CENG 4501
Software Design Patterns

Fall 2024

Özgür Kılıç
Office: E1-03
email: ozgur.kilic10@gmail.com

Official Course Web Page:
dys

Agenda

- Course Overview

- Introduction to Design Patterns

Course Learning Objectives

- To understand and describe the design paterns and design principles

- To apply the appropriate design pattern in the design of a software

- To implement a design pattern using an object-oriented programming language

Course overview

- **Week 1** : Introduction to Design Patterns
- **Week 2** : The Strategy Pattern
- **Week 3** : The Observer Pattern
- **Week 4** : Decorator Pattern
- **Week 5** : Factory Pattern & Singleton Pattern
- **Week 6** : Command Pattern
- **Week 7** : Adapter Pattern & Facade Pattern
- **Week 8 :** Midterm Exam
- **Week 9** : Template Method Pattern
- **Week 10** : Iterator Pattern
- **Week 11** : Composite Pattern
- **Week 12** : State Pattern
- **Week 13** : Proxy Pattern
- **Week 14** : Compound Patterns

Grading

• Midterm	40%
• Quiz	10%
• Final exam	50%

Text Book


O'REILLY
10th Anniversary
Updated for Java 8
Head First
Design Patterns
A Brain-Friendly Guide
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates

Text Book

- "Head First Design Patterns"
  - Eric Freeman & Elisabeth Freeman
- Book describes some of the Gang of Four design patterns
- Easier to read
- Examples are fun, but not necessarily "real world"

Introduction to Design Patterns

Questions

- Can you name a few of design patterns?
- Any idea about design patterns?
- What are the benefits of design patterns?

Designing a OO Software

- Designing OO Software is hard
  - To find relevant objects
  - Factor them into classes
  - Define class Interfaces
  - Define inheritance hierarchies
  - Establish key relationships among them
- OO Design must be
  - Specific to the problem at hand
  - General to address future problems and requirements

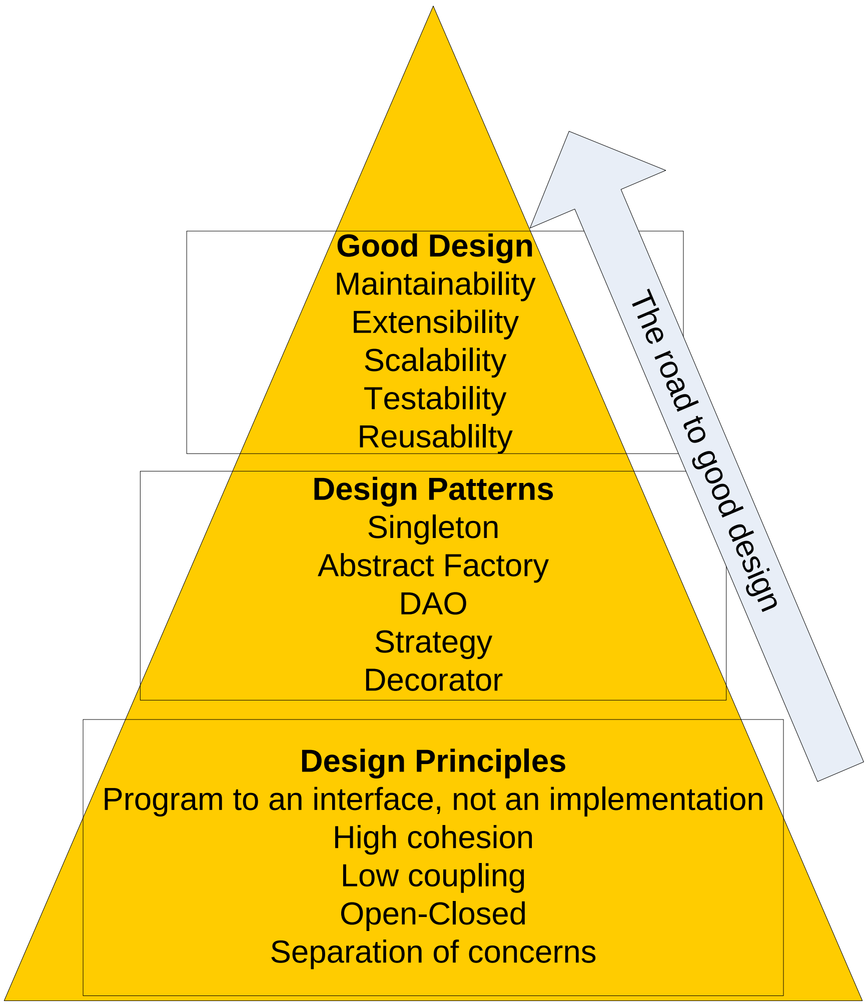
OO Programming Concepts

- Class
- Object
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Decomposition

Why Study Design Patterns?

- Design Objectives
  - Good Design (the “ilities”)
    - High readability and maintainability
    - High extensibility
    - High scalability
    - High testability
    - High reusability

Why Study Design Patterns?



https://www.slideshare.net/CommitSoftwareAlbania/design-patterns-understand-the-patterns-and-design-your-own

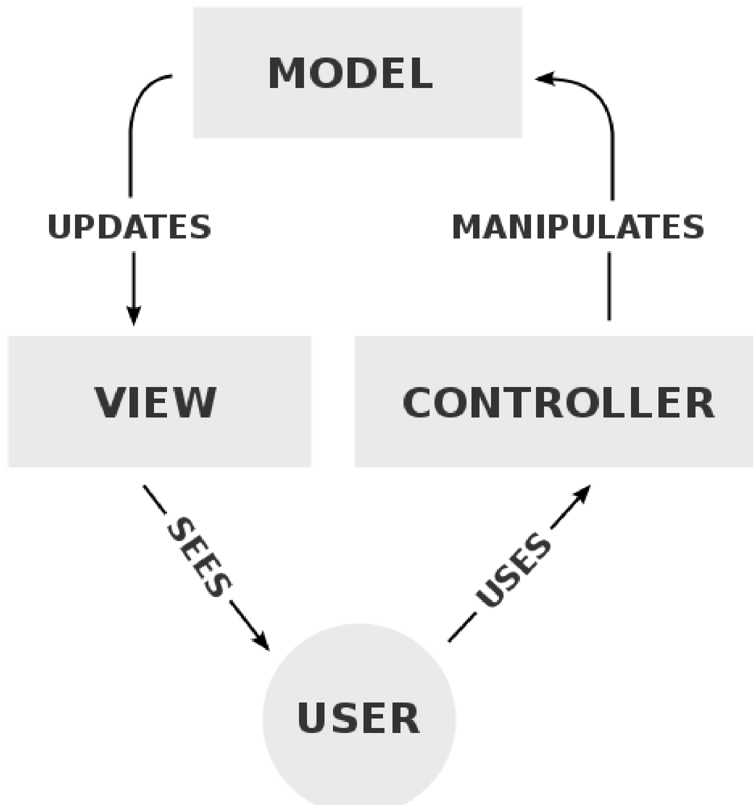
Why Study Design Patterns?

- Many programmers have encountered similar **problems** before, and have used similar '**solutions**' to remedy them. If you encounter these problems, why recreate a solution when you can use an already proven answer?

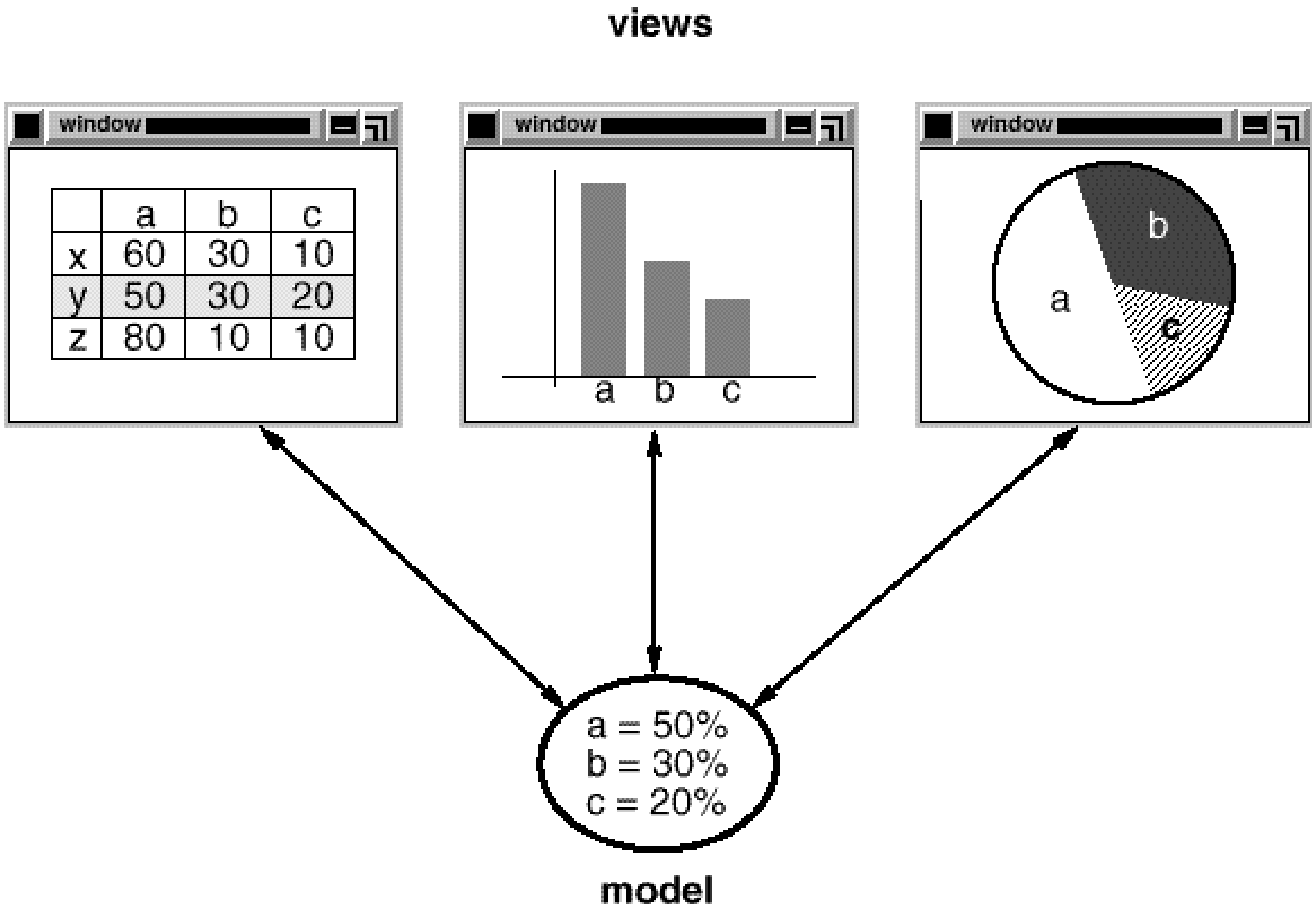
- Provides a vocabulary that can be used amongst software developers.

