Advanced Programming

UML

Source: Learning UML 2.0, Russ Miles
http://math.uaa.alaska.edu/~afkjm/

System Design

- A typical software system may consist of large number of components
- We need to keep track of which components are needed, what their jobs are, and how they meet the customers’ requirements.
- We need to share our design with various stakeholders to ensure the pieces work together

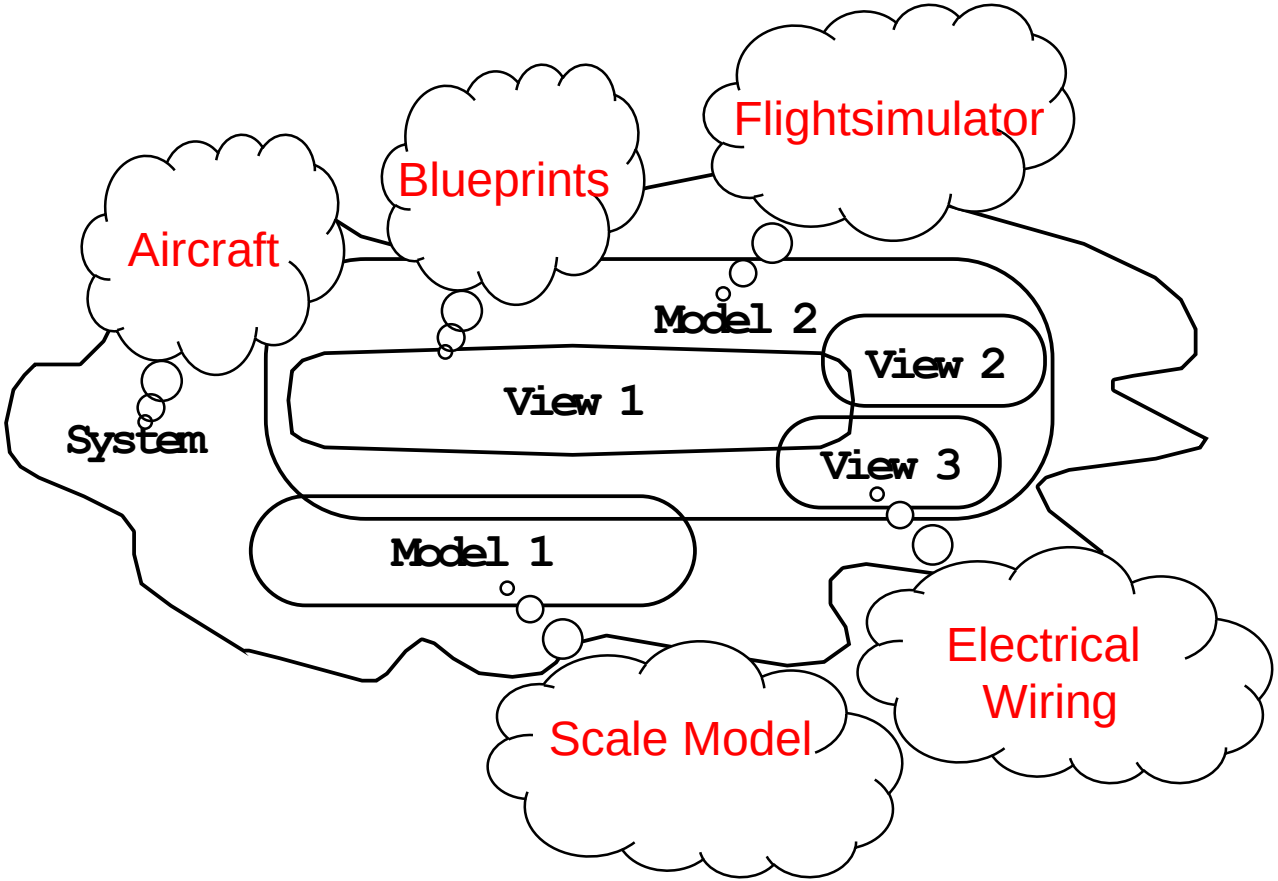
Systems, Models and Views

- A ***model*** is an abstraction describing a subset of a system
- A ***view*** depicts selected aspects of a model
- A ***notation*** is a set of graphical or textual rules for depicting views
- Views and models of a single system may overlap with each other

Examples:

- System: Aircraft
- Models: Flight simulator, scale model
- Views: All blueprints, electrical wiring, fuel system

Systems, Models and Views



Model

A abstraction or simplification of the real system

- Helps to manage complexity.
- Helps to focus on, capture, document, and communicate the important aspects of system’s design.
- Allows the design and viability of a system to be understood, evaluated, and criticized without digging through the actual system itself.

