factor common features of objects into base class
- Specialization: override common features in sub-class

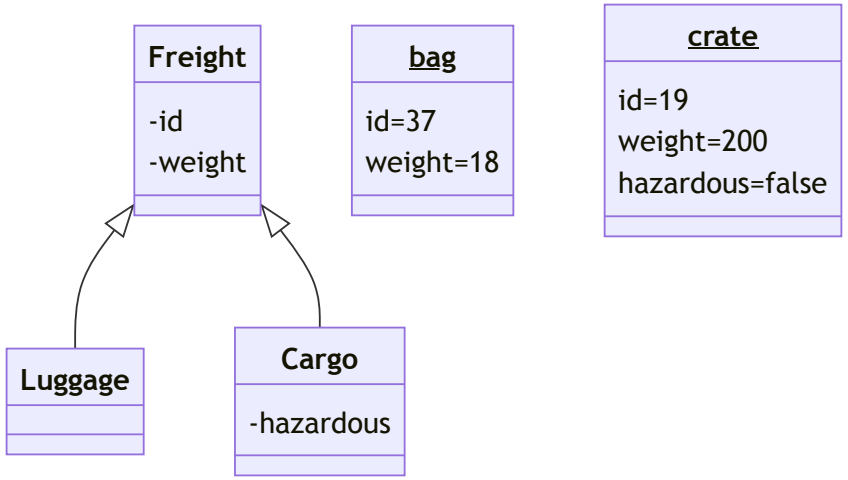


Generalization vs. Specialization

- Generalization: factor common features of objects into base class
- Specialization: override common features in sub-class