- Help you think about how to solve a software problem.

Model View Controller



MVC Model


views
window
window
window
a
b
c
x
60
30
10
y
50
30
20
z
80
10
10
a
b
c
b
a
c
a = 50%
b = 30%
c = 20%
model

What is a Design Pattern

- Wikipedia definition
  - "a design pattern is a general repeatable solution to a commonly occurring problem in software design"
- Not a finished design that can be transformed directly into code.
- "A description of communicating objects and classes that are customized to solve a general design problem in a particular context." GoF
- Typically show relationships between classes or objects, without specifying the final application classes or objects that are involved.

History of Design Patterns

- Christopher Alexander (Civil Engineer) in 1977 wrote
  - " Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice ."
  - It was initially applied for architecture for buildings and towns, but not computer programming.

History of Design Patterns

- In 1995, the principles that Alexander established were applied to software design and architecture. The result was the book:
  - “ Design Patterns: Elements of Reusable Object-Oriented Software” by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
  - The authors are often referred to as the Gang of Four (GoF).

The Gang of Four

- Defines a Catalog of different design patterns
- Three different types
  – **Creational Patterns**: "creating objects in a manner suitable for the situation"
  – **Structural Patterns**: "ease the design by identifying a simple way to realize relationships between entities"
  – **Behavioral Patterns**: "common communication patterns between objects"

The Gang of Four: Pattern Catalog

- Creational
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Structural
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy
- Behavioral
  - ***Chain of Responsibility***
  - ***Command***
  - ***Interpreter***
  - ***Iterator***
  - ***Mediator***
  - ***Memento***
  - ***Observer***
  - ***State***
  - ***Strategy***
  - ***Template Method***
  - ***Visitor***

Catalog of Design Patterns

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (83)	Adapter (108)	Interpreter (191) Template Method (254)
	Object	Abstract Factory (68) Builder (75) Prototype (91) Singleton (99)	Adapter (108) Bridge (118) Composite (126) Decorator (135) Facade (143) Proxy (161)	Chain of Responsibility (173) Command (182) Iterator (201) Mediator (213) Memento (221) Flyweight (151) Observer (229) State (238) Strategy (246) Visitor (259)

Organizing the Catalog

- With respect to “Purpose”
  - Creational patterns concern the process of object creation
  - Structural patterns deal with the composition of classes or objects
  - Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility
- With respect to “Scope”
  - Class patterns deal with relationships between classes and their subclasses. (with Inheritance, they are static)
  - Object patterns deal with object relationships (dynamic)

How Design Patterns Solve Problems

- Finding Appropriate Objects
- Determine Object Granularity
- Specify Object Interfaces
- Specifying Object Implementations
  - Programming to an Interface, not an implementation
- Encourage Reusability
  - Inheritance versus Compositon
  - Delegation
- Design for Change

Finding Appropriate Objects

- Hard part about OO design is decomposing a system into objects.
- Designer should consider many factors including encapsulation, granularity, dependency, flexibility, reusability, etc..
- Objects that represent a process or algorithm don't occur in nature
- Examples
  - Strategy Pattern describes how to implement interchangeable families of algorithms.
  - State Pattern represents each state of an entity as an object.

Determine Object Granularity

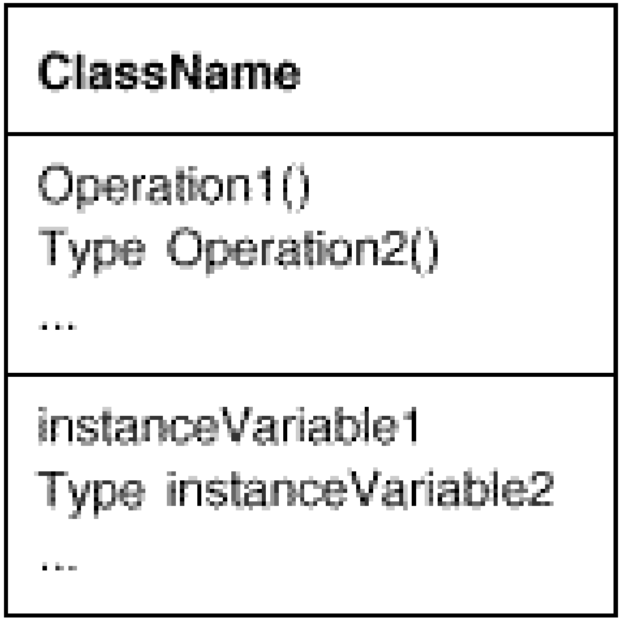
- Objects can vary tremendously in size and number
- They can represent everything down to hardware or all the way up to entire applications.
- Examples
  - Façade pattern describes how to represent complete systems as an object
  - Flyweight pattern describes how to support huge numbers of objects at fines granularities

Specifiying Object Interfaces

- Interfaces are fundamental in object-oriented systems
- Interface says nothing about its implementation
- Design Patterns help you define interfaces by identifying key elements and kind of data that get sent across an interface
- Design patterns also specify relationships between interfaces
- Examples: Memento, Strategy, etc..

Specifying Object Implementation

- An object's implementation is defined by its *class*.



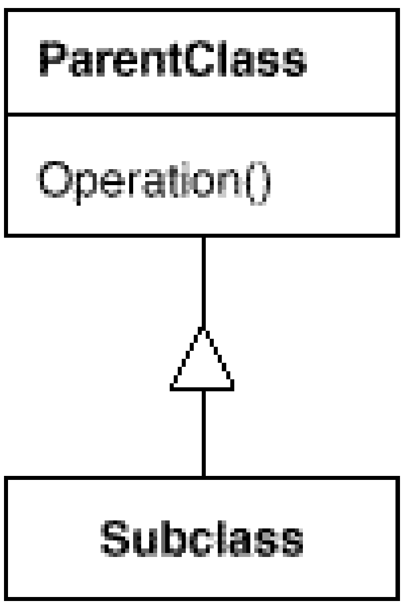
Specifying Object Implementations

- Objects are created by *instantiating* a class.

- The object is said to be the *instance* of the class.

Specifying Object Implementations

- New classes can be defined in terms of existing classes using *class inheritance*.
- When a *sub-class* inherits from a *parent-class*, it includes all the data and operations.

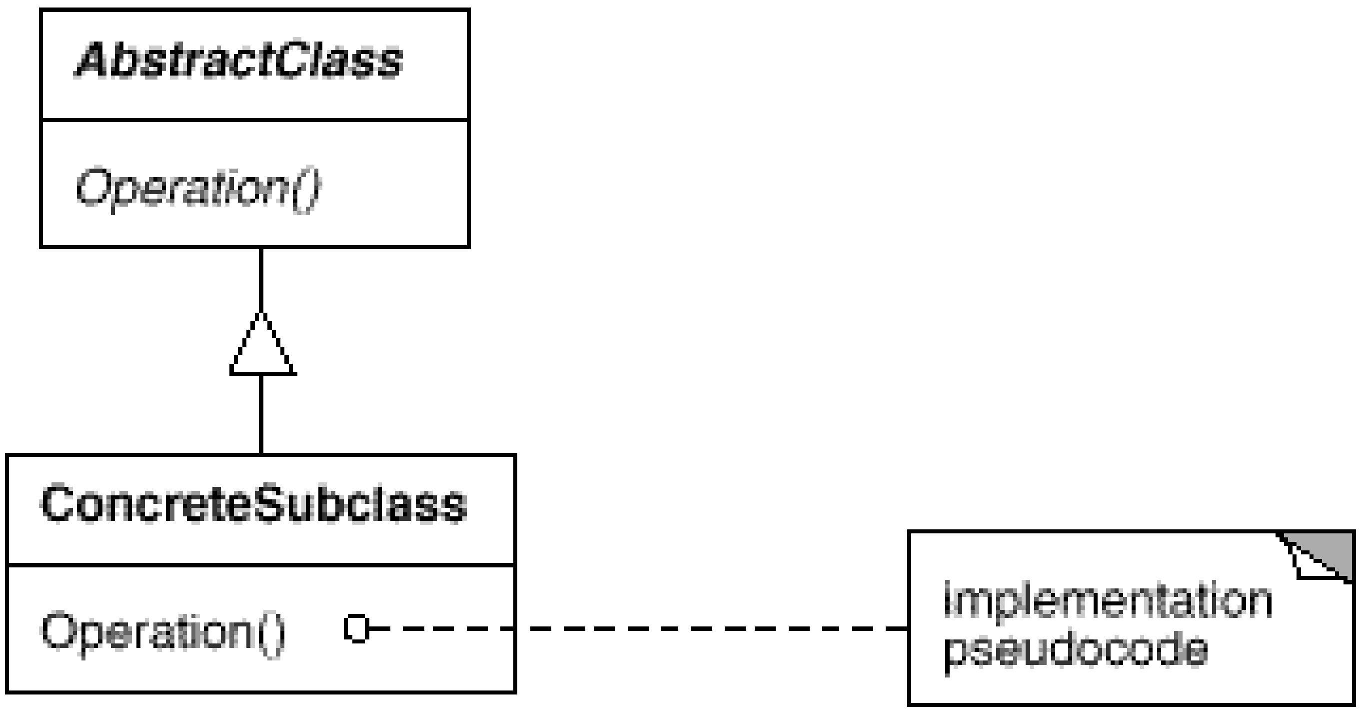


Specifying Object Implementations

- An *abstract class* is one whose main purpose is to define a common interface for its subclasses.
- An abstract class can not be instantiated.
- The operations that an abstract class declares but does not implement is called *abstract operations*.
- Classes that aren’t abstract are called *Concrete Classes*.