Why not use the source itself?

- All the details are included.
- Every notation has precise meaning for compiler.
- Code level comments can help describe the representation.
- Complete unambiguous representation of what the software will do.

What’s missing?

- Its implementation not model.
- Does not tell you how the software is to be used and by whom.
- No details of deployment.
- You have to understand code to read code: Only for developers
- No high level abstract views of your system for designers or customers.

Why not natural languages?

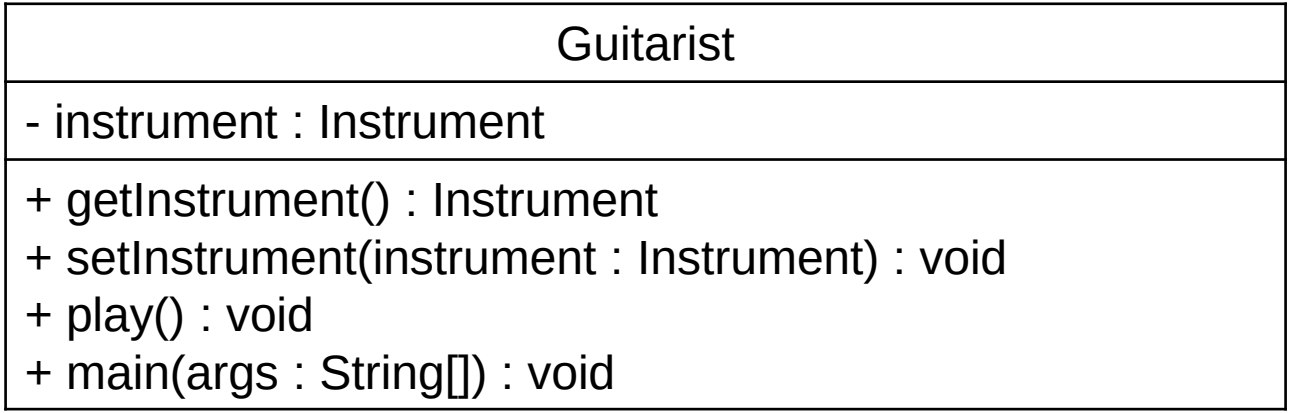
- No formal definitions of the constructs
- Ambiguous
- Verbose
- Imprecise with high chances to be misunderstood.

Unified Modeling Language (UML)

Standard modeling language for software and systems development

- A language with which the model can be described
- Capture high level system requirements
- Model parts of the system and their relationships
- Model how parts of the system work together.

Example


Guitarist
- instrument : Instrument
+ getInstrument() : Instrument
+ setInstrument(instrument : Instrument) : void
+ play() : void
+ main(args : String[]) : void

UML Advantages

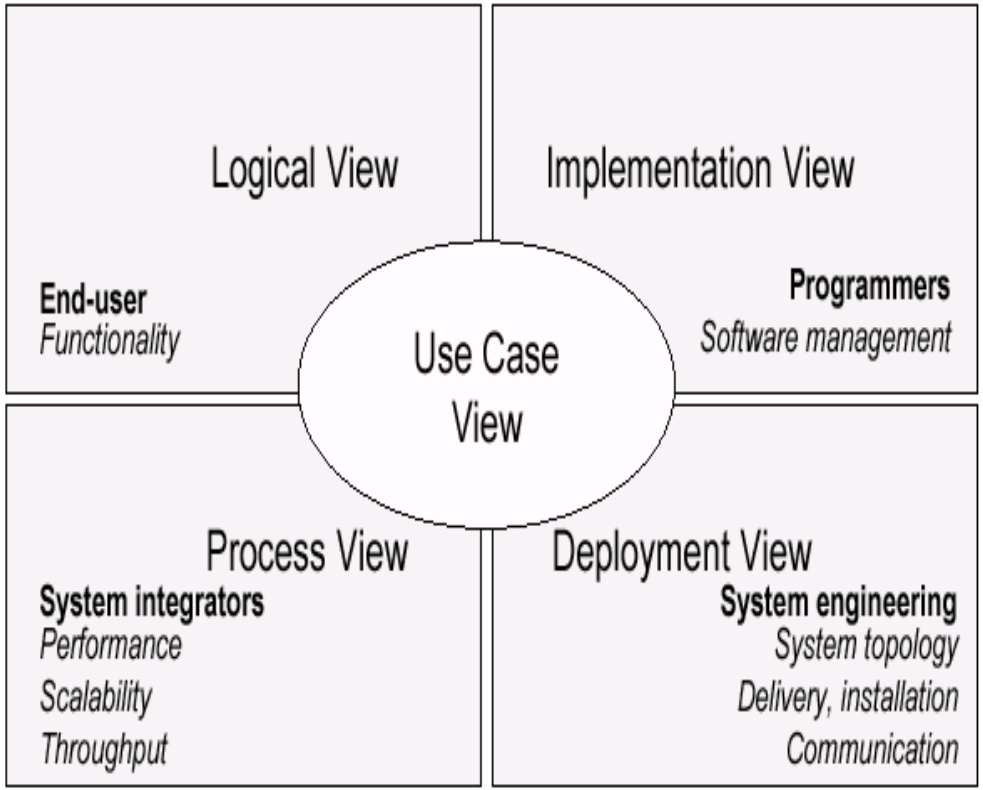
- **Formal language:** Each element of the language has a strongly defined meaning. No part of the system can be misunderstood.
- **Concise:** Simple and straightforward notation
- **Comprehensive:** Can describe all important aspects of a system.
- **Scaleable:** Can model large systems.
- **Open Standard:** Interoperable.

