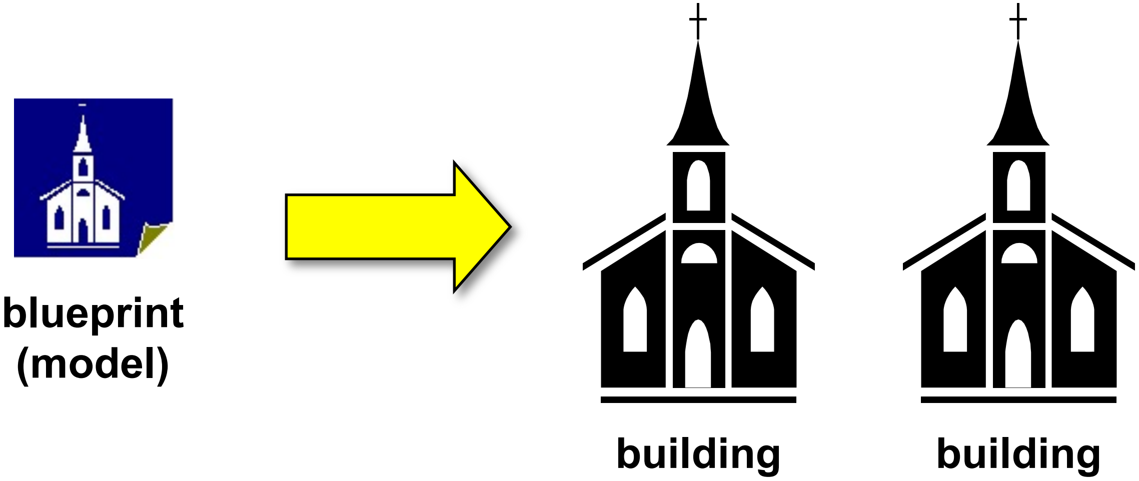
Software Modeling using UML

Software Engineering (Spring 2022)

IIIT Hyderabad

Models

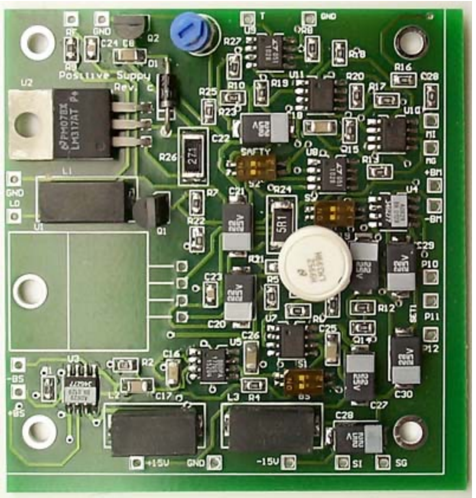
- A *model* is a description of something
  - *"a pattern for something to be made"* (Merriam-Webster)



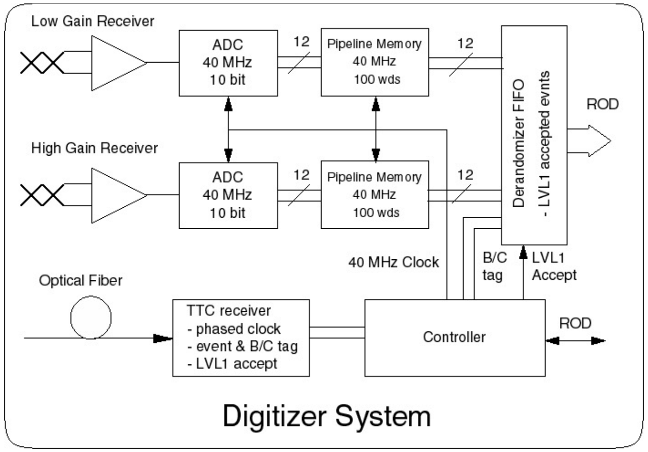
- model ≠ thing that is modeled
  - The Map is Not The Territory

Engineering Models

- Engineering model:
  A reduced representation of some system that highlights the properties of interest from a given viewpoint



Modeled system



Functional Model

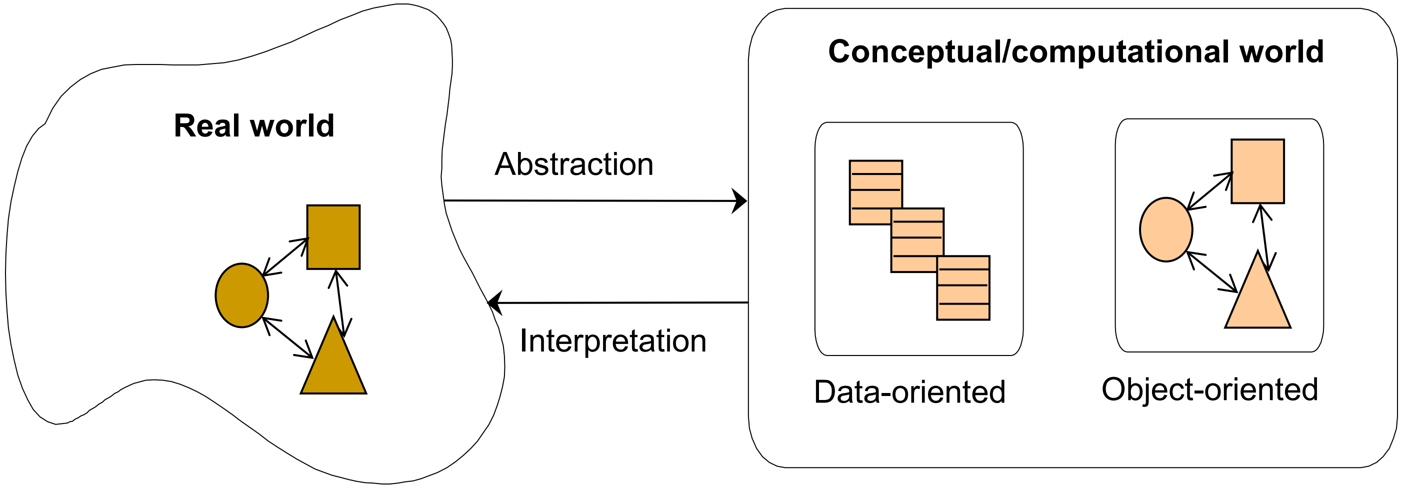
- We don't see everything at once
- We use a representation (notation) that is easily understood for the purpose on hand

Models

- How do we model?
- Modeling Maturity Level
  - Level 0: No specification
  - Level 1: Textual
  - Level 2: Text with Diagrams
  - Level 3: Models with Text
  - Level 4: Precise Models

Object-Oriented Modeling

- Uses object-orientation as a basis of modeling
- Models a system as a set of objects that interact with each others
- No semantic gap (or impedance mismatch)
- Seamless development process



Key Ideas of O-O Modeling

- Abstraction
- Encapsulation
- Relationship
  - Association: relationship between objects
  - Inheritance: mechanism to represent similarity among objects
- Object-oriented

  = object (class) + inheritance + message send

Objects vs. Classes

	Interpretation in the Real World	Representation in the Model
Object	An *object* represents anything in the real world that can be distinctly identified.	An *object* has an identity, a state, and a behavior.
Class	A *class* represents a set of objects with similar characteristics and behavior. These objects are called *instances* of the class.	A *class* characterizes the structure of states and behaviors that are shared by all of its instances.