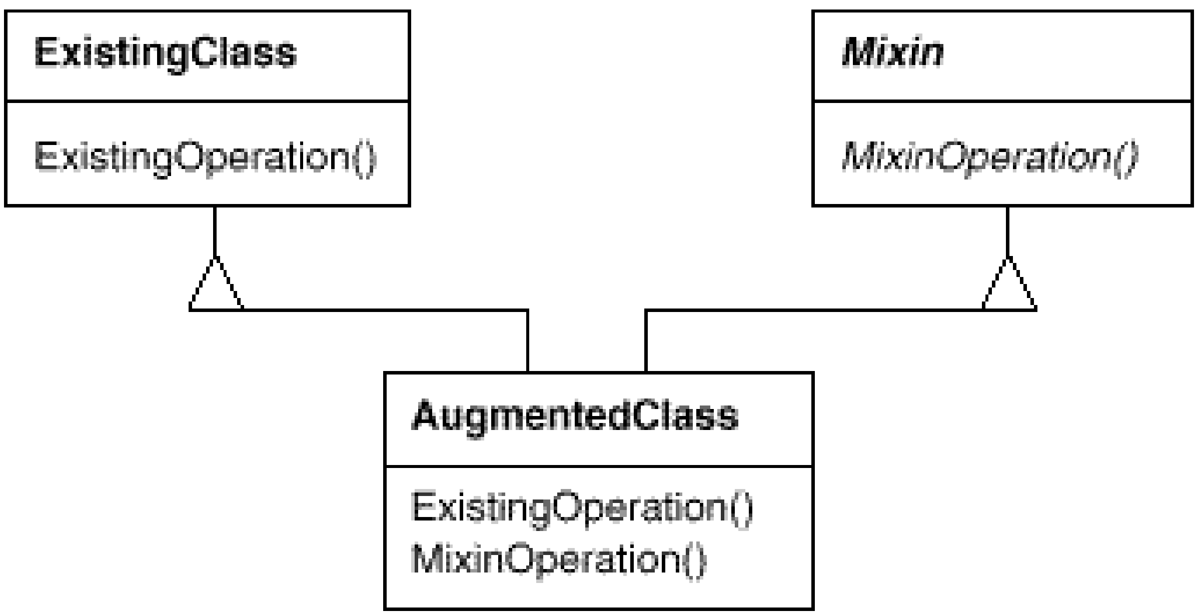
Specifying Object Implementations

- Subclasses can *override* an operation defined by its parent classes.



Specifying Object Implementations

- A *mixin class* is a class that is intended to provide an optional interface or functionality to other classes. Requires multiple inheritance.



Class vs interface Inheritance

- An object’s class defines how the object is implemented.

- An object’s type refers to its interface – the set of requests to which it can respond.

Programming to an Interface not to an Implementation

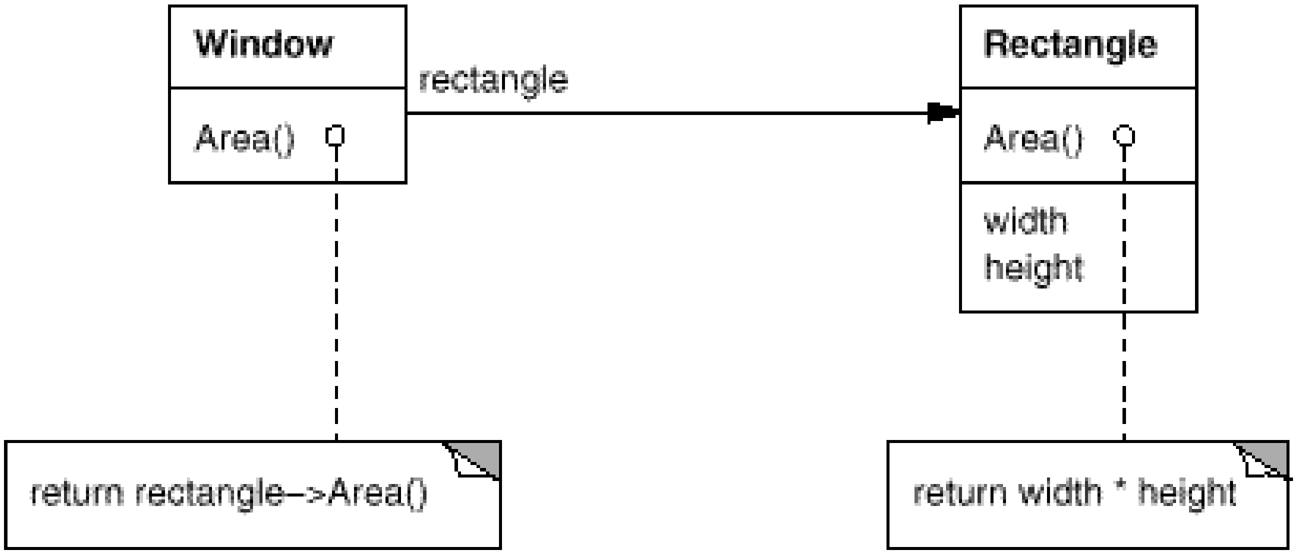
- Two benefits
  - Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.
  - Clients remain unaware of the classes that implement these objects. Clients only know about the abstract classes defining the interface.

Reuse Mechanisms

- Class Inheritance vs. Object Composition
  - Inheritance = white-box reuse, Composition = black-box reuse
  - Class inheritance
      - Is defined at compile-time,
      - Is straightforward to use,
      - Can modify the implementation.
    - But
      - You can't change it at run-time.
      - Breaks-up the encapsulation.
      - Depends to the parent class' implementation.
  - Object composition
      - Is defined at run-time
      - Fewer implementation dependencies
      - Each class will focus on one work.
      - Less classes
- Favor object composition over class inheritance

Reuse Mechanisms

- Delegation
  - *Delegation* is a way of making composition as powerful for reuse as inheritance.
  - In delegation, two objects are involved in handling a request, a receiving object delegates the operations to its *delegate*.



Designing for Change

- Find what varies and encapsulate it
- Allows adding alternative variations later
- Strategy Pattern allows adding new algorithms without effecting other parts of the code

Reality

- Problems with design early on
  - It is sometimes very hard to see a design pattern
  - Not all requirements are known
  - A design that is appliable early on becomes obsolote
  - Analysis paralysis : the state of over-analyzing (or over-thinking) a situation so that a decision or action is never taken
- Due to these realities, refactoring is inevitable

Unit Testing

- If you create unit tests early in the development cycle, then it will be easier to refactor later on when more requirements are known.
  - As a developer, you will have more confidence to make good design adjustments.
  - Good design adjustments may lead to better maintainability of the code!
- What happens if you do not have Unit Tests early on? These statements may be heard:
  - “ I am afraid to breaking something.”
  - “ I know the right thing to do….but I am not going to do it because the system may become unstable.”
  - You may incur “Technical Debt” if you do not refactor well

Unit Testing

- Unit Testing leads to easier Refactoring
- With easier refactoring, you can take risk of applying design patterns, even it means changing a lot of code
- Applying design patterns can decrease Technical debt and improve the maintainability

Unit Testing

- Make unit testing part of the project culture
- When creating a schedule, include unit testing in your estimates
- Create your unit tests before you write the code (Test Driven Programming)

Common Pitfall

- "I've just learned about Design Pattern XYZ. Let's use it!
- Reality: If you are going to use a Design Pattern, you should have a reason to do so.
- The software requirements shoul really drive why you are going to use a Design Pattern.

Summary

- Design Pattern
- Model View Controller Architecure
- Catalog of Design Patterns
- How Design Patterns Solve Problems
- Unit Testing

UML Basics

What is UML?

- An industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling.
- The UML uses mostly graphical notations to express the OO analysis and design of software projects.
- Simplifies the complex process of software design

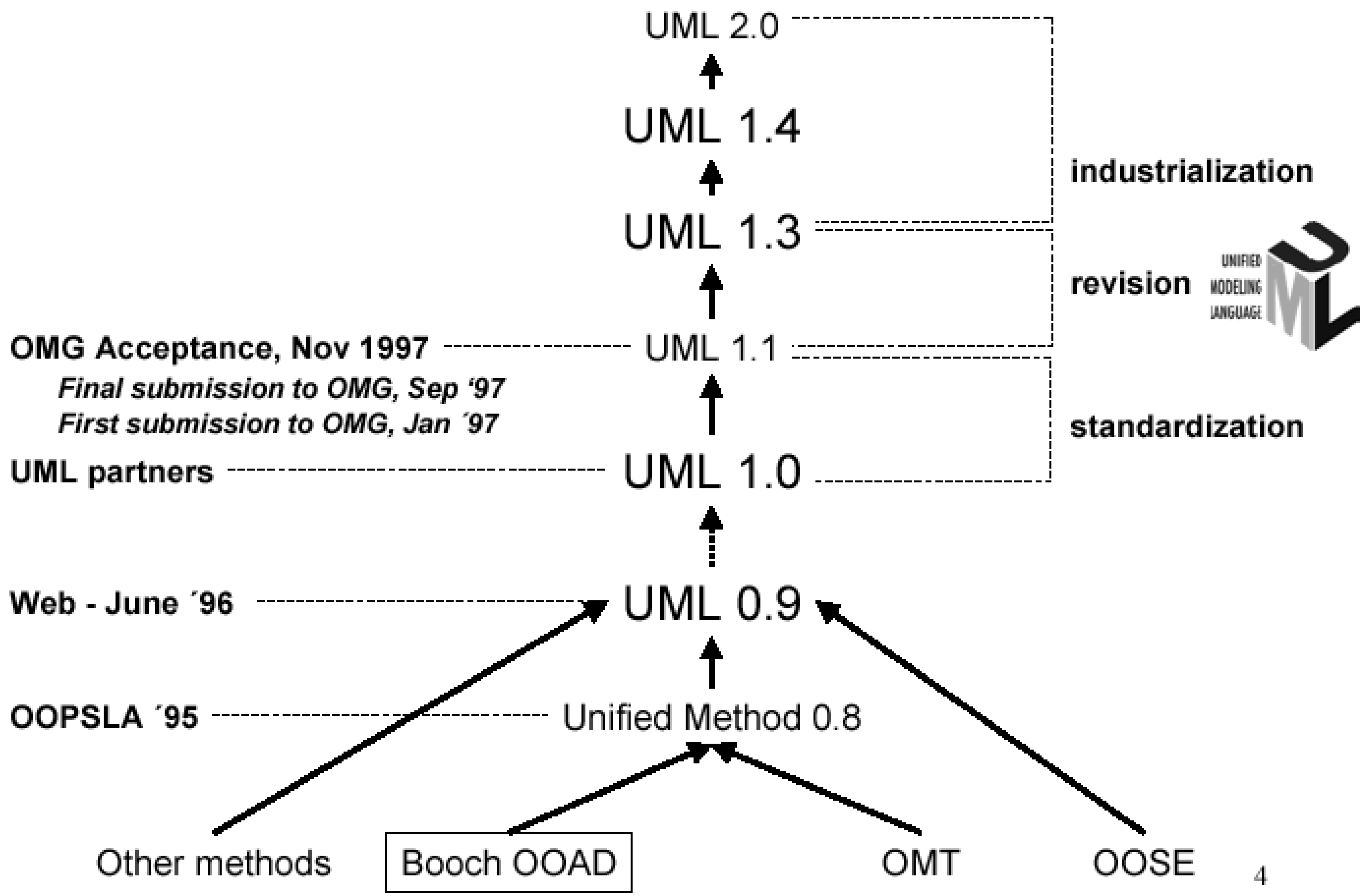
Why use UML

- A diagram/picture = thousands words
- Uses graphical notation to communicate more clearly than natural language (imprecise) and code(too detailed).
- Makes it easier for programmers and software architects to communicate.
- Helps acquire an overall view of a system.
- UML is not dependent on any one language or technology.