UML Models, Views, Diagrams

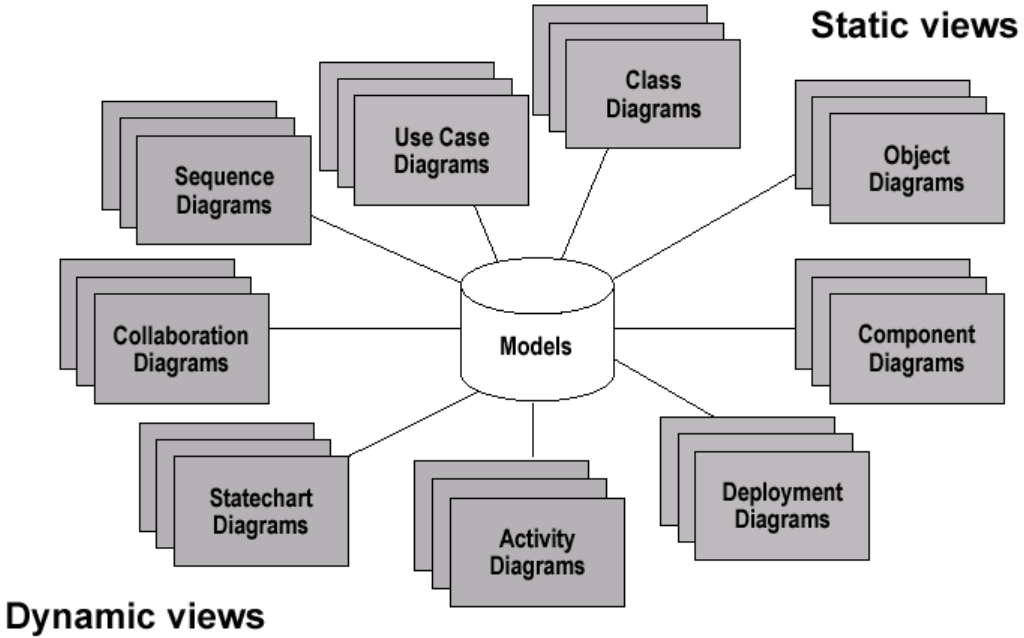
UML is a multi-diagrammatic language.

Each diagram is a view into a model

- Presented from the aspect of a particular stakeholder
- Provides a partial representation of the system
- Is semantically consistent with other views



Models, Views, Diagrams



UML: First Pass

- Not all systems require all the views.
- You can model 80% of most problems by using about 20 % UML
- We only cover less than 20% here!

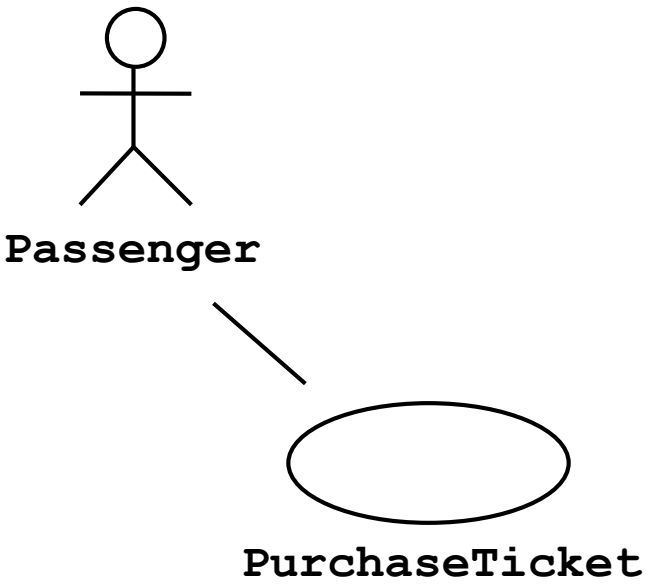
Basic Modeling Steps

- Use Cases
  - Capture requirements

- Domain Model
  - Capture process, key classes

- Design Model
  - Capture details and behaviors of use cases and domain objects.
  - Add classes that do the work and define the architecture.