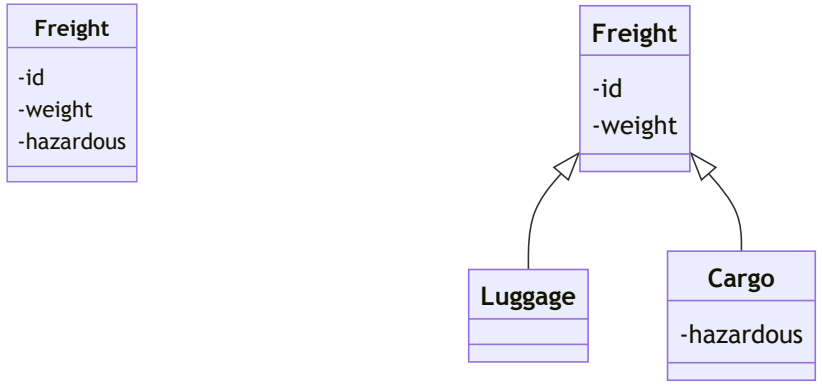
Generalization

Freight generalizes bag



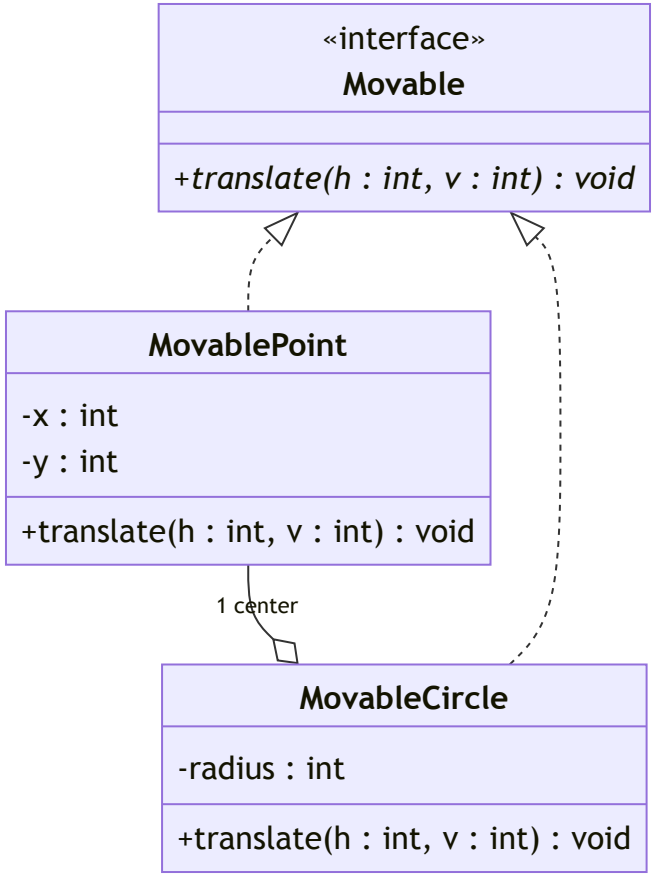
Specialization

Cargo specializes Freight



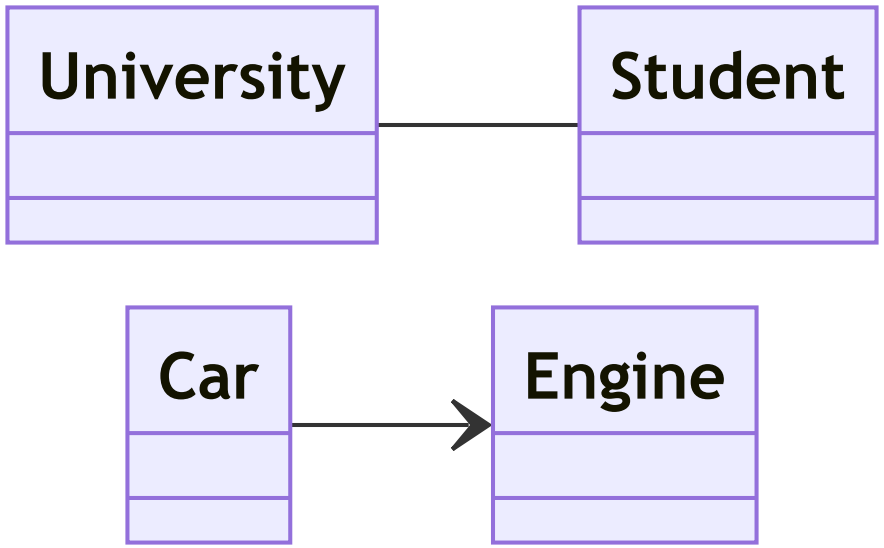
Realization and Implementation

- Class realizes/implements interface
- `A realizes B`, `A implements B`



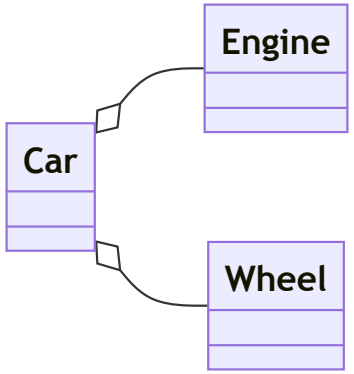
Association

- **Permanent structural relationship**, e.g., field with reference to another object
- Undirected: both classes know about each other
- Optional direction denotes `has-a` relationship: parent has reference to dependent class and can invoke its methods



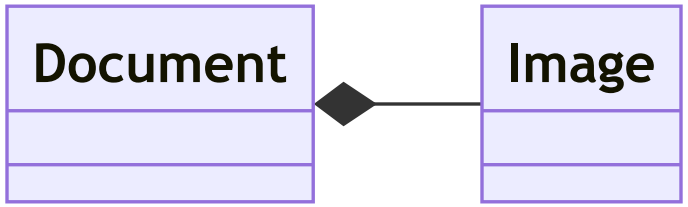
Aggregation

- Strong permanent structural relationship
- **Whole-part** `has-a` relationship
- Always directional: one class `has-a` other class
- Objects of both classes can survive individually
  - Can be allocated separately
  - Can change to different object, e.g., with setters