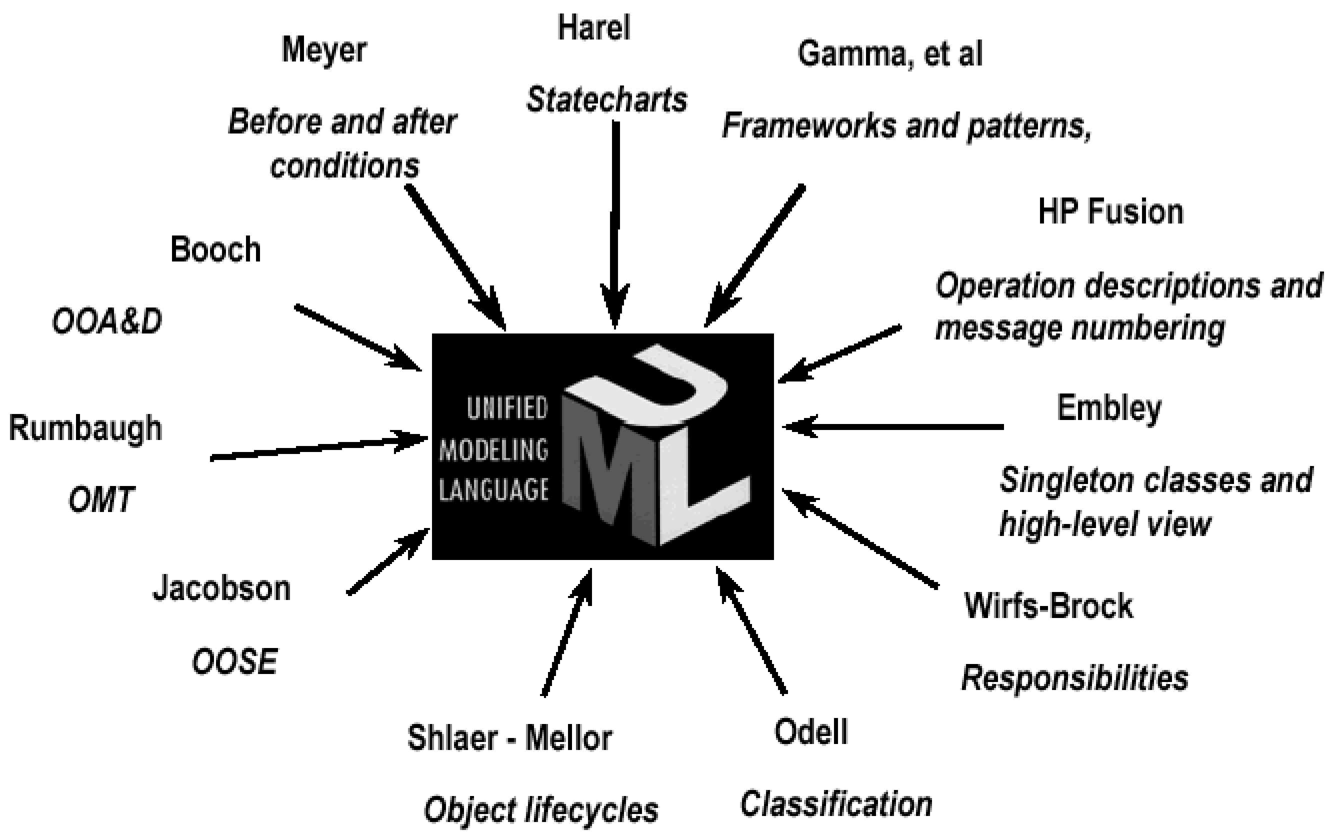
Brief History

- Many methodologies in early 90’s
  - Booch, Jacobson, Yourden, Rumbaugh
- Booch, Jacobson merged methods 1994
- Rumbaugh joined 1995
- 1997 UML 1.1 from OMG includes input from others, e.g. Yourden
- UML v2.5 current version