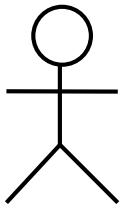
Use Case Diagrams



Used during requirements elicitation to represent external behavior

- ***Actors*** represent roles, that is, a type of user of the system
- ***Use cases*** represent a sequence of interaction for a type of functionality; summary of scenarios
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

Actors



Passenger

An actor models an external entity which communicates with the system:

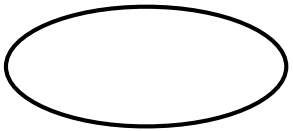
- User
- External system
- Physical environment

An actor has a unique name and an optional description.

Examples:

- Passenger: A person in the train
- GPS satellite: Provides the system with GPS coordinates

Use Case



PurchaseTicket

A use case represents a class of functionality provided by the system as an event flow.

A use case consists of:

- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

Use Case Diagram: Example

Name: `Purchase ticket`

Participating actor: `Passenger`

Entry condition:

`Passenger` standing in front of ticket distributor.

`Passenger` has sufficient money to purchase ticket.

Exit condition:

`Passenger` has ticket.

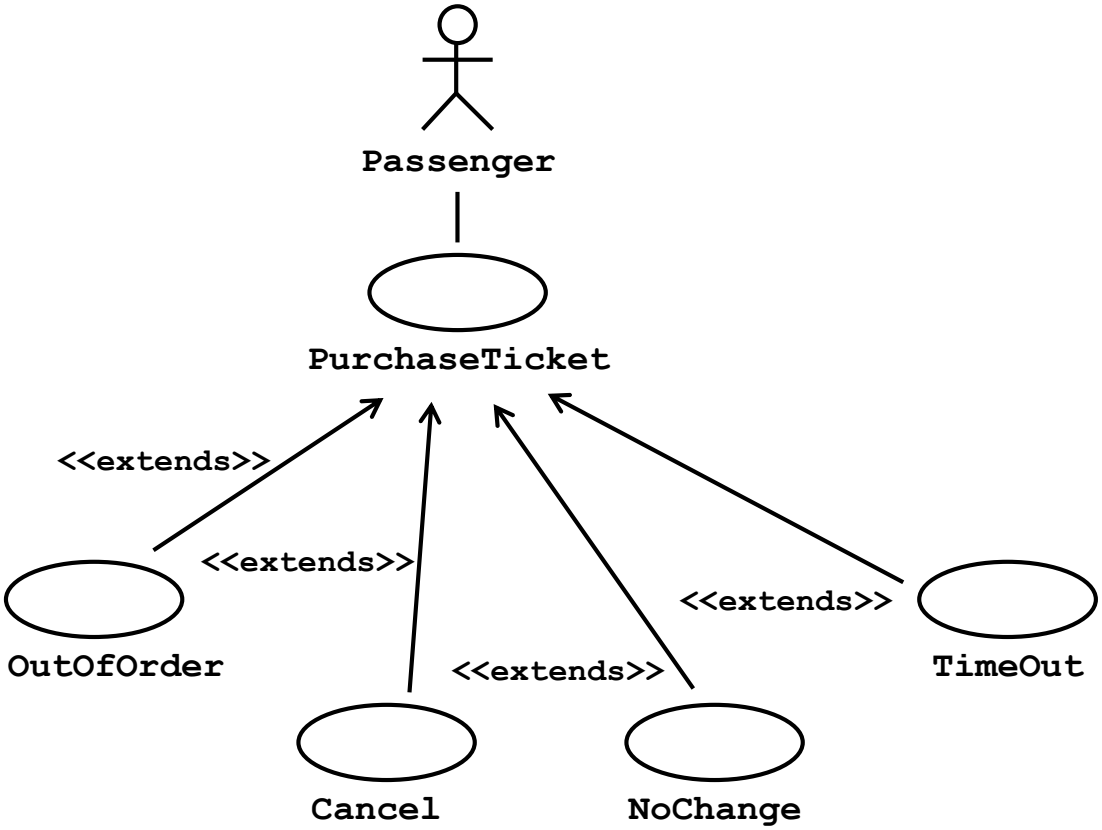
Event flow:

1. `Passenger` selects the number of zones to be traveled.
2. Distributor displays the amount due.
3. `Passenger` inserts money, of at least the amount due.
4. Distributor returns change.
5. Distributor issues ticket.