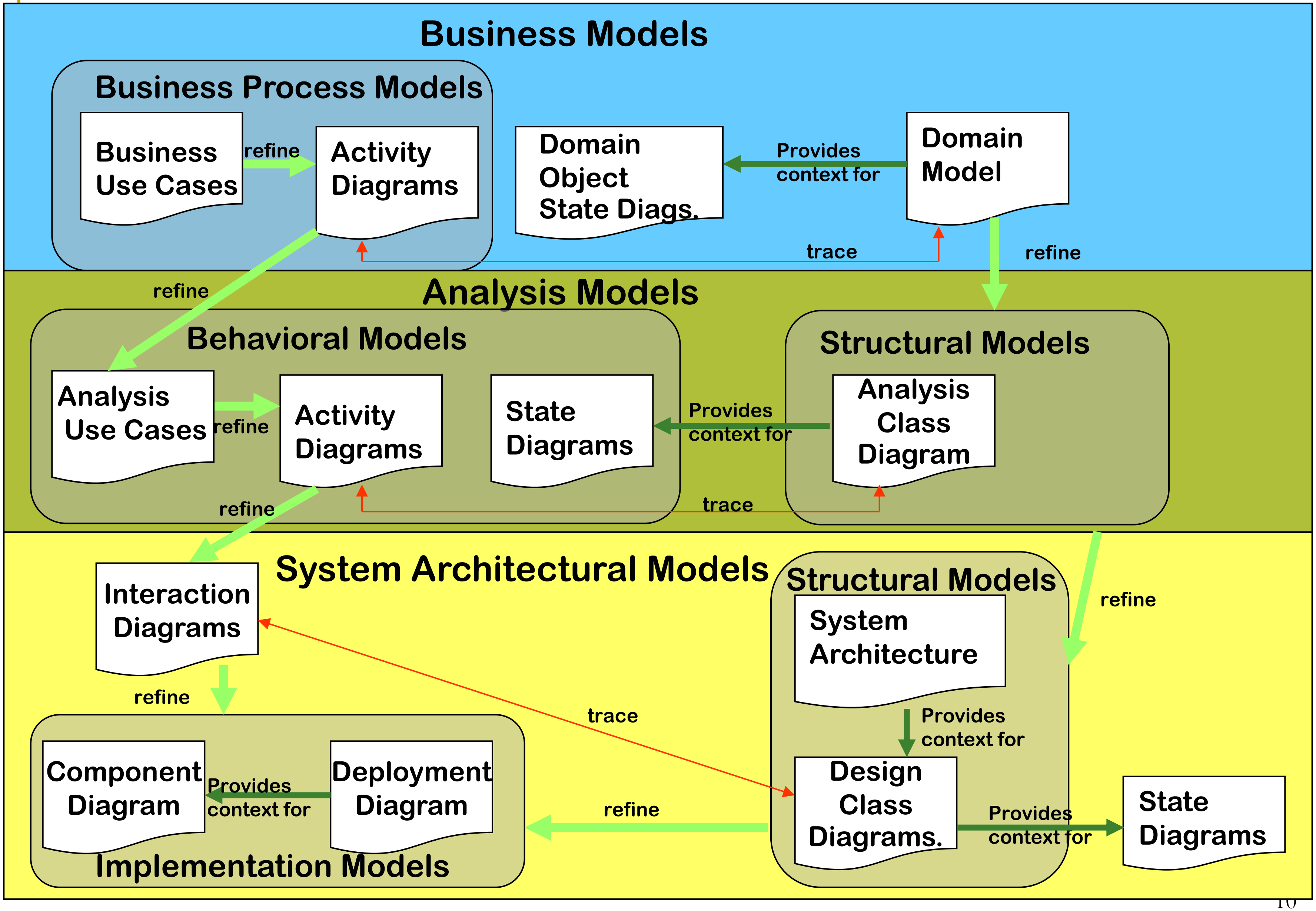
Unified Modeling Language (UML)

- Notation for object-oriented modeling
- Standardized by Object Management Group (OMG)
- Consists of 12+ different diagrams
  - Use case diagram
  - Class diagram
  - Statechart diagram
  - Sequence diagram
  - Communication diagram
  - Component diagram
  - …

What the UML is not

- Not an OO method or process
- Not a visual programming language
- Not a tool specification

A “Full” Process (using UML diagrams)



Business Models

Business Process Models

Business Use Cases

refine

Activity Diagrams

Domain Object State Diags.

Provides context for

Domain Model

trace

refine

refine

Analysis Models

Behavioral Models

Analysis Use Cases

refine

Activity Diagrams

State Diagrams

Provides context for

Structural Models

Analysis Class Diagram

refine

trace

System Architectural Models

Interaction Diagrams

refine

Structural Models

System Architecture

Provides context for

refine

trace

Component Diagram

Provides context for

Deployment Diagram

refine

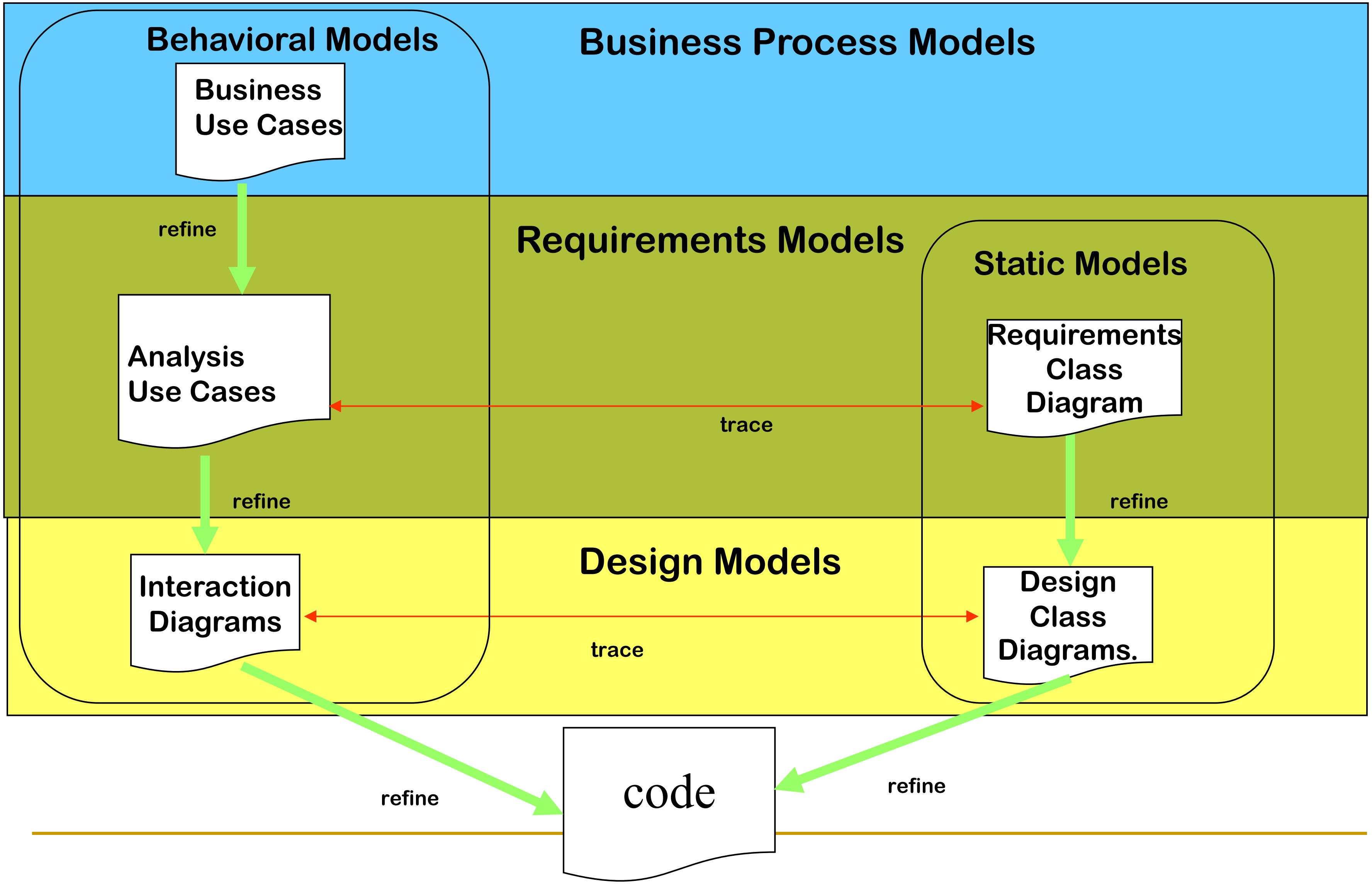
Design Class Diagrams.