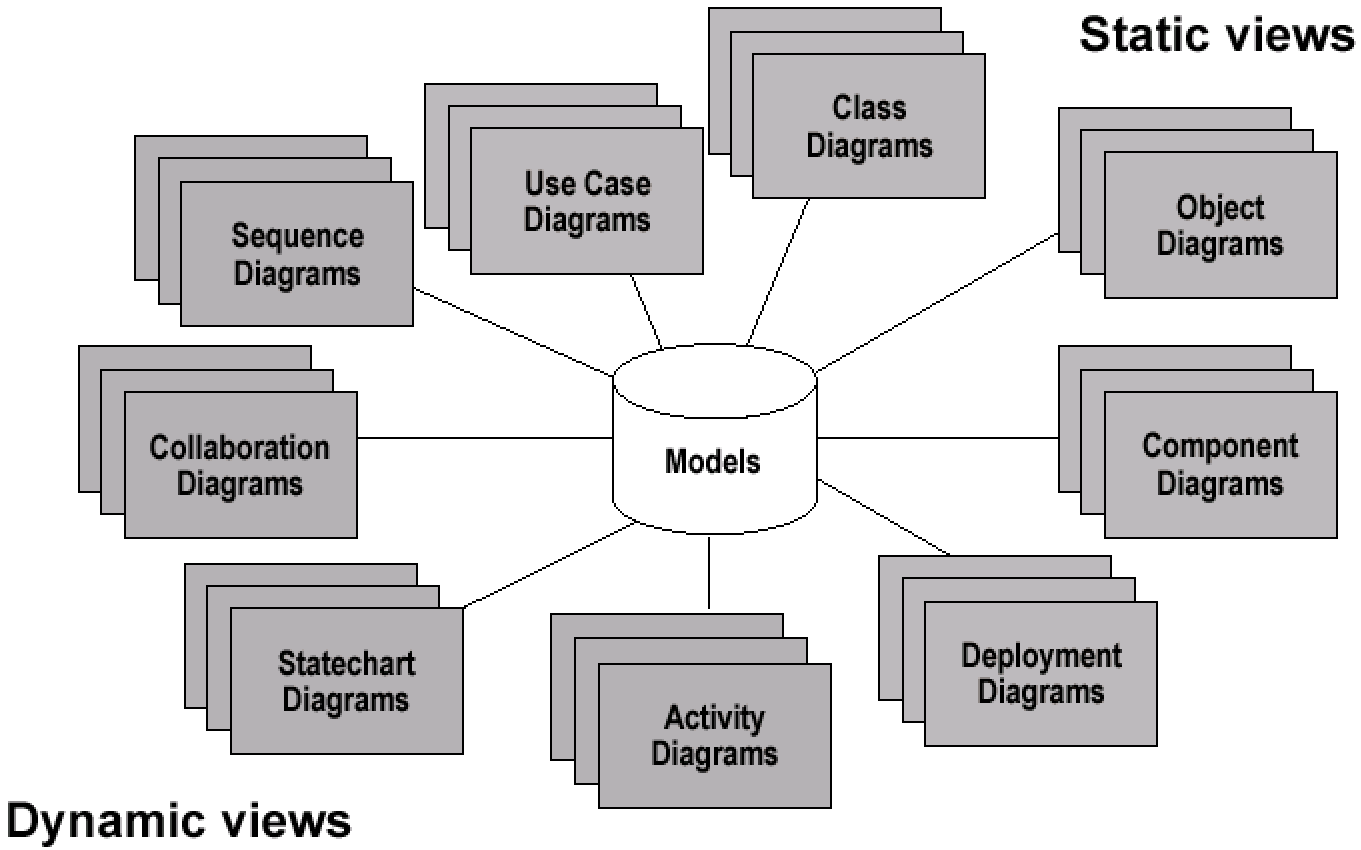
History of UML



Contributions to UML



Models, Views, Diagrams



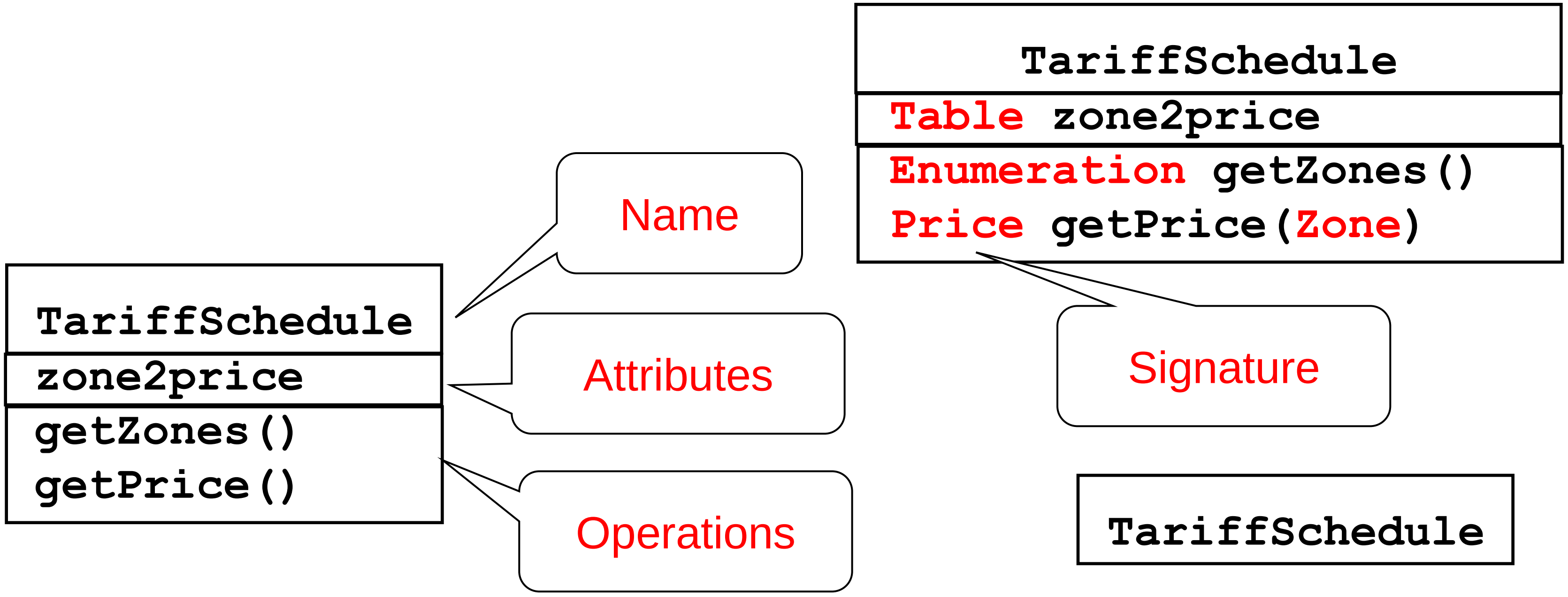
UML Baseline

- Use Case Diagrams
- Class Diagrams
- Package Diagrams
- Interaction Diagrams
  - Sequence
  - Collaboration
- Activity Diagrams
- State Transition Diagrams
- Deployment Diagrams

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

Classes



- A ***class*** represent a concept
- A class encapsulates state ***(attributes)*** and behavior ***(operations).***
- Each attribute has a ***type***.
- Each operation has a ***signature***.

Instances

```
tarif_1974:TariffSchedule
zone2price = {
{'1', .20},
{'2', .40},
{'3', .60}}
```

- An ***instance*** represents a object.
- 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: #
  - Static: Underlined (i.e. shared among all members of the class)
- Abstract class: Name in italics

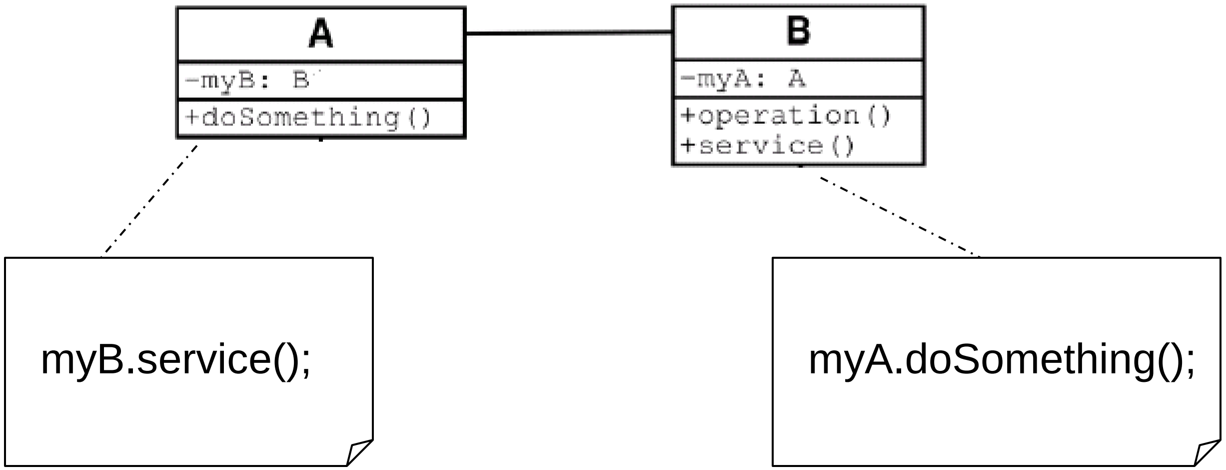
Employee
-Name : string +ID : long #Salary : double
+getName() : string +setName() -calcInternalStuff(in x : byte, in y : decimal)

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 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
  - 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

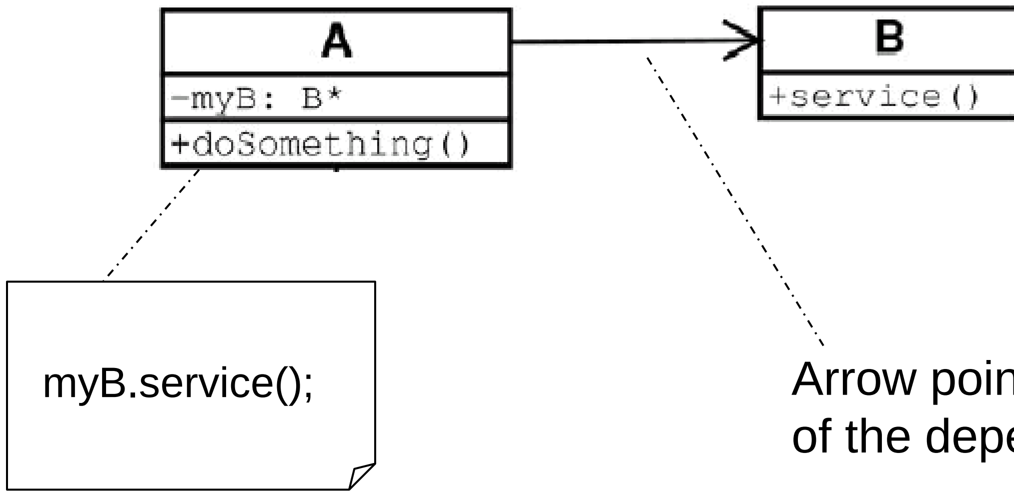
Binary Association: Both entities “Know About” each other



Optionally, may create an Associate Class

Unary Association

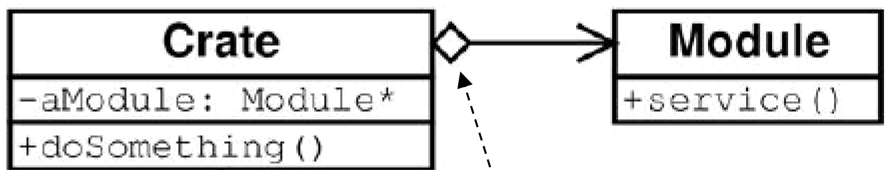
A knows about B, but B knows nothing about A



Arrow points in direction
of the dependency

Aggregation

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



void doSomething()

aModule.service();