Anything missing?

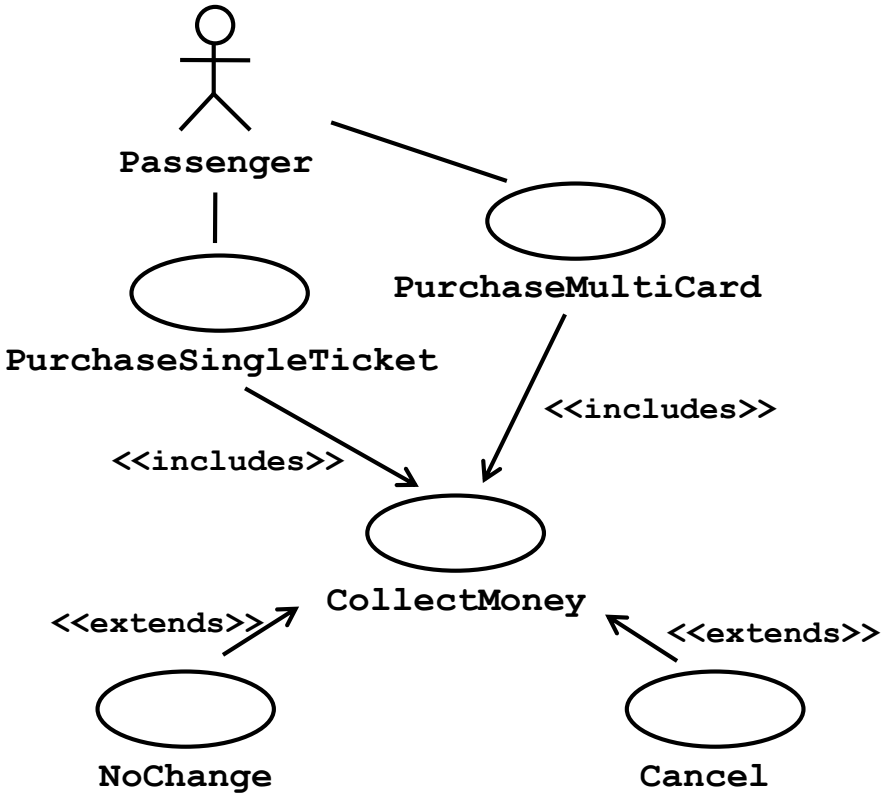
Exceptional cases!

The <<extends>> Relationship



- <<extends>> relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extends>> relationship is to the extended use case

The *<<includes>>* Relationship



- <<includes>> relationship represents behavior that is factored out of the use case.
- <<includes>> behavior is factored out for reuse, not because it is an exception.
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships).

Use Cases are useful to…

- Determining requirements
  - New use cases often generate new requirements as the system is analyzed and the design takes shape.
- Communicating with clients
  - Their notational simplicity makes use case diagrams a good way for developers to communicate with clients.
- Generating test cases
  - The collection of scenarios for a use case may suggest a suite of test cases for those scenarios.

Use Case Diagrams: Summary

- Use case diagrams represent external behavior
- Use case diagrams are useful as an index into the use cases
- Use case descriptions provide meat of model, not the use case diagrams.
- All use cases need to be described for the model to be useful.

Class Diagrams

- Gives an overview of a system by showing its classes and the relationships among them.
    - Class diagrams are static
    - they display what interacts but not what happens when they do interact