Provides context for

State Diagrams

Implementation Models

An “UltraLite” Process

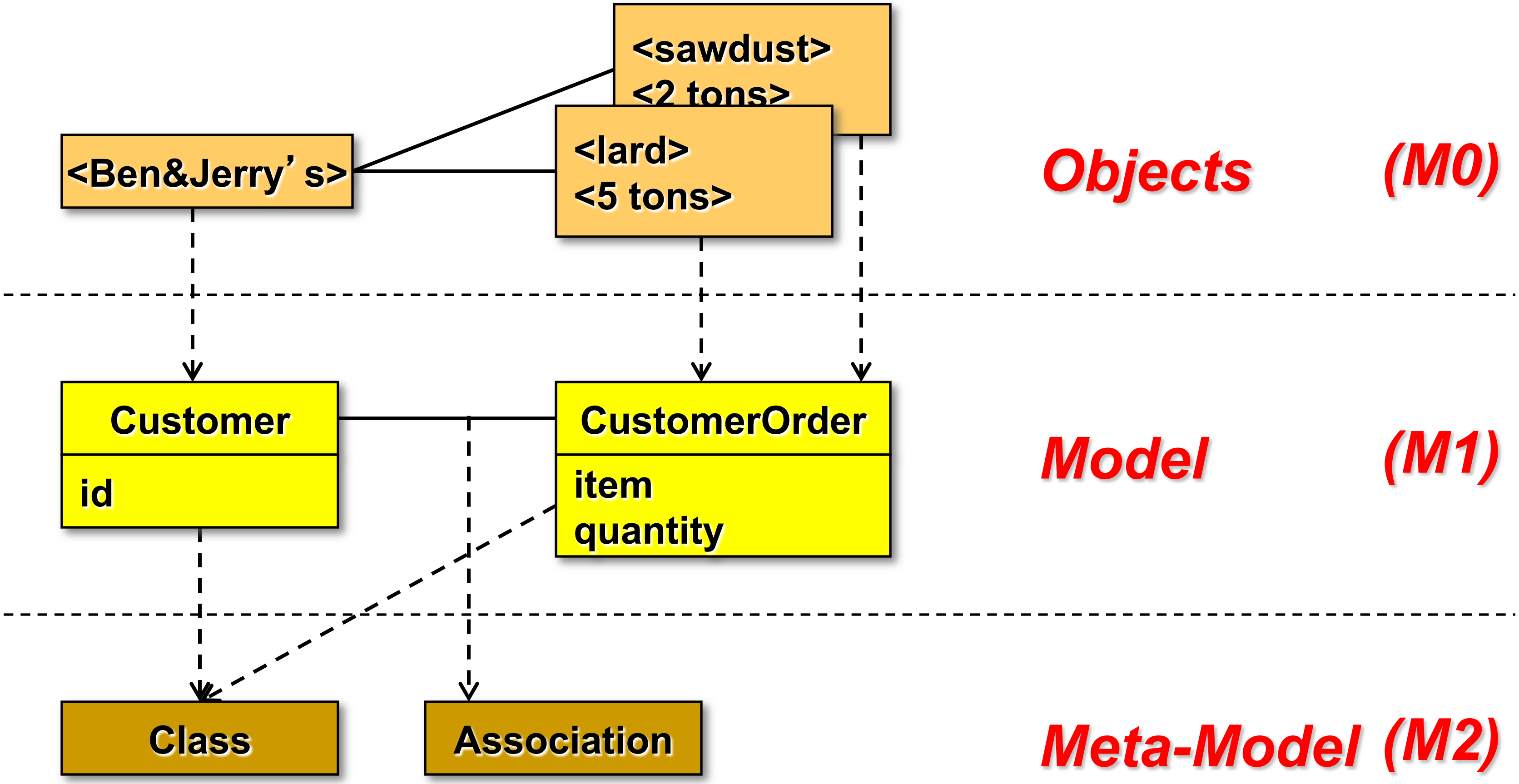


Static vs. Dynamic Models

- Static model
  - Describes static structure of a system
  - Consists of a set of objects (classes) and their relationships
  - Represented as class diagrams
- Dynamic model
  - Describes dynamic behavior of a system, such as state transitions and interactions (message sends)
  - Represented as statechart diagram, sequence diagrams, and collaboration diagrams

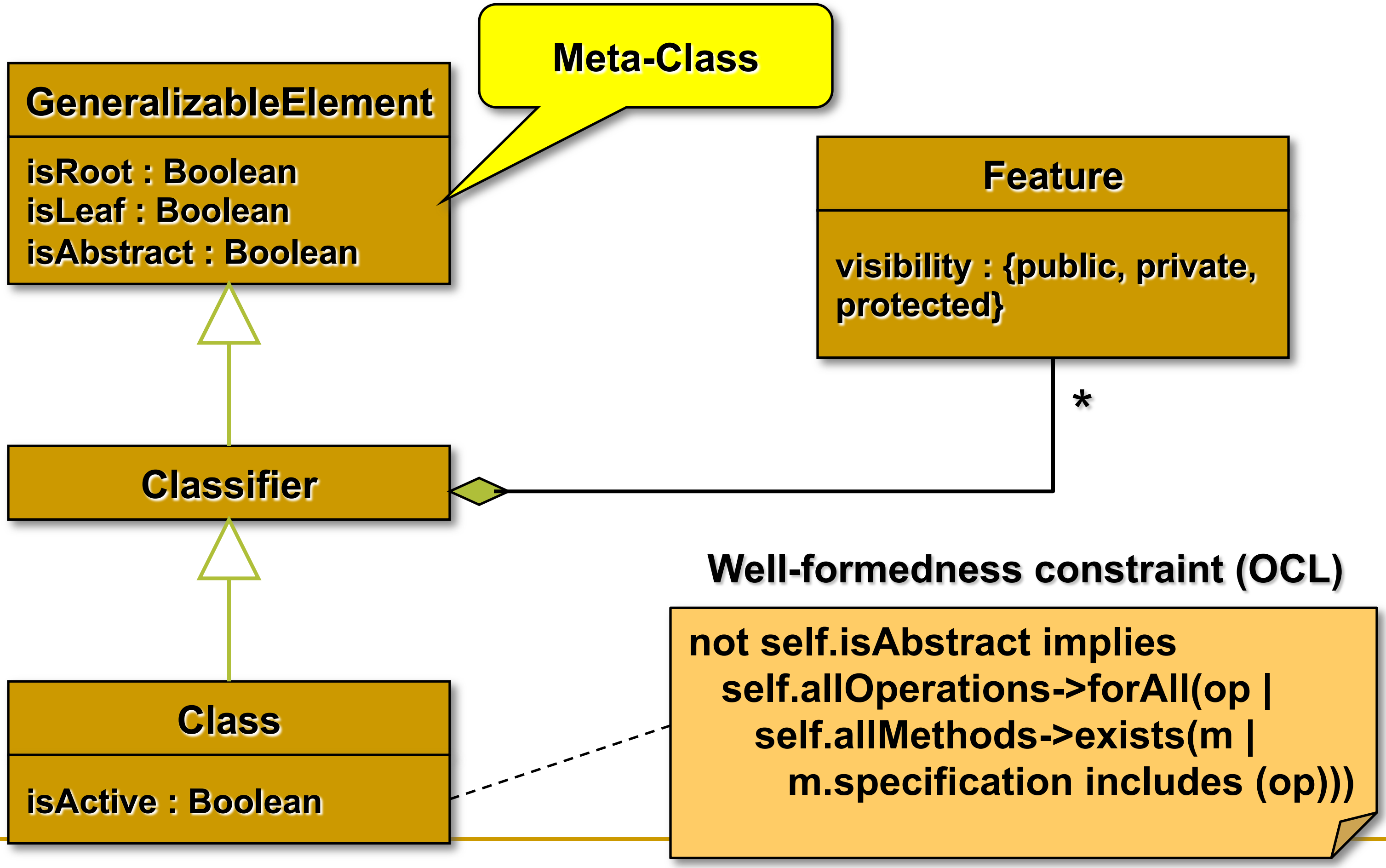
Models and Meta-Models

- Meta-models are simply Models of Models



The UML Meta-Model

- Is a UML Model of UML



Meta-Class

GeneralizableElement

isRoot : Boolean
isLeaf : Boolean
isAbstract : Boolean

Feature

visibility : {public, private, protected}

*

Classifier

Class

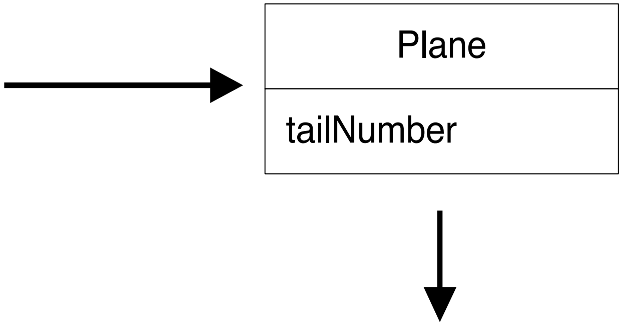
isActive : Boolean

Well-formedness constraint (OCL)

```
not self.isAbstract implies
  self.allOperations->forAll(op |
    self.allMethods->exists(m |
      m.specification includes (op)))
```

Domain concept
vs.
design representation of domain concept
vs.
code representation of domain concept

domain concept


Plane

tailNumber

visualization of domain concept

```
public class Plane
{
private String tailNumber;

public List getFlightHistory() {...}
}
```

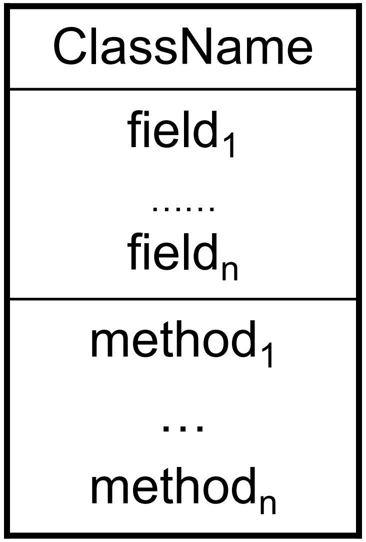
representation in an object-oriented programming language

UML Class Diagram

- Most common diagram in OO modeling
- Describes the static structure of a system
- Consist of:
  - Nodes representing classes
  - Links representing of relationships among classes
    - Inheritance
    - Association, including aggregation and composition
    - Dependency

Notation for Classes

- The UML notation for classes is a rectangular box with as many as three compartments.