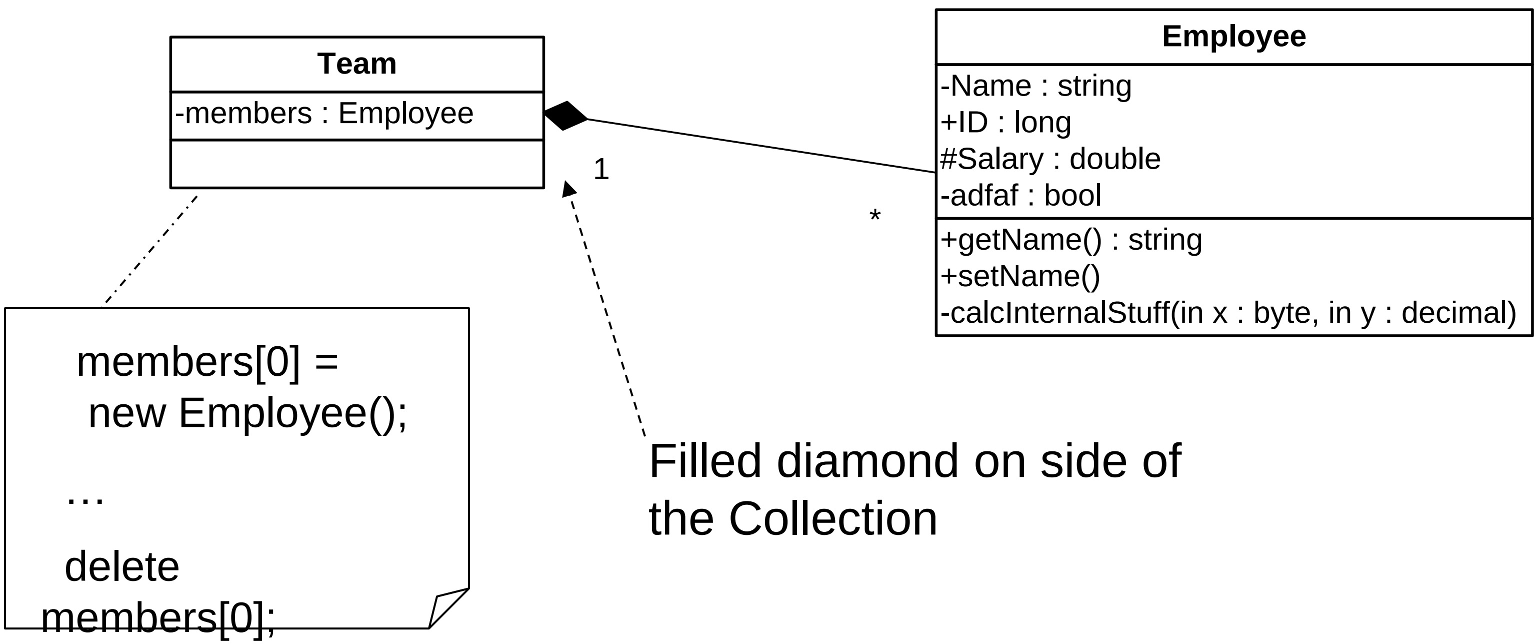
Hollow diamond on
the Collection side

No sole ownership implied

Composition

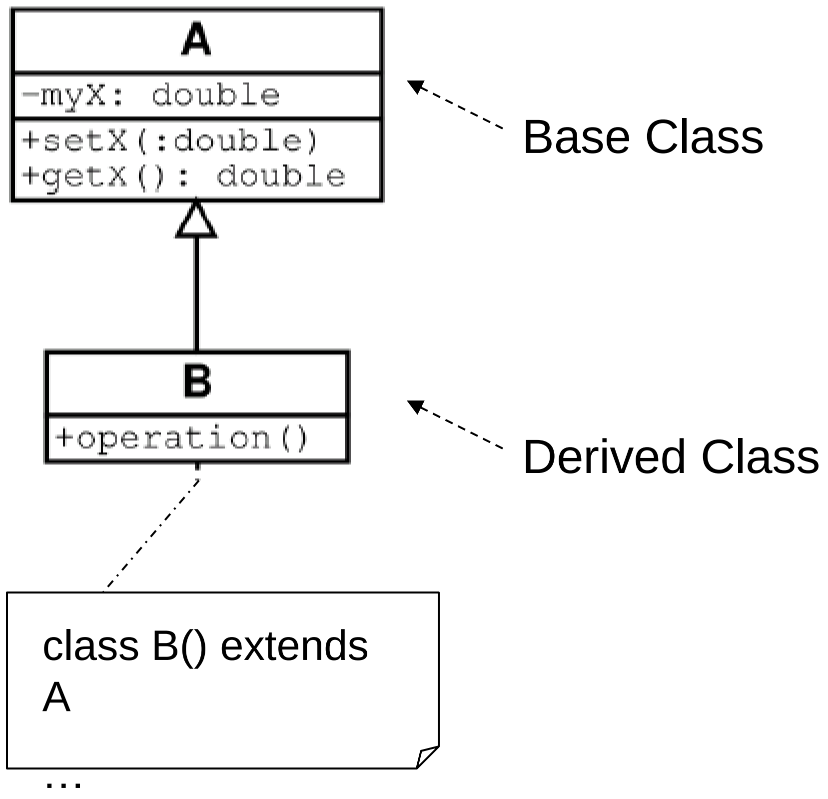
Composition is Aggregation with:

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



Inheritance

Standard concept of inheritance

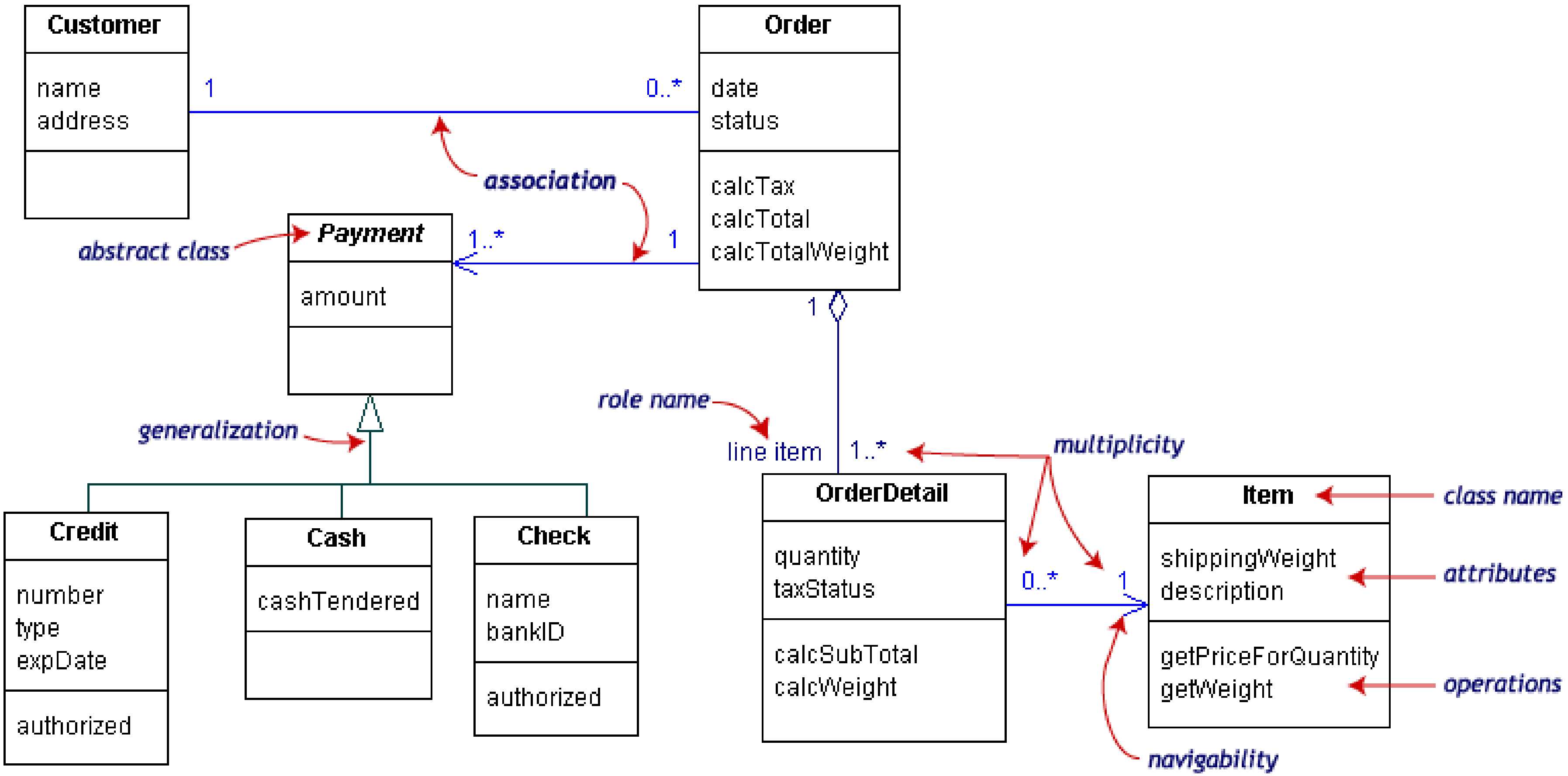


UML Multiplicities

Links on associations to specify more details about the relationship

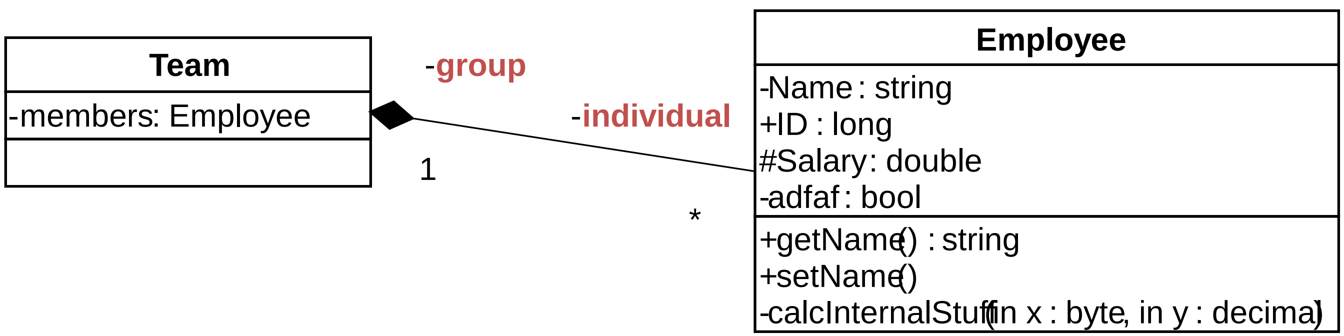
Multiplicities	**Meaning**
0..1	zero or one instance. The notation ***n . . m*** indicates ***n*** to ***m*** instances.
0..* *or* *****	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

UML Class Example



Association Details

- Can assign names to the ends of the association to give further information



Static vs. Dynamic Design

- Static design describes code structure and object relations
  - Class relations
  - Objects at design time
  - Doesn’t change
- Dynamic design shows communication between objects
  - Similarity to class relations
  - Can follow sequences of events
  - May change depending upon execution scenario
  - Called Object Diagrams

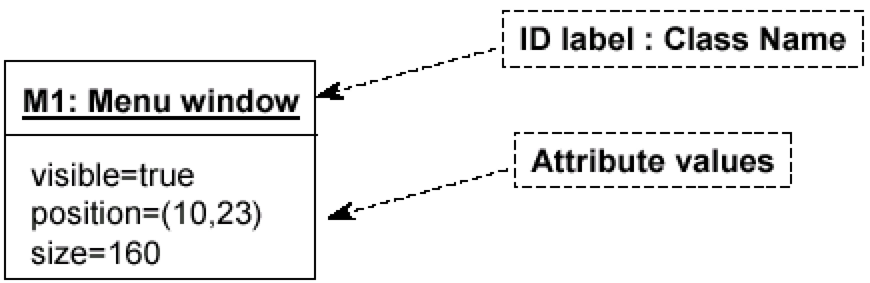
Object Diagrams

- Shows instances of Class Diagrams and links among them
  - An object diagram is a snapshot of the objects in a system
    - At a point in time
    - With a selected focus
      - Interactions – Sequence diagram
      - Message passing – Collaboration diagram
      - Operation – Deployment diagram