- Also shows attributes and operations of each class

- Good way to describe the overall architecture of system components

Class Diagram Perspectives

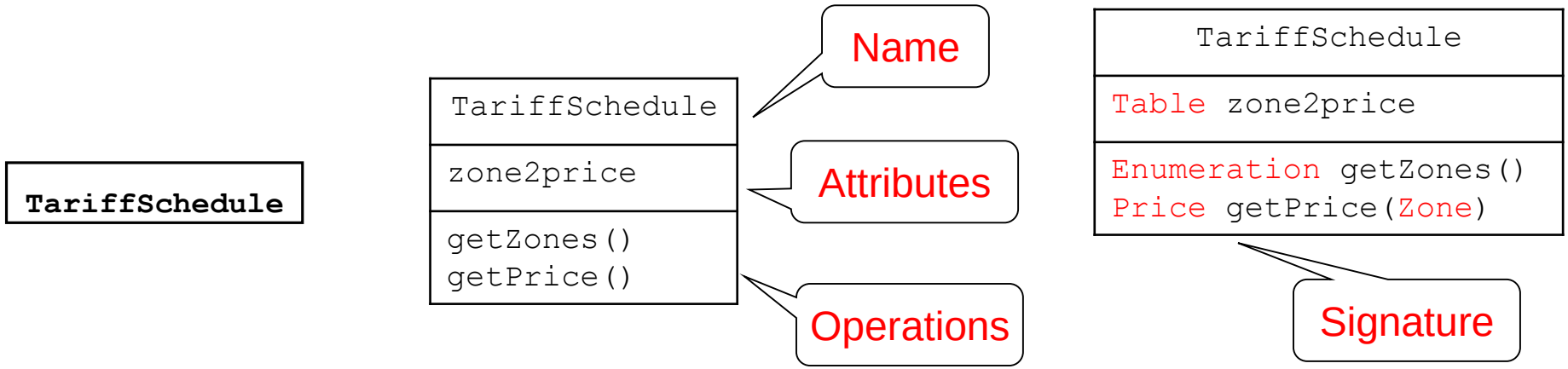
We draw Class Diagrams under three perspectives

- Conceptual
    - Software independent
    - Language independent

- Specification
    - Focus on the interfaces of the software

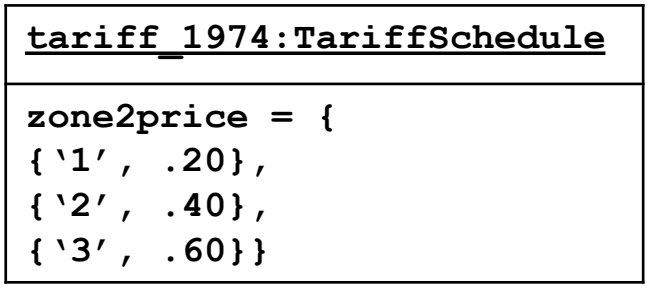
- Implementation
    - Focus on the implementation of the software

Classes – Not Just for Code



- A ***class*** represent a concept
- A class encapsulates state ***(attributes)*** and behavior ***(operations).***
- Each attribute has a ***type***.
- Each operation has a ***signature***.
- The class name is the only mandatory information.

Instances



An ***instance*** represents a phenomenon.

The name of an instance is underlined and can contain the class of the instance.

The attributes are represented with their ***values***.

UML Class Notation

A class is a rectangle divided into three parts
- Class name
- Class attributes (i.e. data members, variables)
- Class operations (i.e. methods)

Modifiers
- Private: -
- Public: +
- Protected: #
- Package: ~
- Static: Underlined

Abstract class: Name in italics

Employee
-Name: string +ID: long #Salary: double
+getName: string +setName() -calcInternalStuff(in x : byte, in y : decimal)

UML Class Notation

Lines or arrows between classes indicate relationships

- Association
  - A relationship between instances of two classes, where one class must know about the other to do its work, e.g. client communicates to server
  - indicated by a straight line or sometime arrow
- Aggregation
  - An association where one class belongs to a collection, e.g. instructor part of Faculty
  - Indicated by an empty diamond on the side of the collection