ClassName

field$_1$

......

field$_n$

method$_1$

…

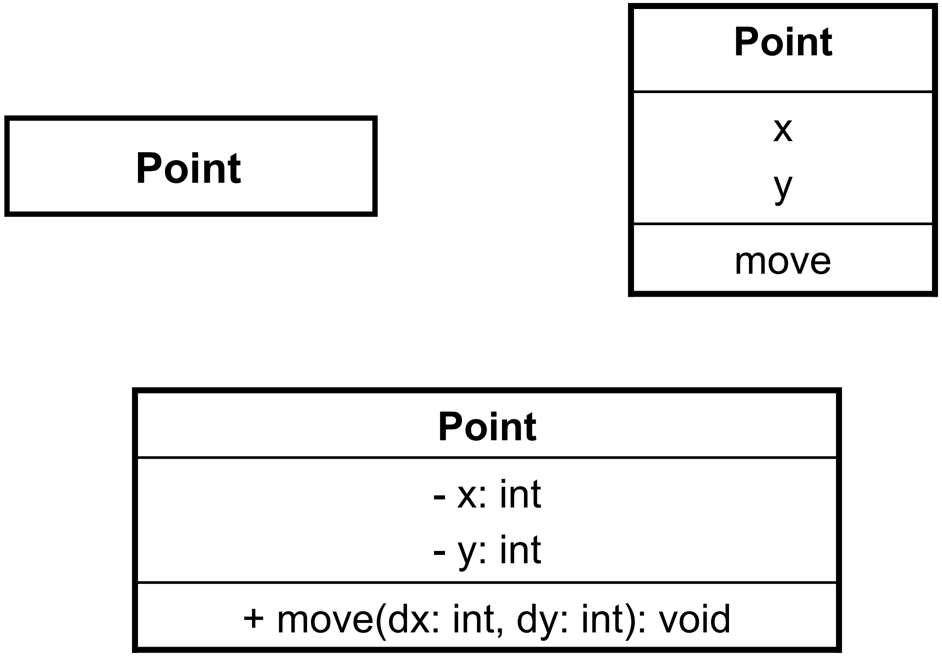
method$_n$

The top compartment show the class name.

The middle compartment contains the declarations of the fields, or *attributes,* of the class.

The bottom compartment contains the declarations of the methods of the class.

Example



Point

Point

x

y

move

Point

- x: int

- y: int

+ move(dx: int, dy: int): void

Field and Method Declarations in UML

- Field declarations
  - birthday: Date
  - +duration: int = 100
  - -students[1..MAX_SIZE]: Student
- Method declarations
  - +move(dx: int, dy: int): void
  - +getSize(): int

Visibility	Notation
public	+
protected	#
package	~
private	-

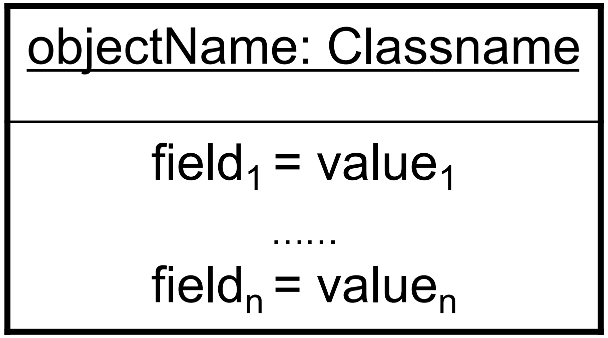
Exercise

- Draw a UML class diagram for the following Java code.

```
class Person {
  private String name;
  private Date birthday;
  public String getName() {
    // …
  }
  public Date getBirthday() {
    // …
  }
}
```

Notation for Objects

- Rectangular box with one or two compartments



objectName: Classname

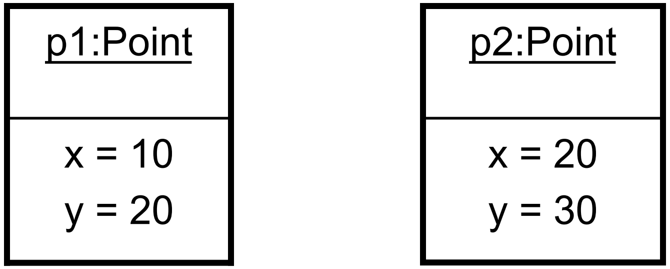
field$_1$ = value$_1$

……

field$_n$ = value$_n$

The top compartment shows the name of the object and its class.

The bottom compartment contains a list of the fields and their values.



p1:Point

x = 10

y = 20

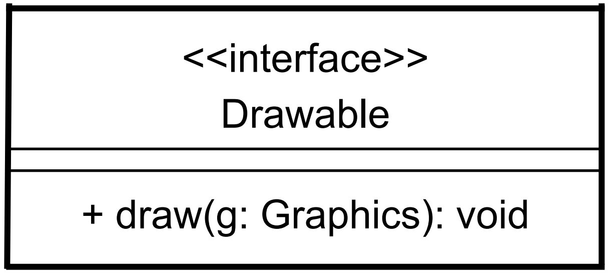
p2:Point

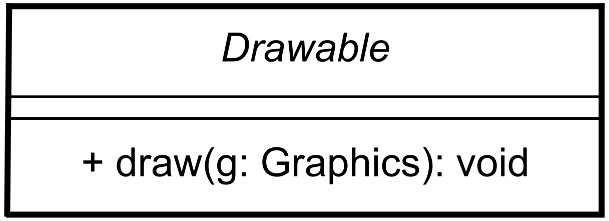
x = 20

y = 30

UML Notation for Interfaces

```
interface Drawable {
   void draw(Graphics g);
}
```



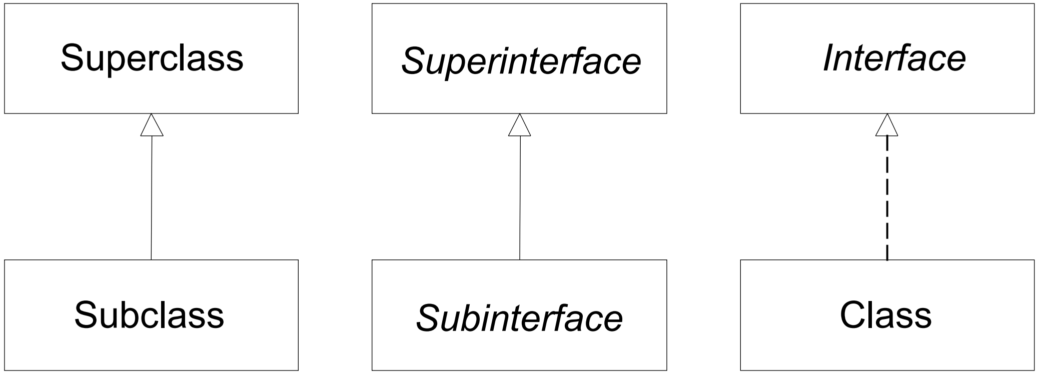


Inheritance in Java

- Important relationship in OO modeling
- Defines a relationship among classes and interfaces.
- Three kinds of inheritances
  - *extension* relation between two classes (*subclasses* and *superclasses*)
  - *extension* relation between two interfaces (*subinterfaces* and *superinterfaces*)
  - *implementation* relation between a class and an interface

Inheritance in UML

- An extension relation is called *specialization* and *generalization*.
- An implementation relation is called *realization*.