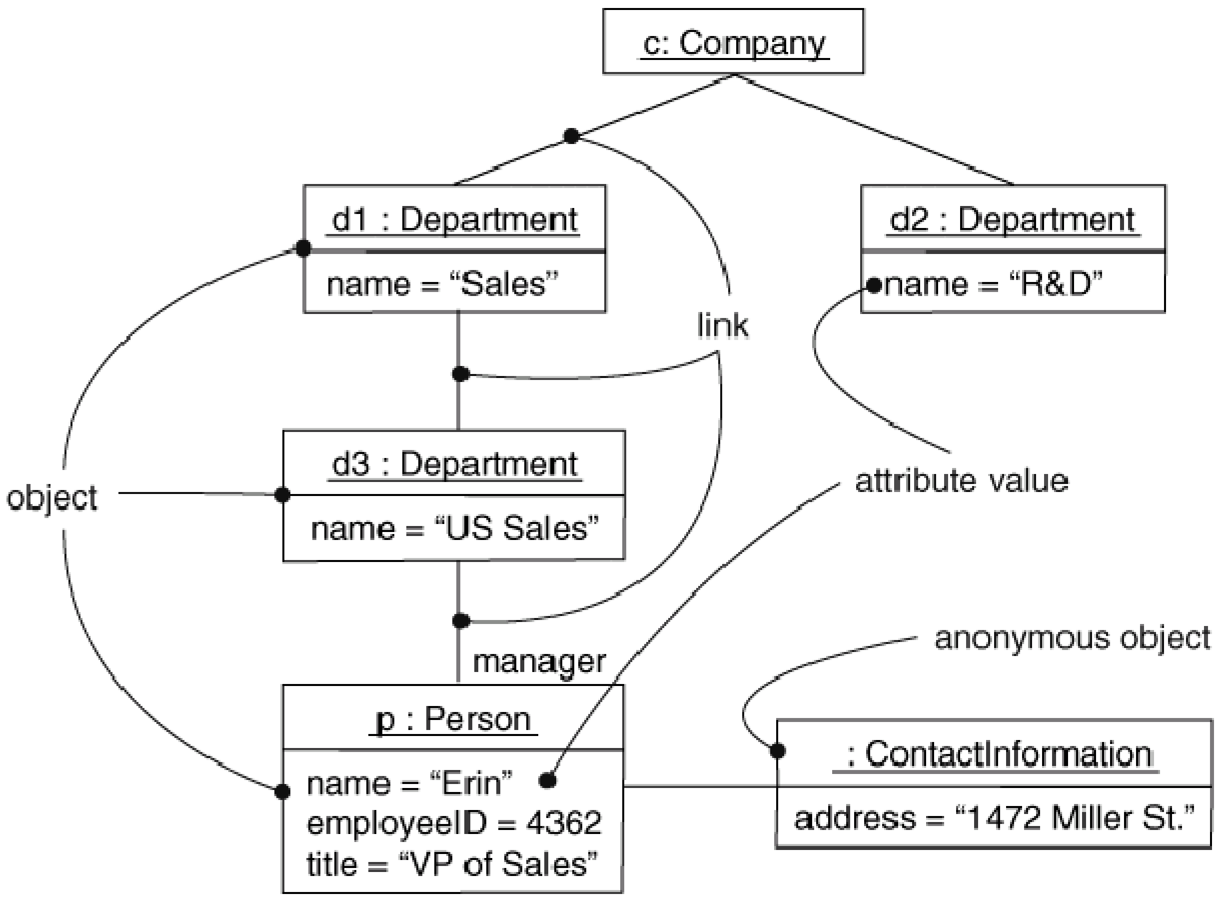
Object Diagrams

- Format is
  - Instance name : Class name
  - Attributes and Values

  - Example:



Objects and Links

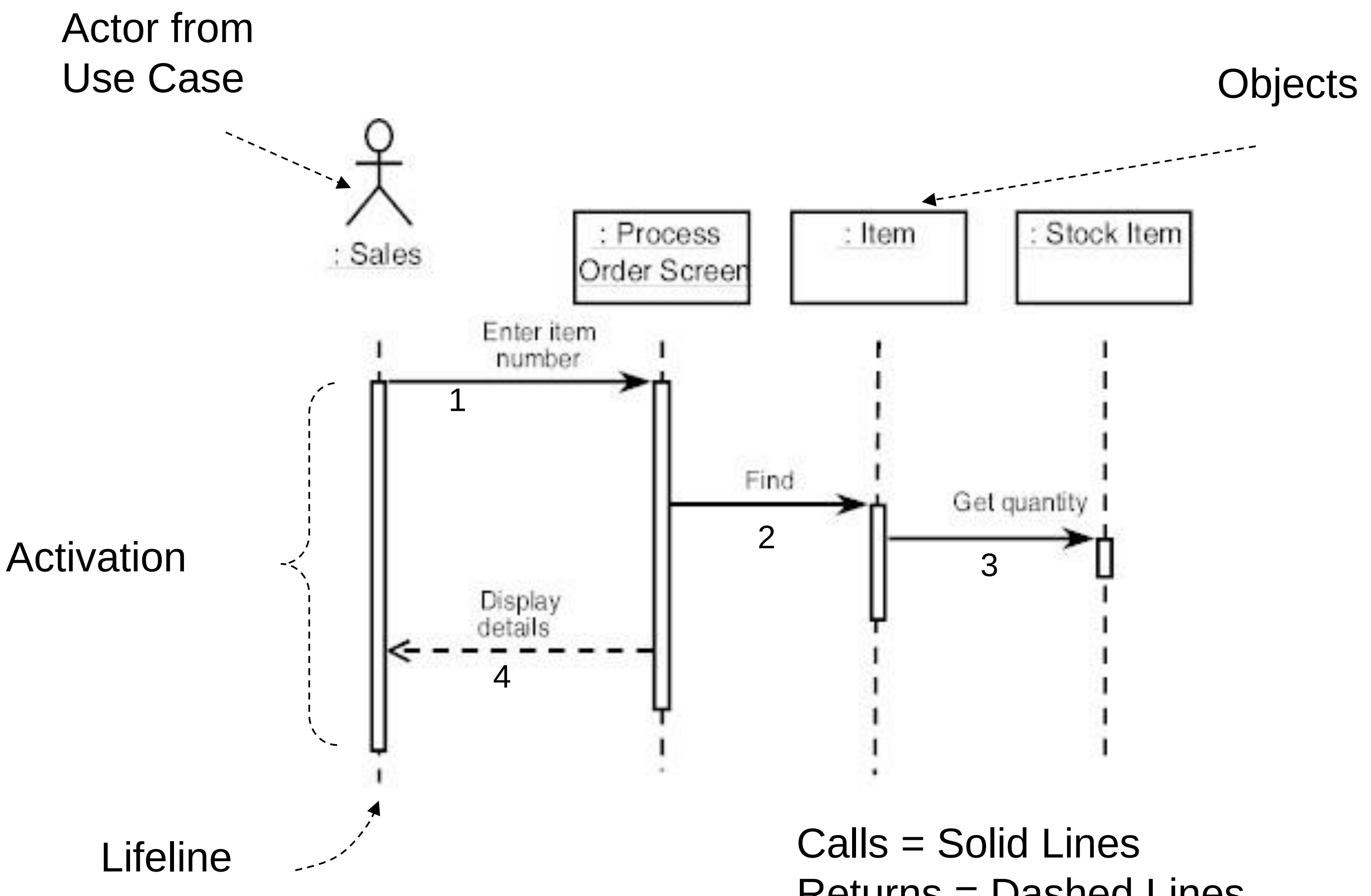


Can add association type and also message type

Interaction Diagrams

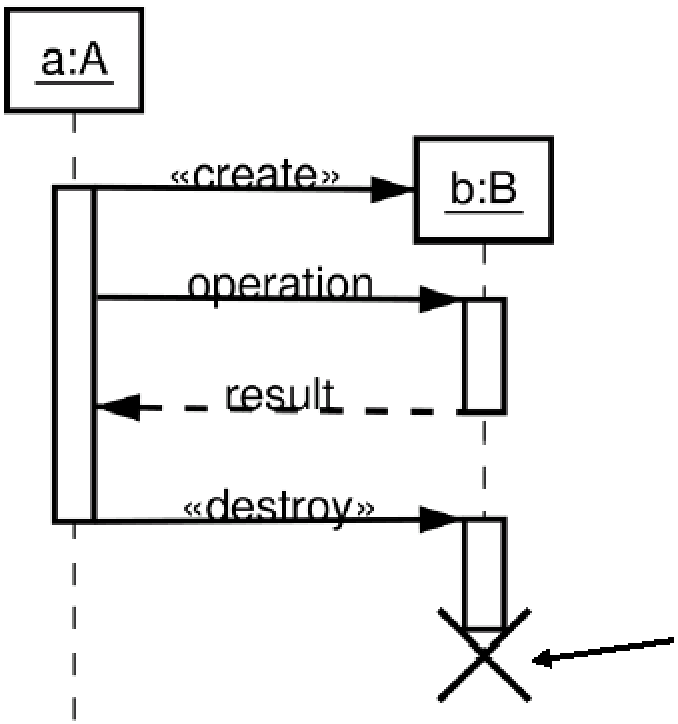
- Interaction diagrams are dynamic -- they describe how objects collaborate.
- A Sequence Diagram:
  - Indicates what messages are sent and when
  - Time progresses from top to bottom
  - Objects involved are listed left to right
  - Messages are sent left to right between objects in sequence

Sequence Diagram Format



Calls = Solid Lines
Returns = Dashed Lines

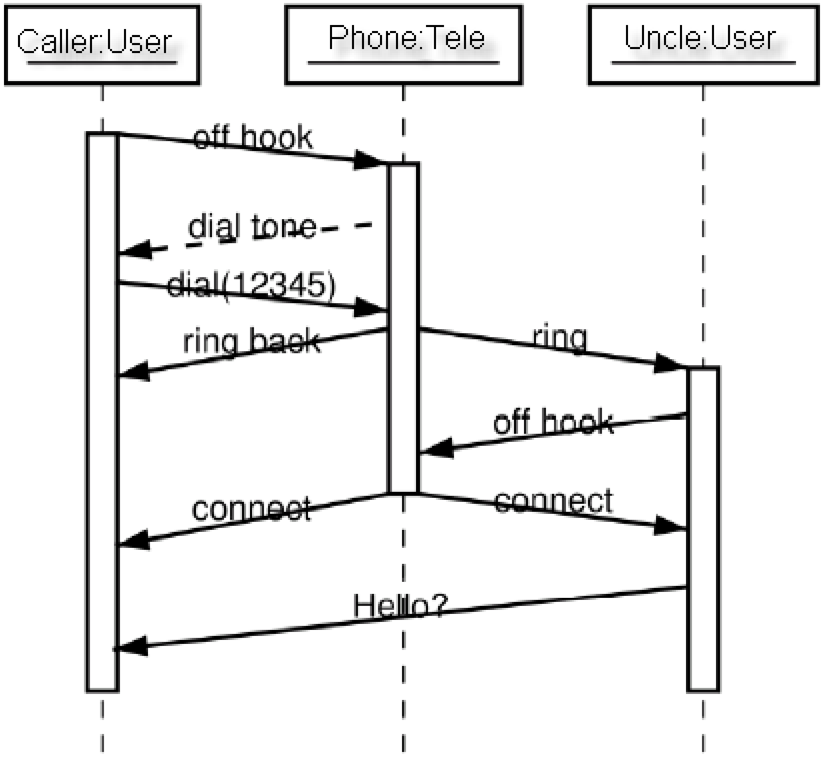
Sequence Diagram : Destruction



Shows Destruction of b
(and Construction)