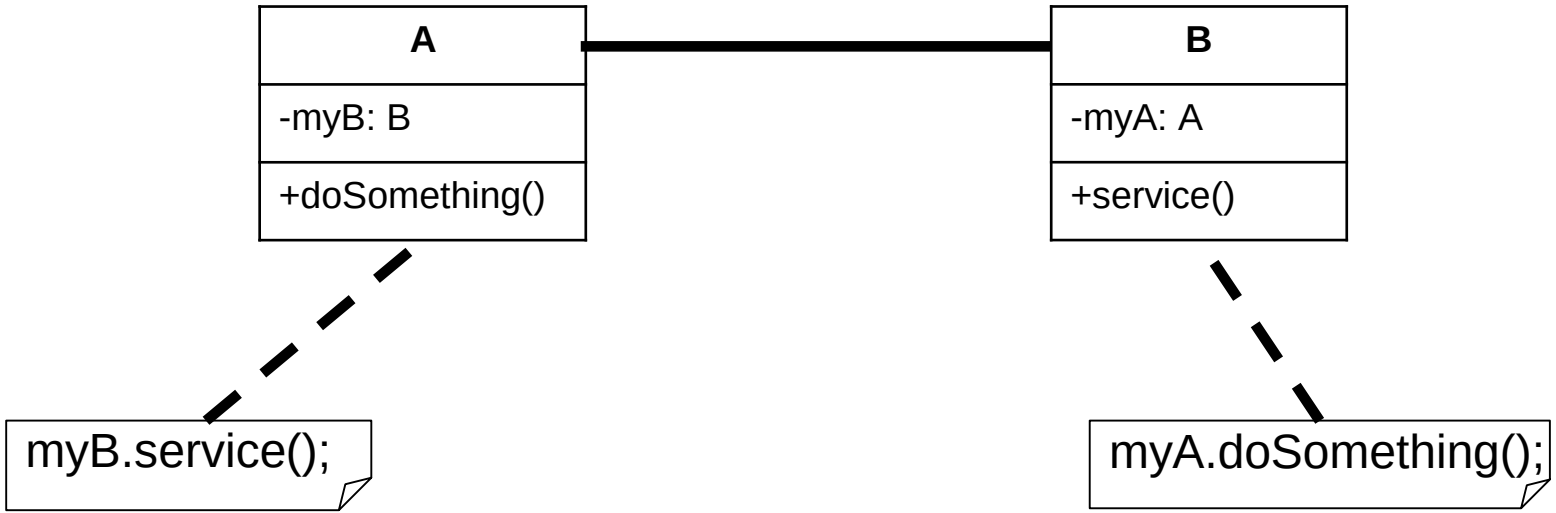
UML Class Notation

- Composition
  - Strong form of Aggregation
  - Lifetime control; components cannot exist without the aggregate
  - Indicated by a solid diamond on the side of the collection

- Inheritance
  - An inheritance link indicating one class a superclass relationship, e.g. bird is part of mammal
  - Indicated by triangle pointing to superclass