Superclass

Subclass

Superinterface

Subinterface

Interface

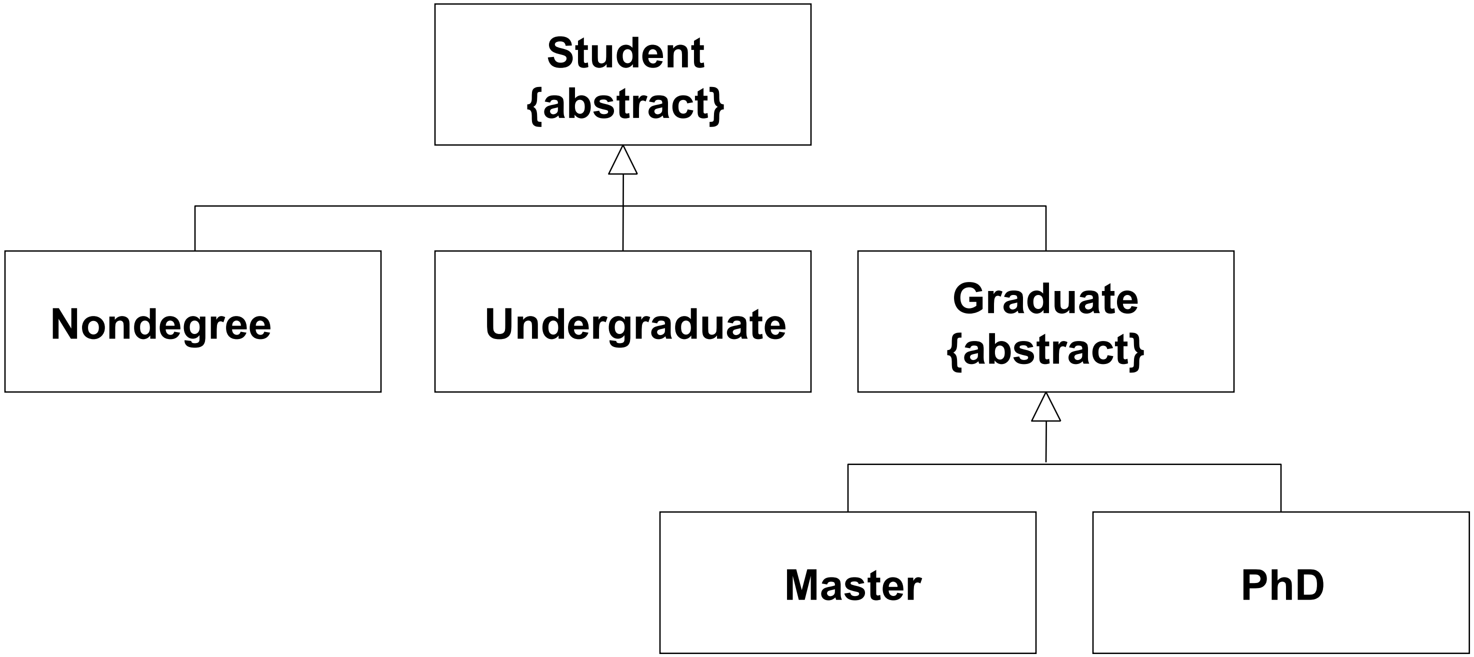
Class

extension of classes

extension of interfaces

implementation of interfaces

Example

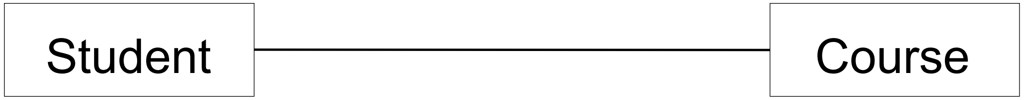

Student
{abstract}
Nondegree
Undergraduate
Graduate
{abstract}
Master
PhD

Exercise

- Draw a UML class diagram showing possible inheritance relationships among classes Person, Employee, and Manager.

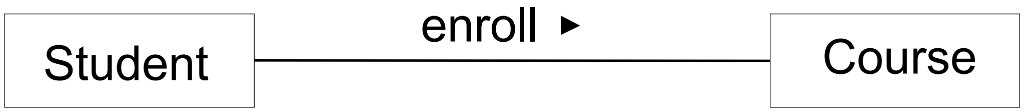
Association

- General binary relationships between classes
- Commonly represented as direct or indirect references between classes



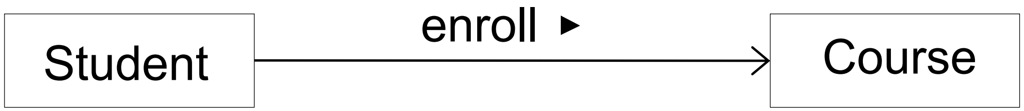
Association (Cont.)

- May have an optional label consisting of a name and a direction drawn as a solid arrowhead with no tail.
- The direction arrow indicates the direction of association with respect to the name.



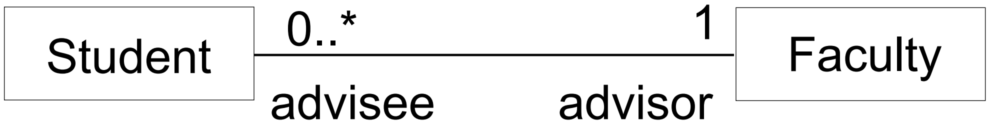
Association (Cont.)

- An arrow may be attached to the end of path to indicate that *navigation* is supported in that direction
- What if omitted?

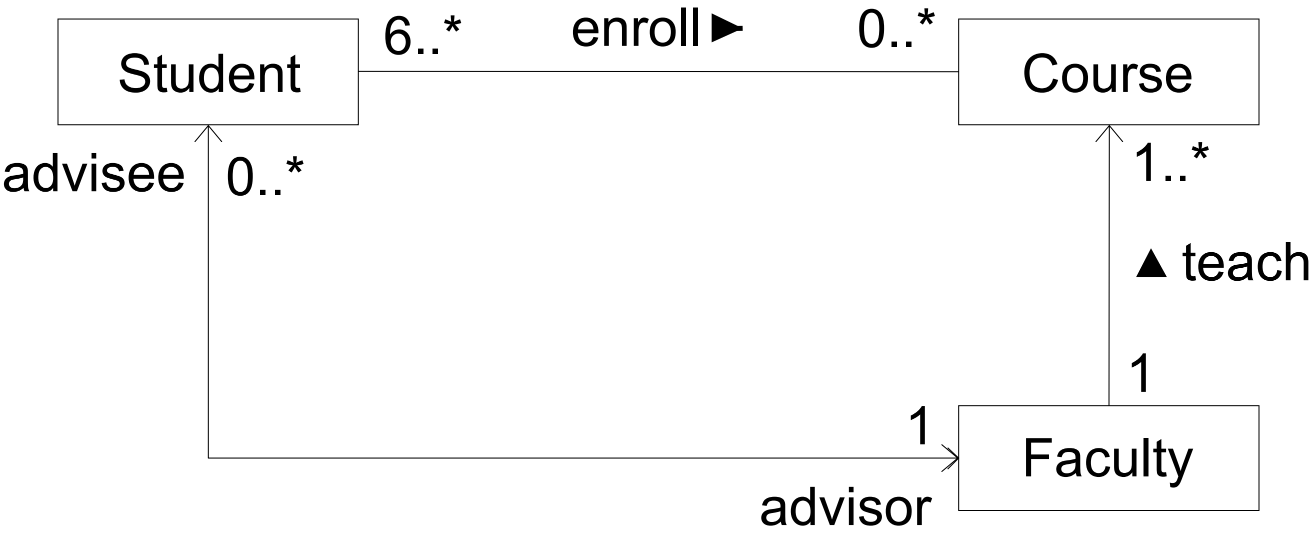


Association (Cont.)

- May have an optional *role name* and an optional *multiplicity specification.*
- The multiplicity specifies an integer interval, e.g.,
  - *l..u* closed (inclusive) range of integers
  - i singleton range
  - 0..* entire nonnegative integer, i.e., 0, 1, 2, …



Example

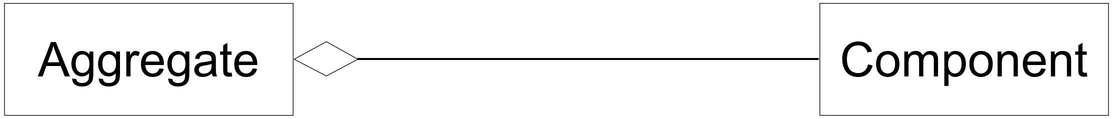

Student
6..*
enroll ►
0..*
Course
advisee
0..*
1..*
▲ teach
1
1
Faculty
advisor

Exercise

- Identify possible relationships among the following classes and draw a class diagram
  - Employee
  - Manager
  - Department

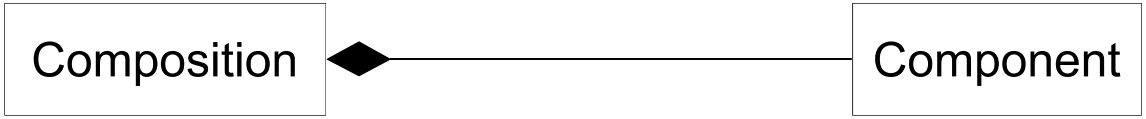
Aggregation

- Special form of association representing *has-a* or *part-whole* relationship.
- Distinguishes the whole (aggregate class) from its parts (component class).
- No relationship in the lifetime of the aggregate and the components (can exist separately).

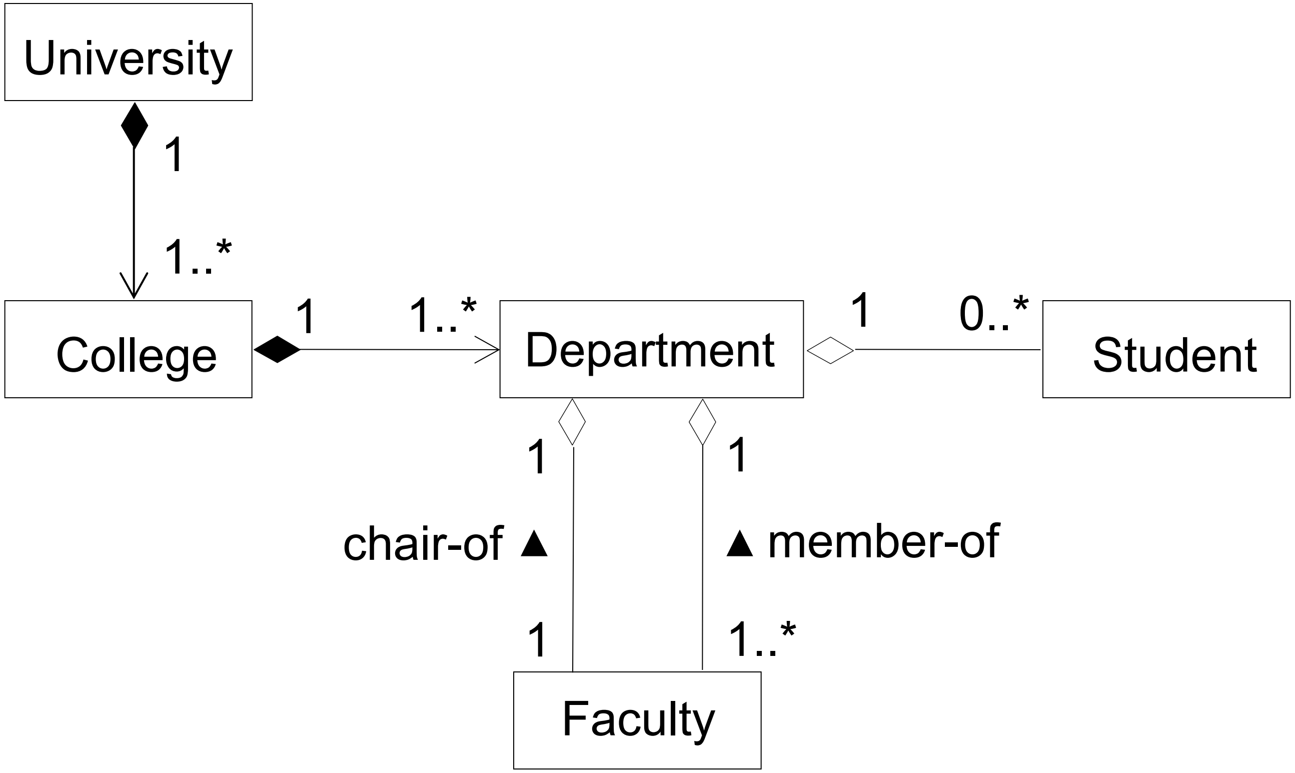


Composition

- Stronger form of aggregation
- Implies exclusive ownership of the component class by the aggregate class
- The lifetime of the components is entirely included in the lifetime of the aggregate (a component can not exist without its aggregate).

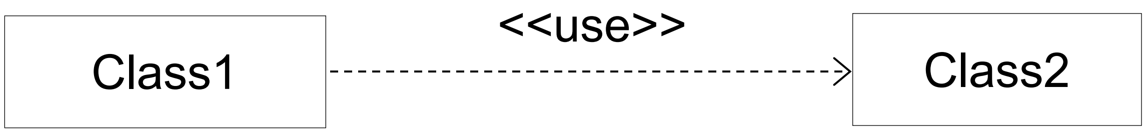


Example

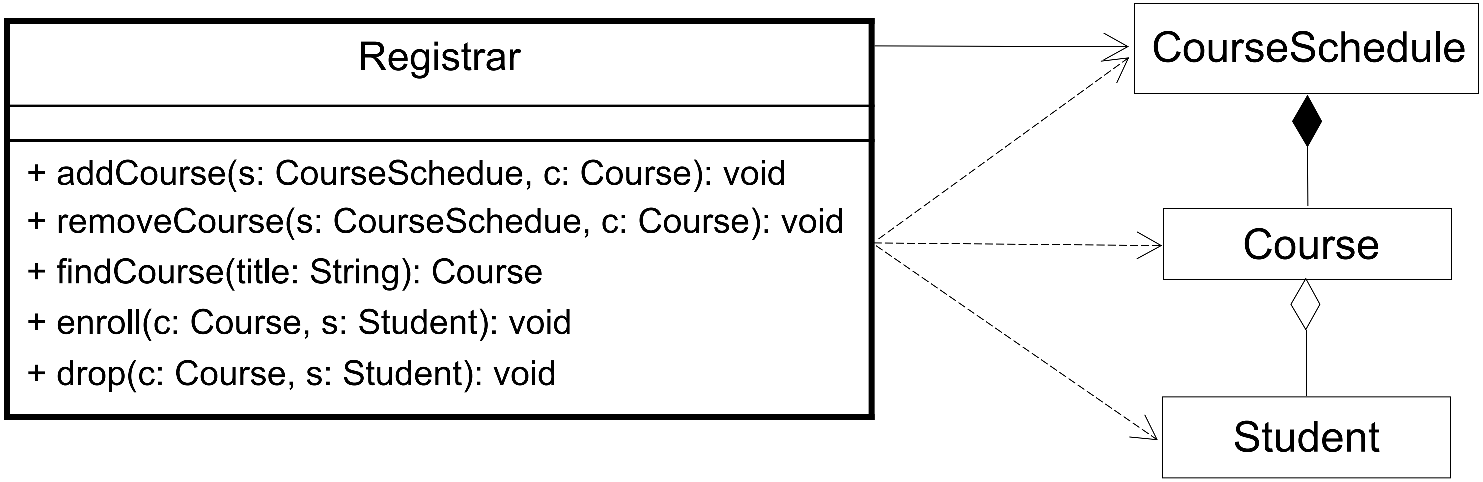

University
1
1..*
College
1
1..*
Department
1
0..*
Student
1
1
chair-of ▲
▲ member-of
1
1..*
Faculty

Dependency

- Relationship between the entities such that the proper operation of one entity depends on the presence of the other entity, and changes in one entity would affect the other entity.
- The common form of dependency is the *use* relation among classes.



Example



Registrar

+ addCourse(s: CourseSchedue, c: Course): void
+ removeCourse(s: CourseSchedue, c: Course): void
+ findCourse(title: String): Course
+ enroll(c: Course, s: Student): void
+ drop(c: Course, s: Student): void

CourseSchedule

Course

Student

Dependencies are most often omitted from the diagram unless they convey some significant information.

Group Exercise: E-book Store

Develop an OO model for an e-bookstore. The core requirements of the e-bookstore are to allow its customers to browse and order books, music CDs, and computer software through the Internet. The main functionalities of the system are to provide information about the titles it carries to help customers make purchasing decisions; handle customer registration, order processing, and shipping; and support management of the system, such as adding, deleting, and updating titles and customer information.

1. Identify classes. Classes can represent physical objects, people, organizations places, events, or concepts. Class names should be noun phrases.
2. Identify relevant fields and methods of the classes. Actions are modeled as the methods of classes. Method names should be verb phrases.
3. Identify any inheritance relationships among the classes and draw the class diagram representing inheritance relationships.
4. Identify any association relationships among the classes and draw the class diagram representing association relationships.
5. Identify any aggregation and composition relationships among the classes and draw the class diagram representing dependency relationships.

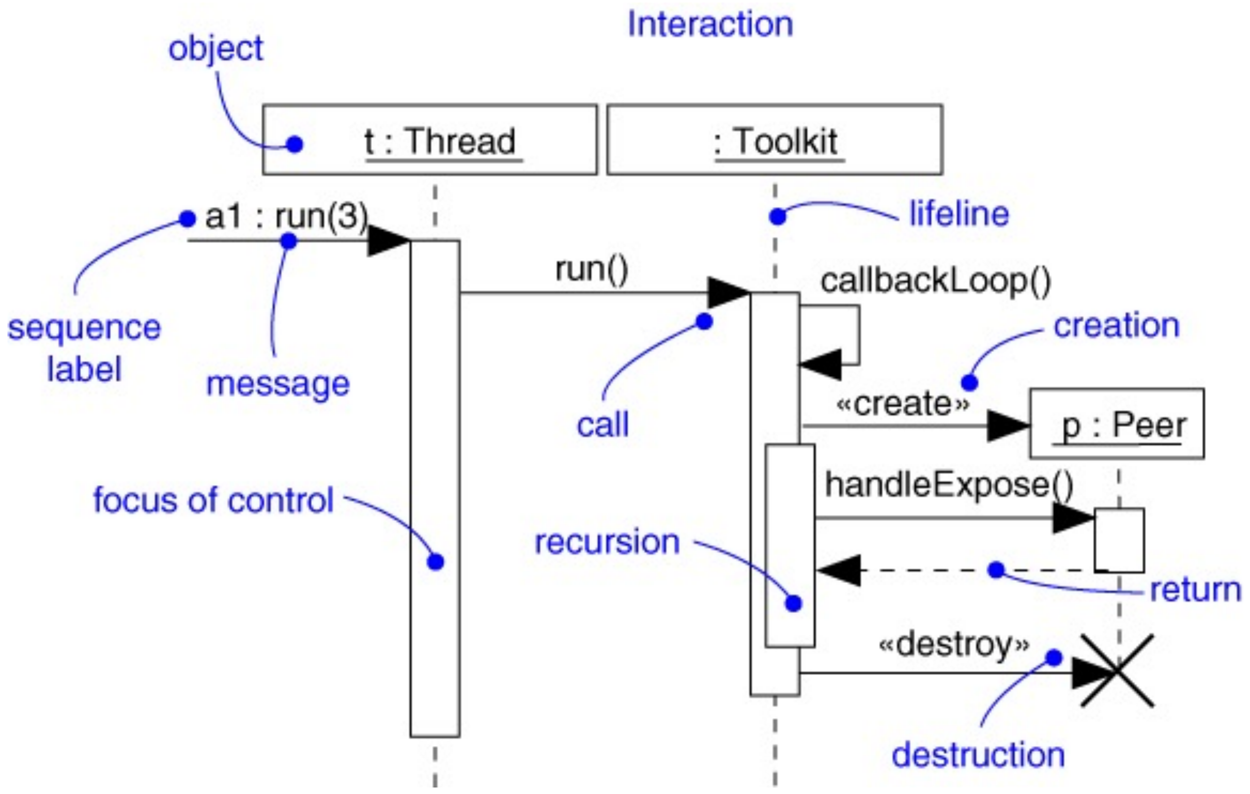
Interaction models

UML Interaction models

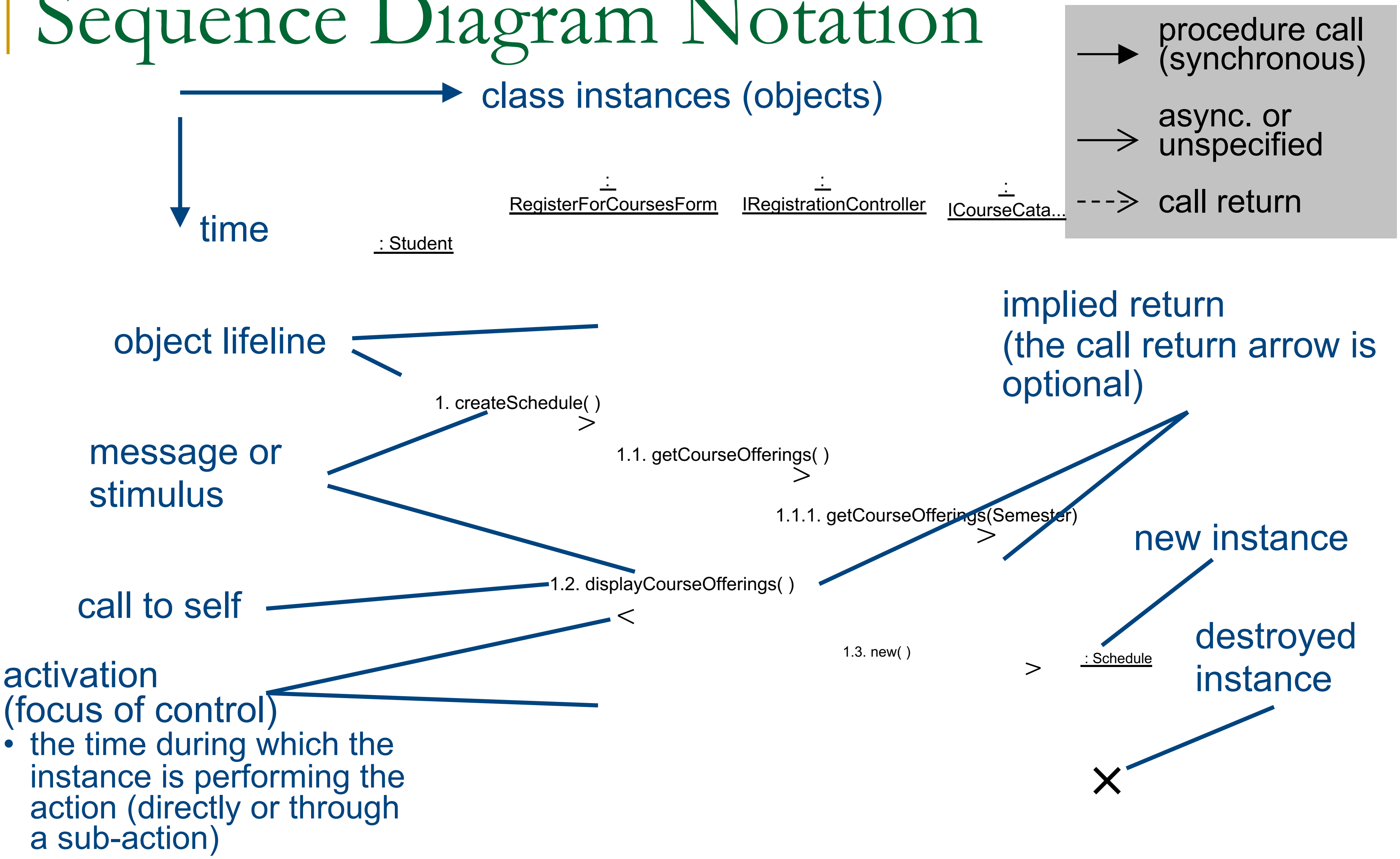
- An interaction model shows the interactions that take place between objects in a system
- An interaction “is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose” (UML user guide)
- Interaction models provide a view of system behavior

Sequence Diagram

- Captures dynamic behavior (time-oriented)
- Purpose
  - Model flow of control
  - Illustrate typical scenarios



Sequence Diagram Notation



: Student
: RegisterForCoursesForm
: IRegistrationController
: ICourseCata...
1. createSchedule( )
1.1. getCourseOfferings( )
1.1.1. getCourseOfferings(Semester)
1.2. displayCourseOfferings( )
1.3. new( )
: Schedule

procedure call (synchronous)

async. or unspecified

call return

class instances (objects)

time

object lifeline

message or stimulus

call to self

activation (focus of control)

- the time during which the instance is performing the action (directly or through a sub-action)

implied return (the call return arrow is optional)

new instance

destroyed instance

A More Complex Sequence Diagram (UML 2.0)

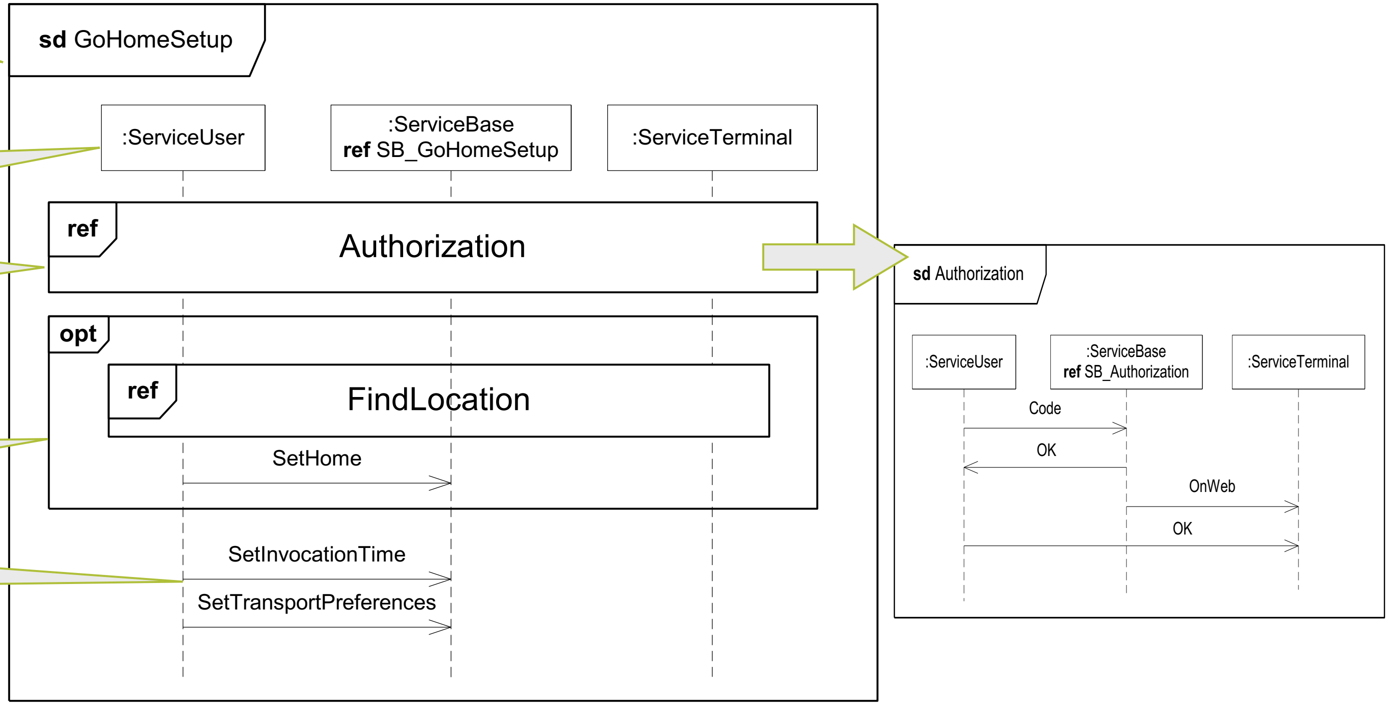
Frame and Name

Lifeline is an object

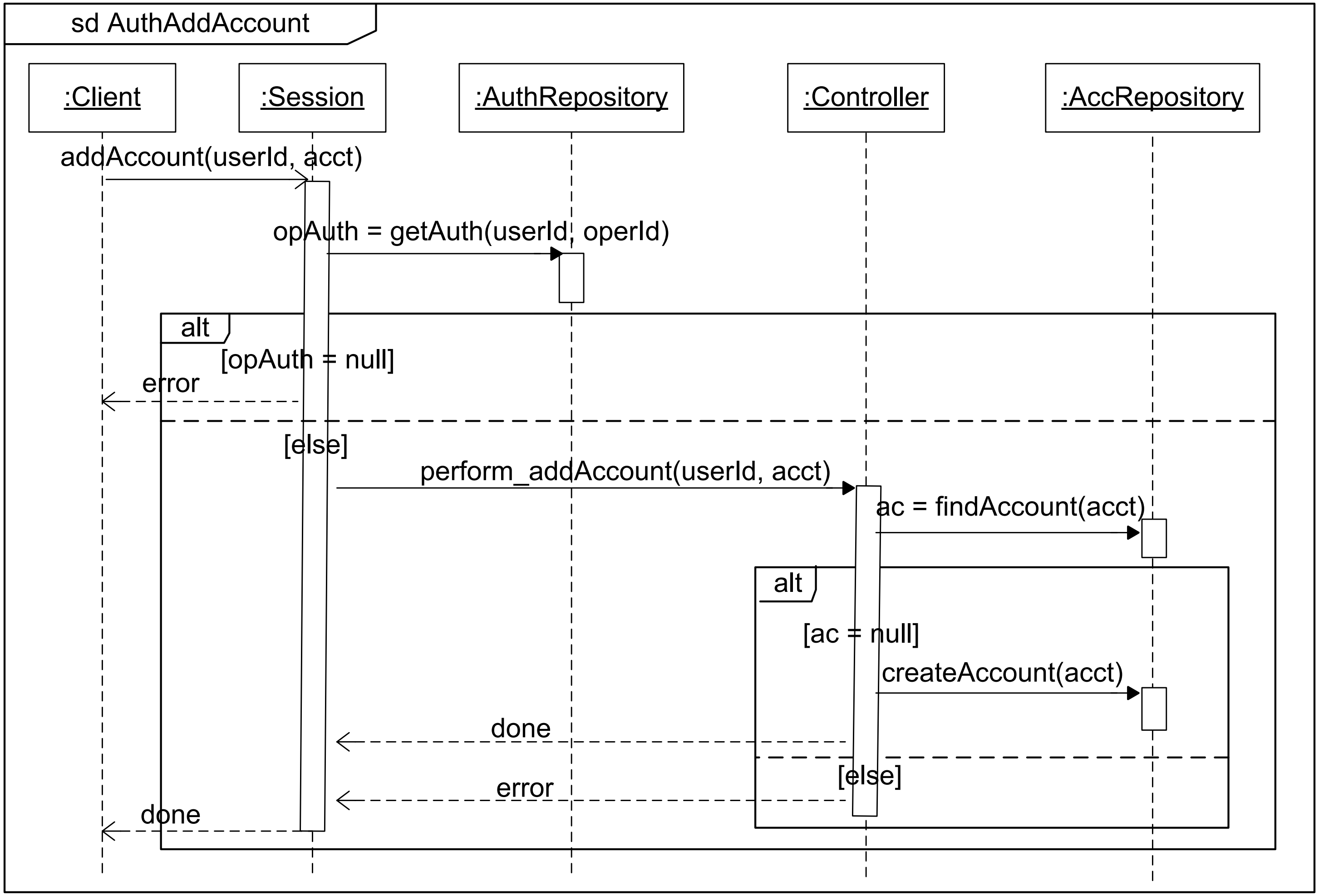
Interaction Occurrence

Combined Fragment

Plain asynchronous message



Sequence diagram - Example



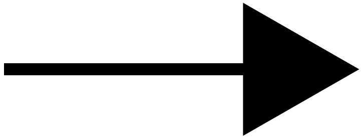
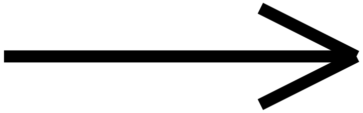
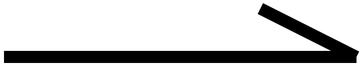
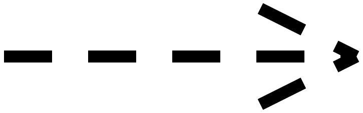
Combined Fragment Types

- Alternatives (**alt**)
  - choice of behaviors – at most one will execute
  - depends on the value of the guard ("else" guard supported)
- Option (**opt**)
  - Special case of alternative
- Break (**break**)
  - Represents an alternative that is executed instead of the remainder of the fragment (like a break in a loop)
- Parallel (**par**)
  - Concurrent (interleaved) sub-scenarios
- Negative (**neg**)
  - Identifies sequences that must not occur

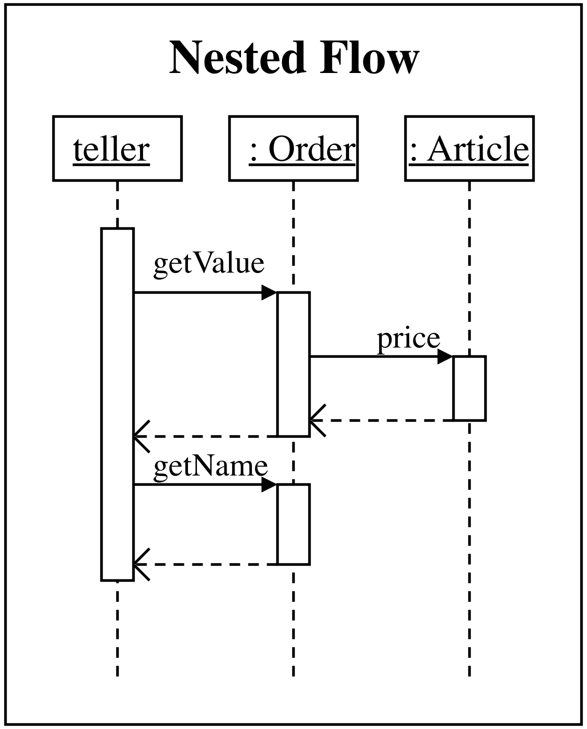
Combined Fragment Types

- Critical Region (**region**)
  - Traces cannot be interleaved with events on any of the participating lifelines
- Assertion (**assert**)
  - Only valid continuation
- Loop (**loop**)
  - Optional guard: [<min>, <max>, <Boolean-expression>]
  - No guard means no specified limit

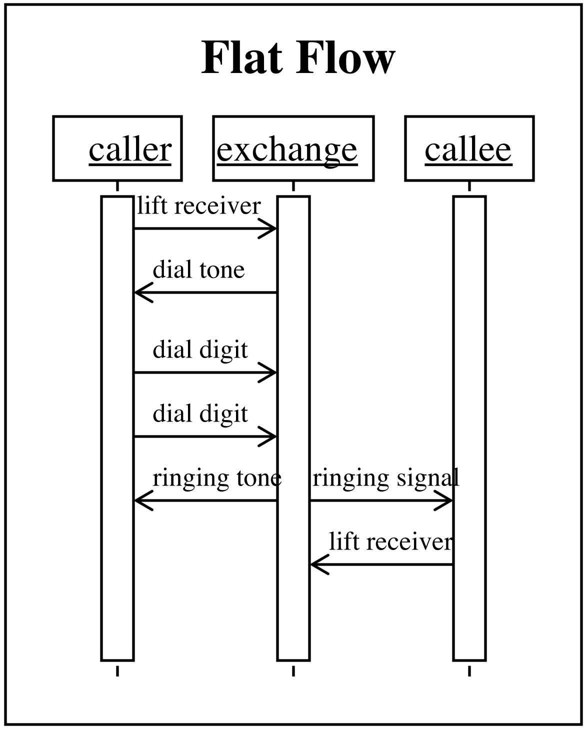
Different Kinds of Arrows

	Procedure call or other kind of nested flow of control
	UML 1.4: Asynchronous ~~Flat flow of control~~
	~~Explicit asynchronous flow of control~~ UML 1.4: Variant of async
	Return

Example: Different Arrows



Nested Flow



Flat Flow

Interaction Modeling Tips

- Set the context for the interaction.
- Express the flow from left to right and from top to bottom.
- Put active objects to the left/top and passive ones to the right/bottom.
- Use sequence diagrams
  - to show the explicit ordering between the stimuli
  - when modeling real-time