Sequence Diagram : Timing

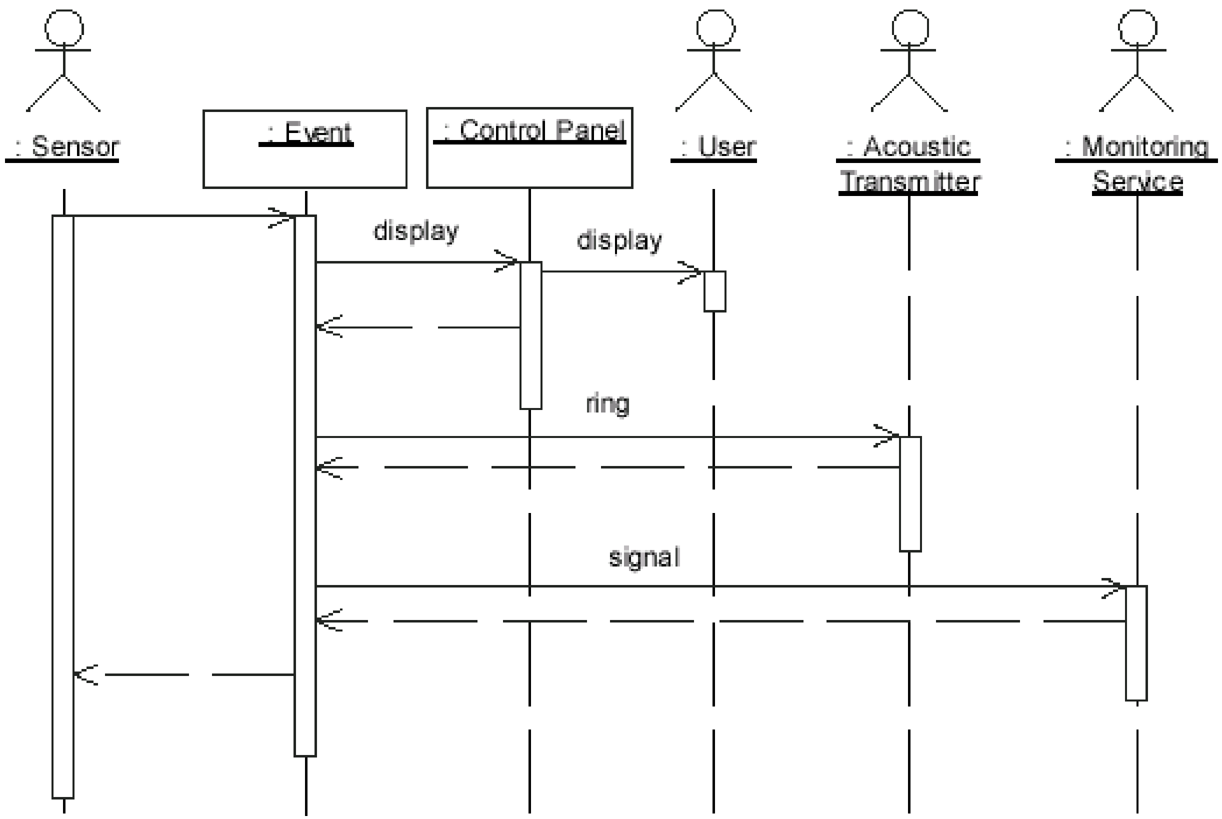
Slanted Lines show propagation delay of messages
Good for modeling real-time systems



If messages cross this is usually problematic – race conditions

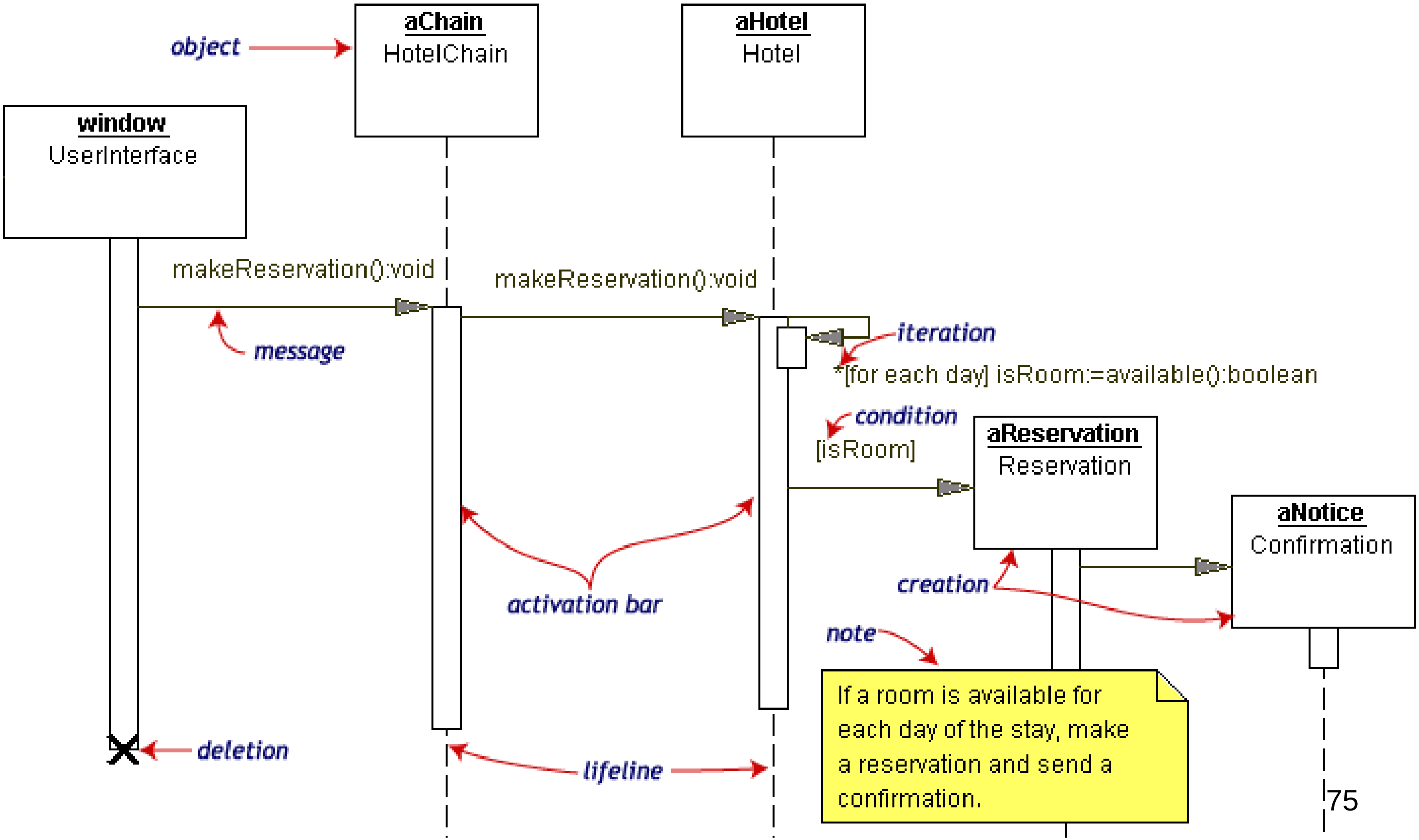
Sequence Example: Alarm System

- When the alarm goes off, it rings the alarm, puts a message on the display, notifies the monitoring service



Sequence Diagram Example

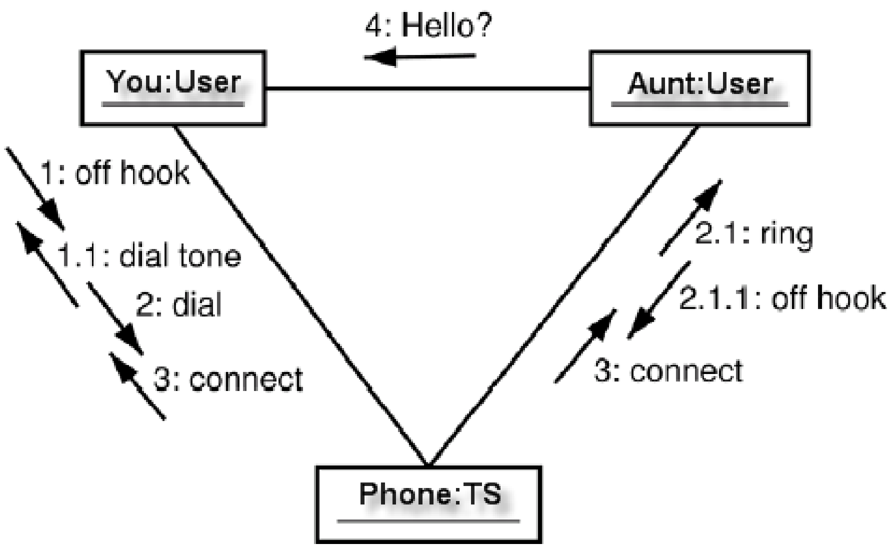
Hotel Reservation

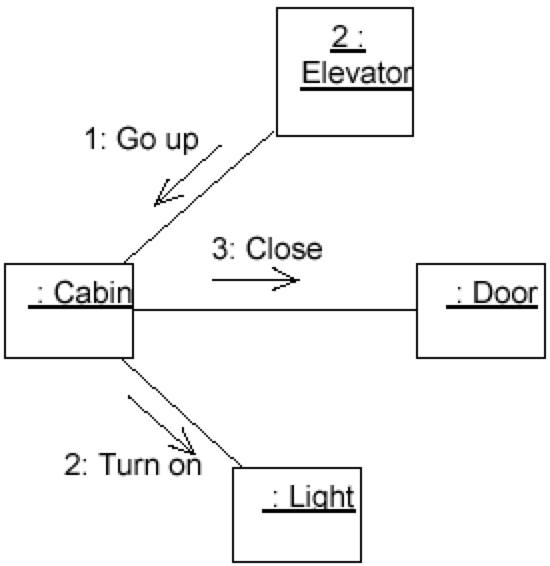


Collaboration Diagram

- Collaboration Diagrams show similar information to sequence diagrams, except that the vertical sequence is missing. In its place are:
  - Object Links - solid lines between the objects that interact
  - On the links are Messages - arrows with one or more message name that show the direction and names of the messages sent between objects
- Emphasis on static links as opposed to sequence in the sequence diagram

Collaboration Diagram





Summary

- Unified Modeling Language
- Class, Object, Association, etc..
- Class Diagram,
- Object Diagram
- Sequence Diagram
- Collaboration Diagram