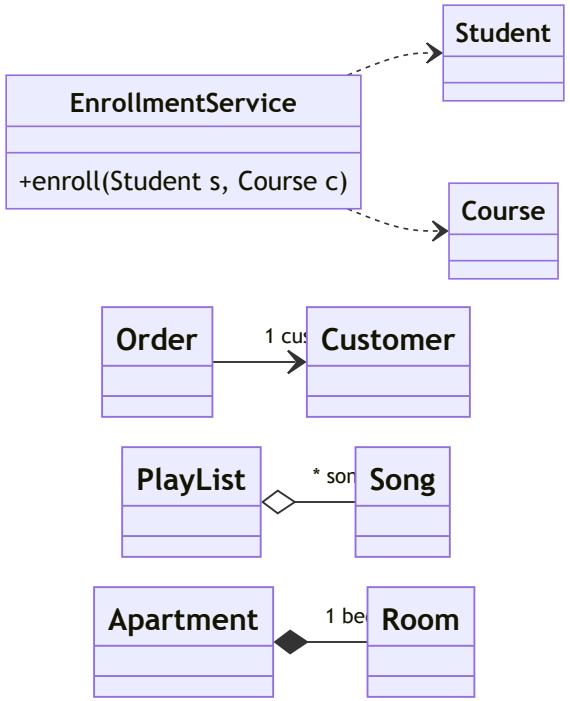


Composition

- Strong aggregation
- `has-a` relationship
- Mutually dependent classes
- Parent unusable without dependent class
  - Parent initialized with dependent object, e.g., in constructor
- Objects of dependent class destroyed with parent object



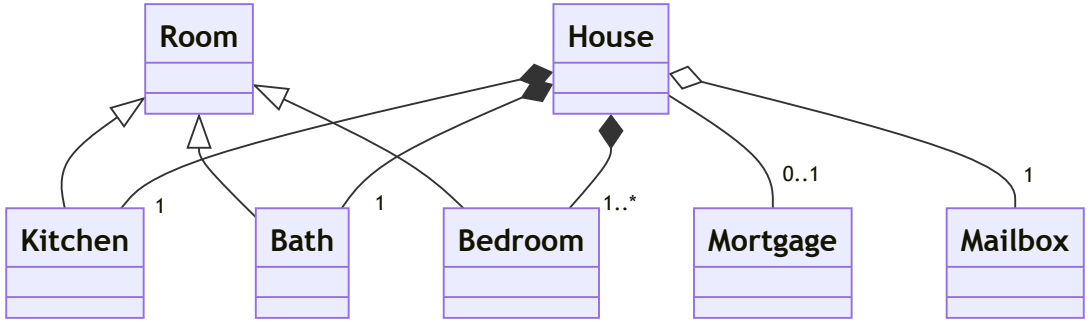
Relationship Examples



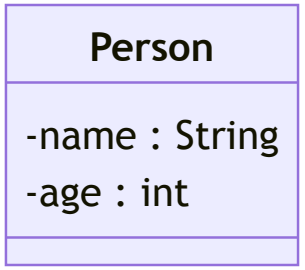

```java
// Dependency
public class EnrollmentService {
  public void enroll(s : Student, c : Course) {}
}

// Association (has-a)
public class Order {
  private Customer customer;
}

// Aggregation (has-a + whole-part)
public class PlayList {
  private List<Song> songs;
}

// Composition (has-a + ownership)
public class Apartment {
  private Room bedroom = new Room();
}
```

Multiplicity of Associations

- How many objects of a class are related to how many objects of other class
  - `*` 0 or more
  - `1` exactly 1
  - `2..5` between 2 and 5, inclusive
  - `3..*` 3 or more

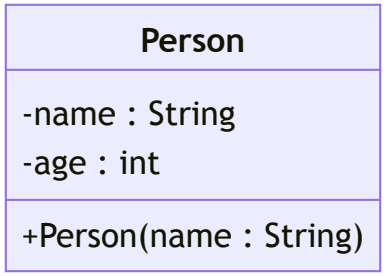


Class and Attributes



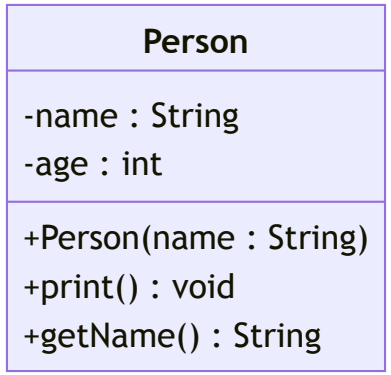

```java
public class Person {
  private String name;
  private int age;
}
```

Constructors




```java
public class Person {
  private String name;
  private int age;

  public Person(String name) {
    // unspecified
  }
}
```

Methods



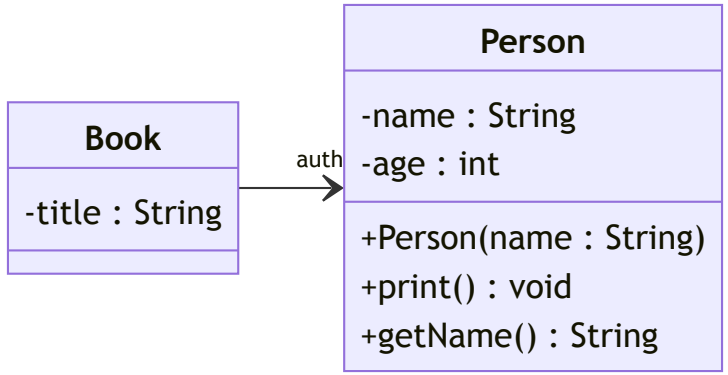

```java
public class Person {
  private String name;
  private int age;

  public Person(String name) {
    // unspecified
  }

  public void print() {
    // unspecified
  }

  public String getName() {
    // unspecified
  }
}
```

Class Member Visibility and Modifier Recap

Java	UML	Description
`public`	`+`	Can be accessed from any object
`private`	`-`	Only visible to objects of the defining class
`protected`	`#`	Visible to objects of the defining class or a subclass of it
package	`~`	Visible to objects of classes in the same package

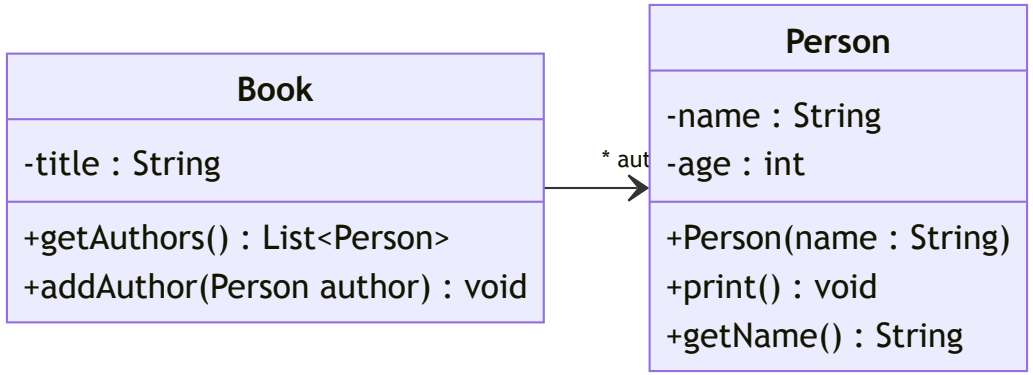
Connections between Classes

- Roles annotated close to their class, e.g., a `Person` has role `author` in association to a `Book`



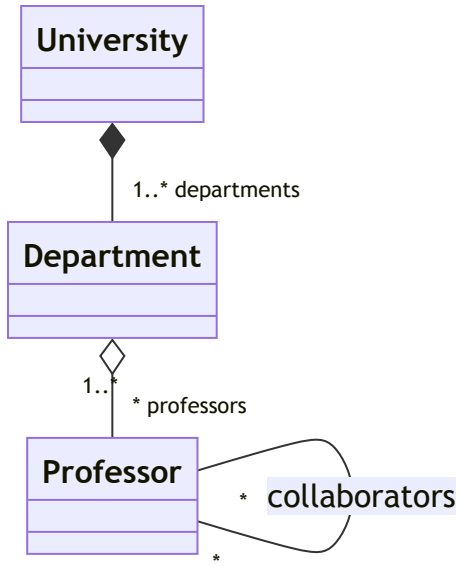

```java
public class Book {
  private String title;
  private Person author;
  // ...
}
```

Connections between Classes




```java
public class Book {
  private String title;
  private List<Person> authors;

  public List<Person> getAuthors() {
    // unspecified
  }

  public void addAuthor(Person author) {
    // unspecified
  }
}
```

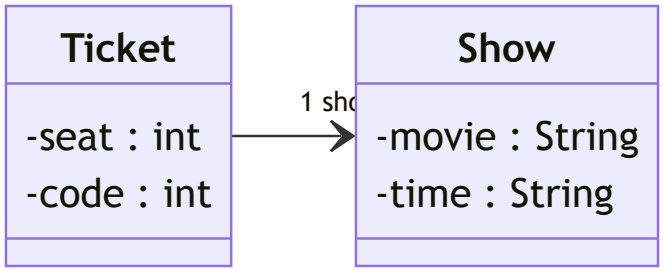
Associations, Aggregations, Compositions




```java
public class University {
  private List<Department> departments;
}

public class Department {
  private List<Professor> professors;
}

public class Professor {
  private Department department;
  private List<Professor> collaborators;
}
```

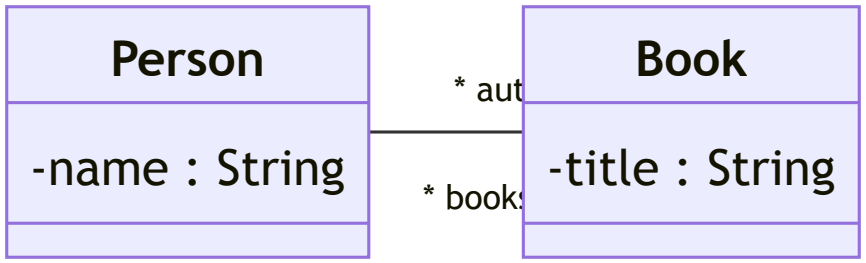
Exercise: Ticket and Show

- Implement the classes in the diagram in Java



▶ Solution

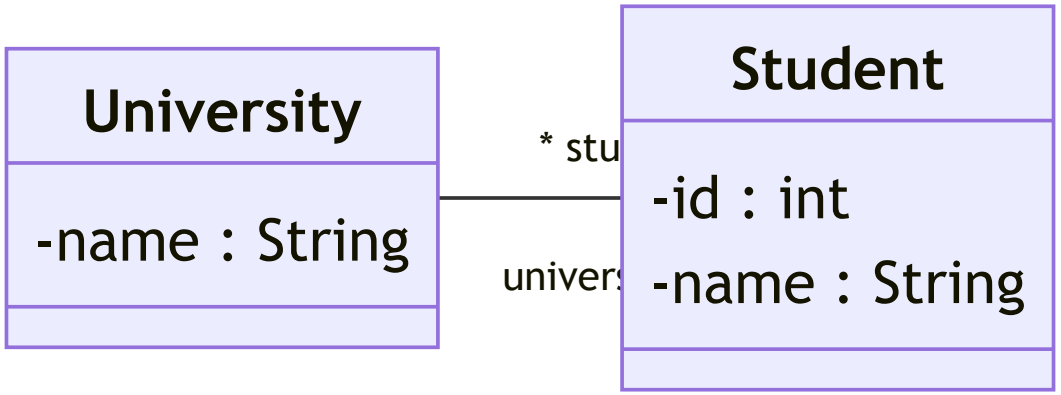
Exercise: Bidirectional Association



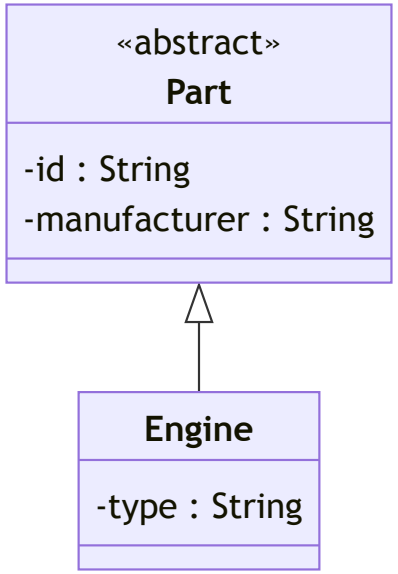
▶ Solution

Exercise: Student at University

▶ Solution

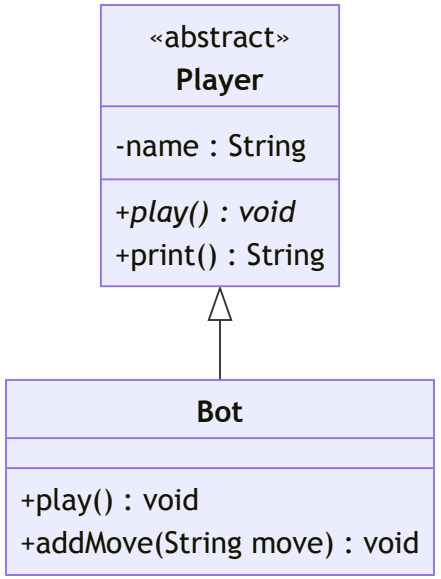


Exercise: Inheritance



▶ Solution

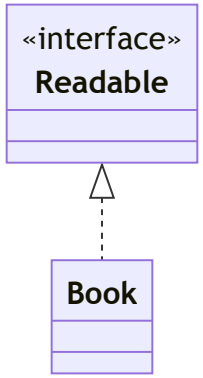
Exercise: Player and Bot



▶ Solution

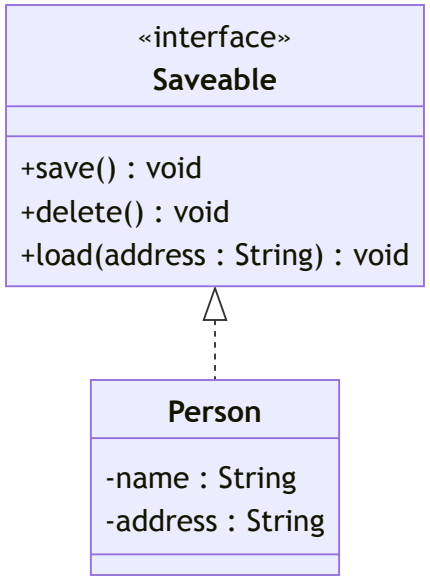
Interfaces

- Interfaces are identified with annotation `<<interface>>`, otherwise like classes



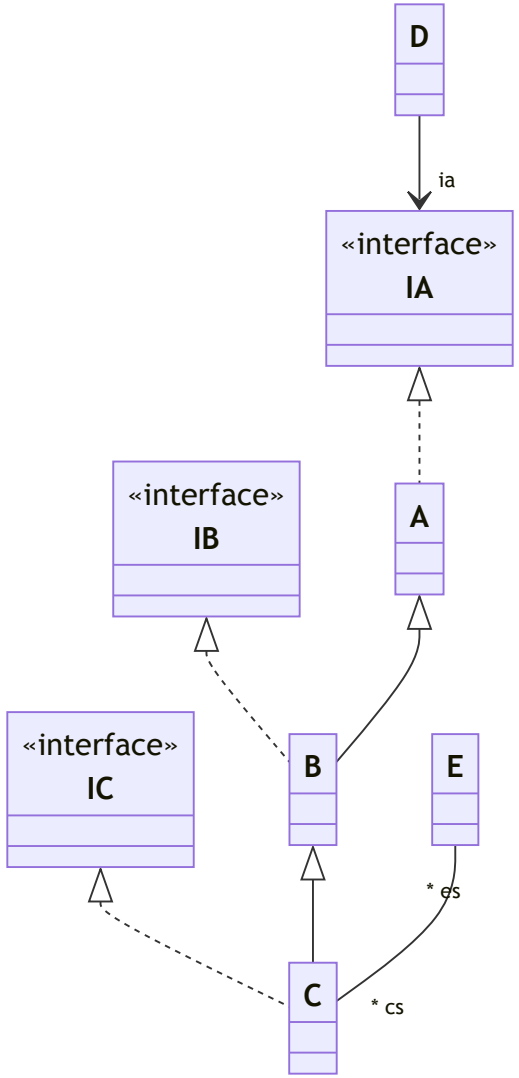

```
public interface Readable {

}

public class Book implements Readable {

}
```

Exercise: Implement an Interface



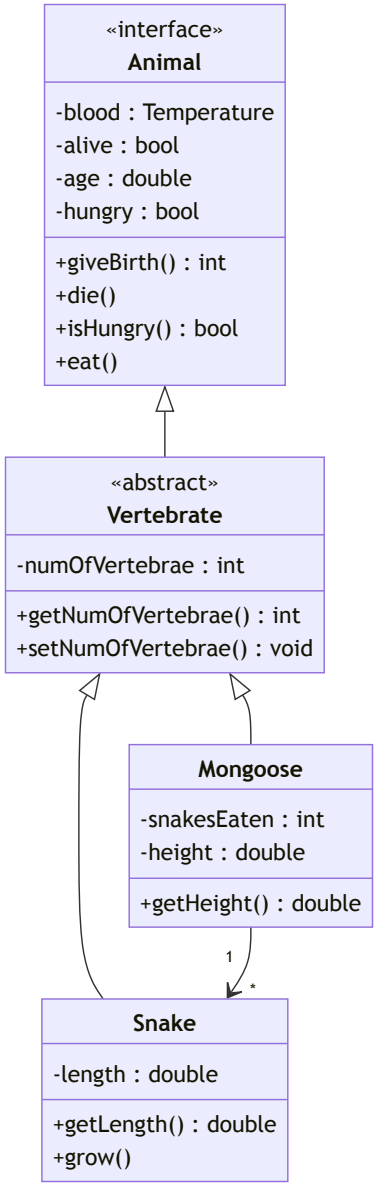
▶ Solution

Exercise: A Larger Example



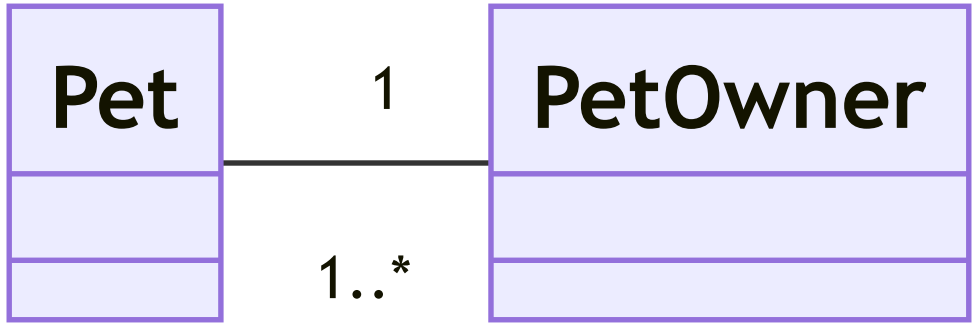
▶ Solution

Exercise: Read UML



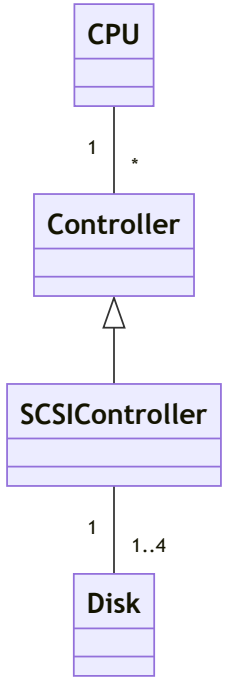
- Relationship between `Vertebrate` and `Animal`?
  - ▶ Solution
- What is wrong with this relationship?
  - ▶ Solution
- Relationship between `Snake` and `Animal`?
  - ▶ Solution
- Relationship between `Mongoose` and `Snake`?
  - ▶ Solution
- Can 2 `Mongoose` eat 1 `Snake`?
  - ▶ Solution

Exercise: Multiplicity



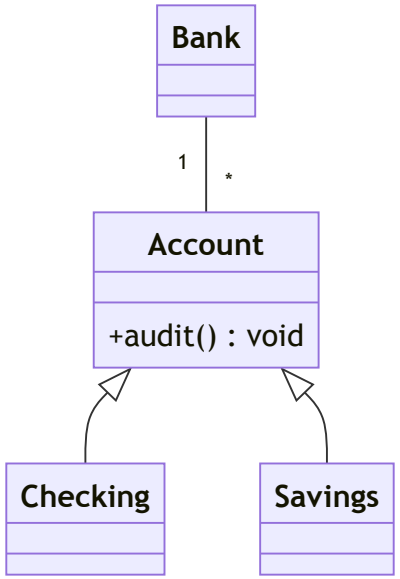
- What is the relationship between pet and owner?
  - ▶ Solution
- How many pets can an owner have?
  - ▶ Solution
- How many owners can a pet have?
  - ▶ Solution

Exercise: Multiplicity



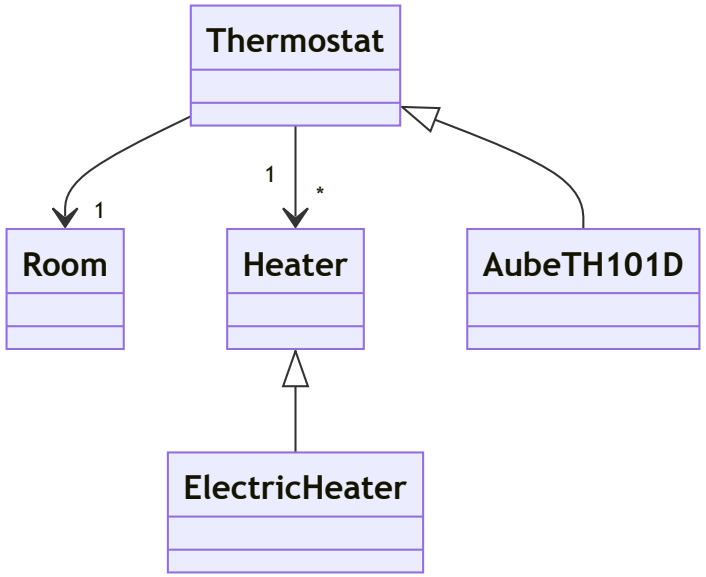
▶ Solution

Example: Bank



- Solution

Example: Home Heating System



- Solution

Example: Printer

```
public class Printer {
  private Job current;
  public void print();
  public boolean busy();
  public boolean on();
}

public class Job {}

public class Queue {
  private List<Job> jobs;
  private Printer printer;
  private Registry registry;
  public void newJob();
  public int length();
}

public class Registry {
  public Queue findQueue();
}
```

▶ Solution

UML Specification and Cheat Sheets

- OMG UML Specification: for UML tool developers
- Allen Holub's UML Quick Reference: good overview of basic modeling elements
- Martin Fowler: UML Distilled: a concise yet comprehensive book
- Martin Fowler: UML Articles
- Lou Franco's Class and Sequence Diagram Cheatsheet: 1-page to the point
- Wikipedia List of UML Tools