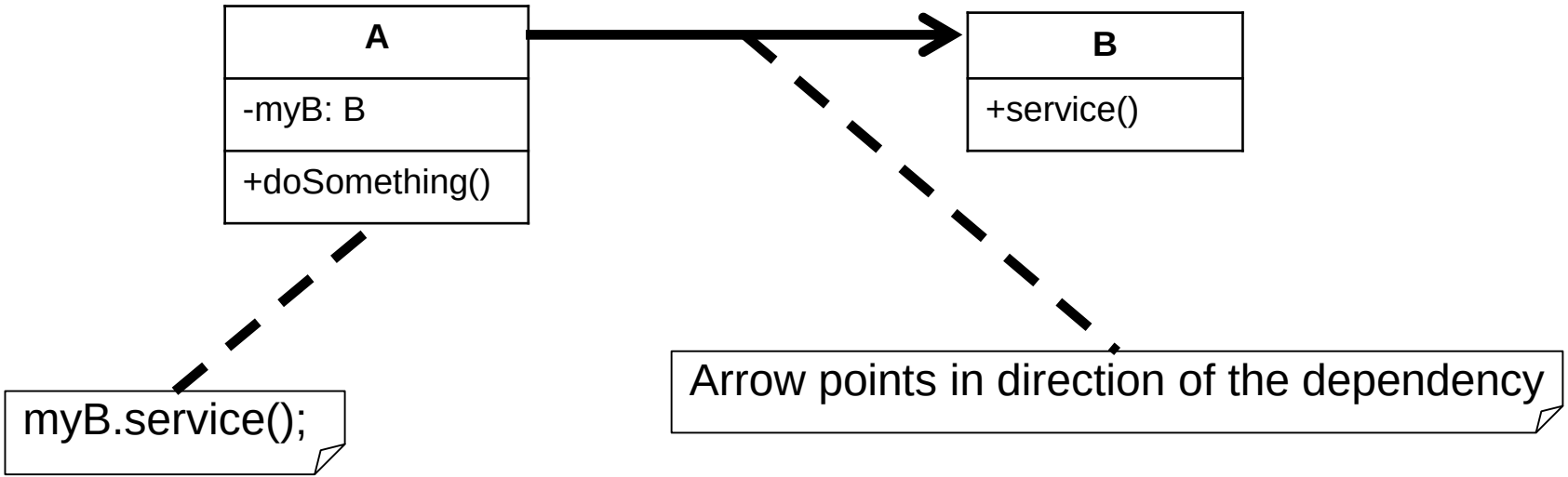
Binary Association

Both entities “Know About” each other



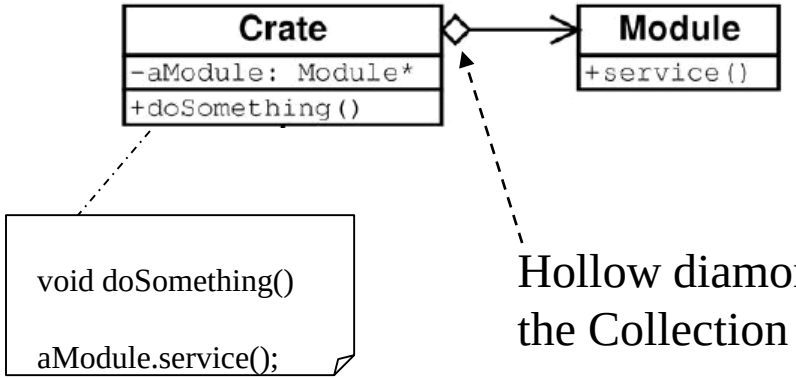
Unary Association

A knows about B, but B knows nothing about A



Aggregation

Aggregation is an association with a “collection-member” relationship



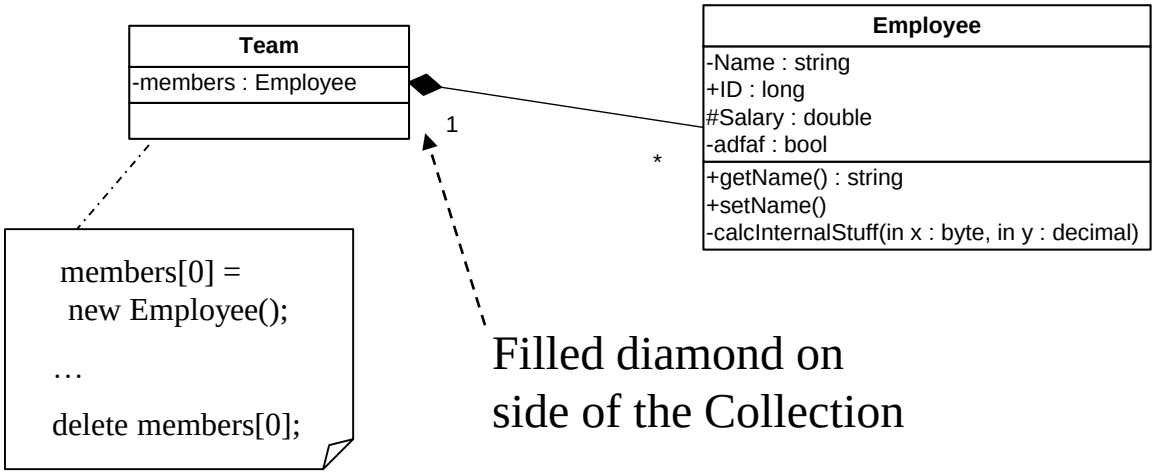
Hollow diamond on
the Collection side

No sole ownership implied

Composition

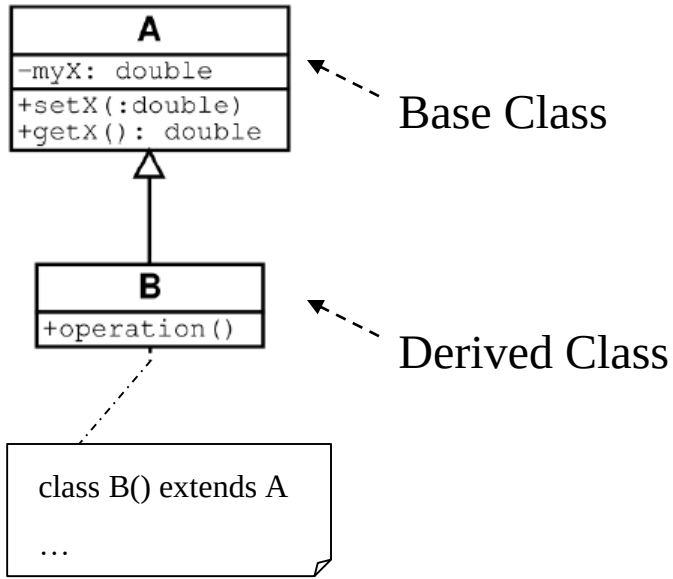
Composition is Aggregation with:

- Lifetime Control (owner controls construction, destruction)
- Part object may belong to only one whole object



Inheritance

Standard concept of inheritance



UML Multiplicities

Links on associations to specify more details about the relationship

Multiplicities	**Meaning**
0..1	zero or one instance. The notation ***n . . M*** indicates ***n*** to ***m*** instances.
0..* *or* *****	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance