Object – Oriented Design with UML and Java

Part III: The Unified Modeling Language

Version 2011

The Unified Modeling Language (UML)

- The Unified Modeling Language has rich notational syntax.
- We will not cover it all. Nor should you feel compelled to use it all.

CSCI-4448 Students:

- For the purposes of this course (ie: tests) if it isn't in these notes, you don't have to learn it.

Use UML to:

- ***Analyze*** the domain & end-user requirements.
- ***Design*** your solution before you start to code.
- ***Visualize*** & ***document*** your design.
- ***Generate code*** (if precise, unambiguous & complete).

Diagram Types

Structure Diagrams:

- Class, Object, Component, Package, Deployment.

Behaviour Diagrams:

- Use Case, Activity, State Machine, Sequence, Communication.

There are other UML diagram types.

And there are useful diagrams that are not UML.

Modeling

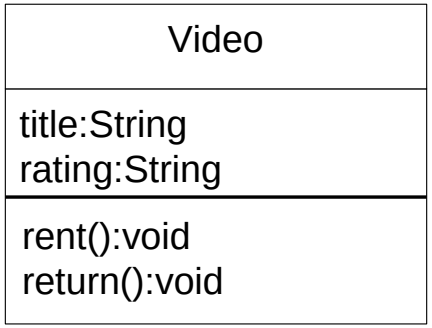
Successful OO designs usually begin with a visual model of the problem domain, involving both the domain experts and software designers alike.

- You don’t need to be a programmer to understand UML.
- A picture is worth a thousand words (1000 lines of code?)
- If it’s complicated and/or it needs to be understood by many people, make a model.
- The vocabulary used by the designers should not differ from that of the users & business analysts.
- Would a contractor build a house without blueprints?
- Understand the purpose of each model, its audience, the appropriate level of detail, and what information is therefore important or relevant.

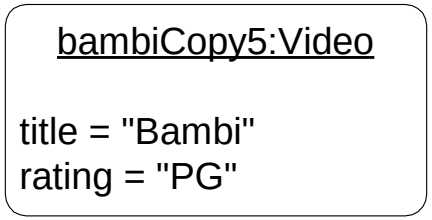
Modeling Advice

- Discuss models in small groups, with a white board.
- Use various types of models, not just class diagrams.
- Encourage your internal complexity alarm to alert you to poor design. If it is neither clear, simple, nor intuitively satisfying, it can likely be designed better.
- Minimize inter-class dependencies.
- Plan for future extensions.
- Use artistic license, *refactoring* your model as you see fit. This is fast & easy on a white board compared to changing code.
- Iterate, iterate, iterate...
- Foresight is not 20/20; that is why *iterative* approaches almost always result in higher quality designs.

Classes & Objects



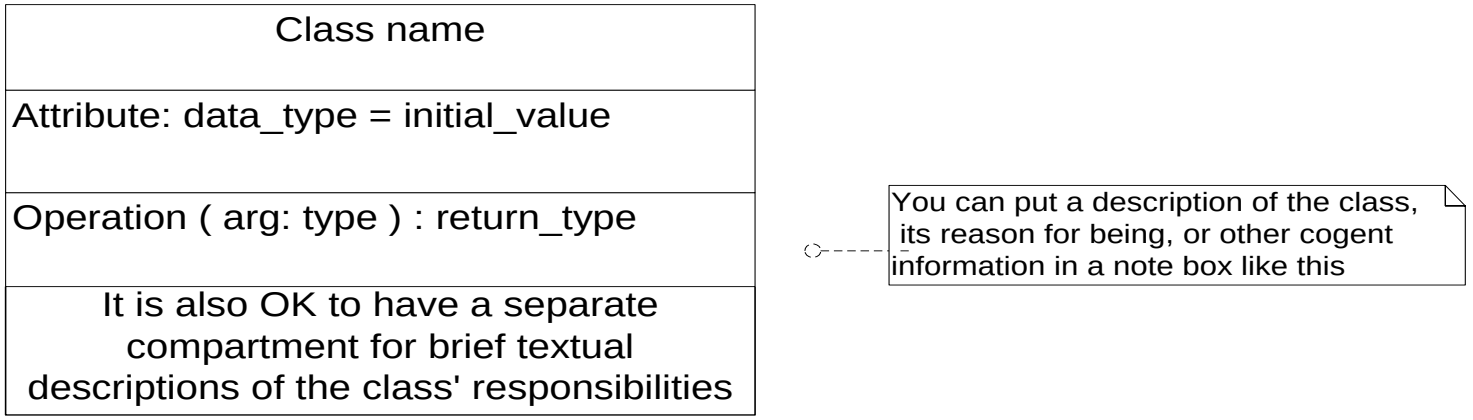
A Class



An Object (Instance)

- Note: The UML uses rectangular boxes for both objects and classes; we will use rounded corners on objects to help visually distinguish between the two. This approach is more whiteboard friendly.

Class Adornments



- Even though attributes are shown first, remember:
  - Class elaboration should be *responsibility driven*.
- Only show relevant information.

Class Adornments

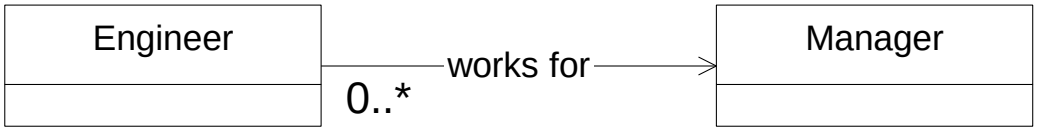
Attribute and Method Visibility

(degree of encapsulation):

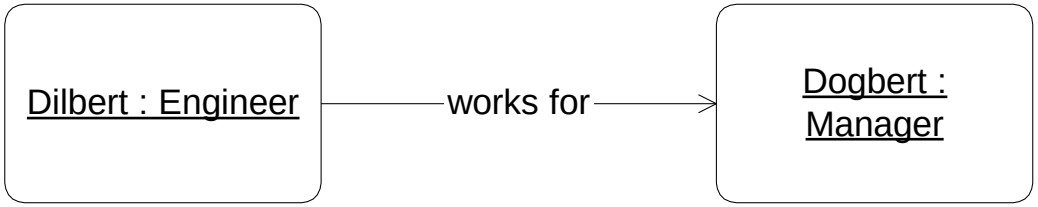
- **+ public**
- **- private**
- **# protected**
- **~ Java's package visibility**
- **/ derived**
- **$ static - $ is not standard UML**

Person
+publicAttribute #protectedAttribute -dob // private / age // derived -$numPeople << or >> -numPeople
+$getNumPeople() +getAge( )

Links and Associations



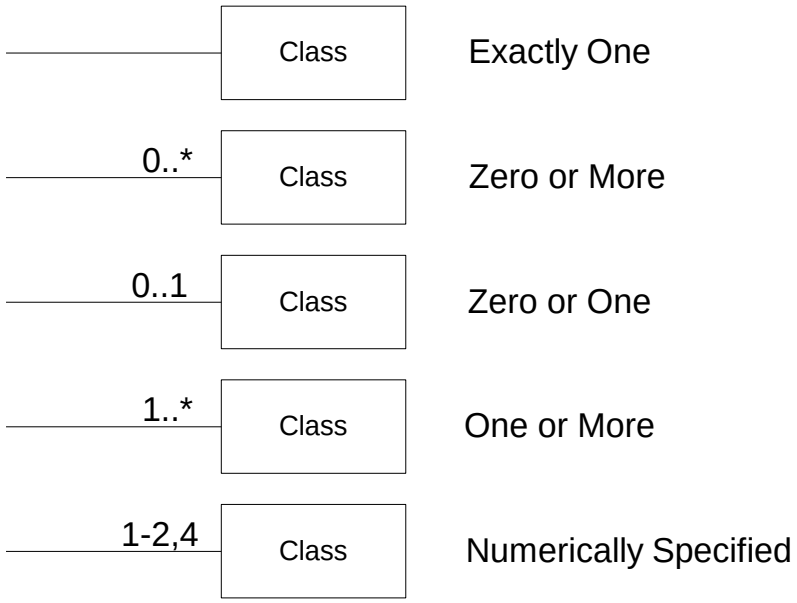
Associations connect classes



Links connect objects

Multiplicity (Cardinality)

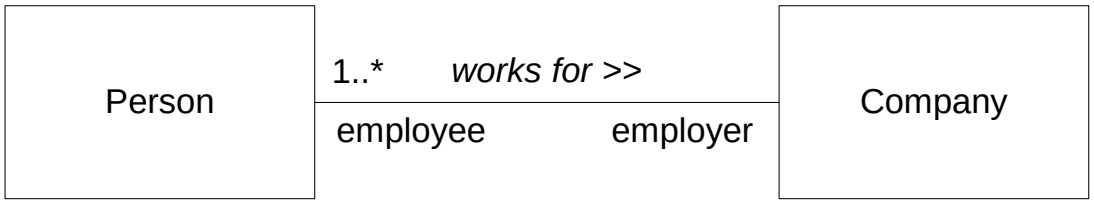
- Specify constraints on the number of instances (objects) on either end of the association.

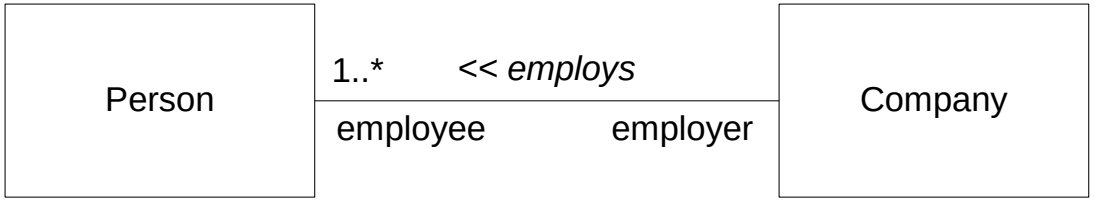


Note: ***n*** and ***** may be used instead of **0..***

Roles and Association Names

- Name the **role** as it appears to the class at the other end of the association. Usually a noun.
- Name the **association** in a way that creates a subject – verb – object sentence (may require an arrow to specify the direction of the association).





Stereotypes

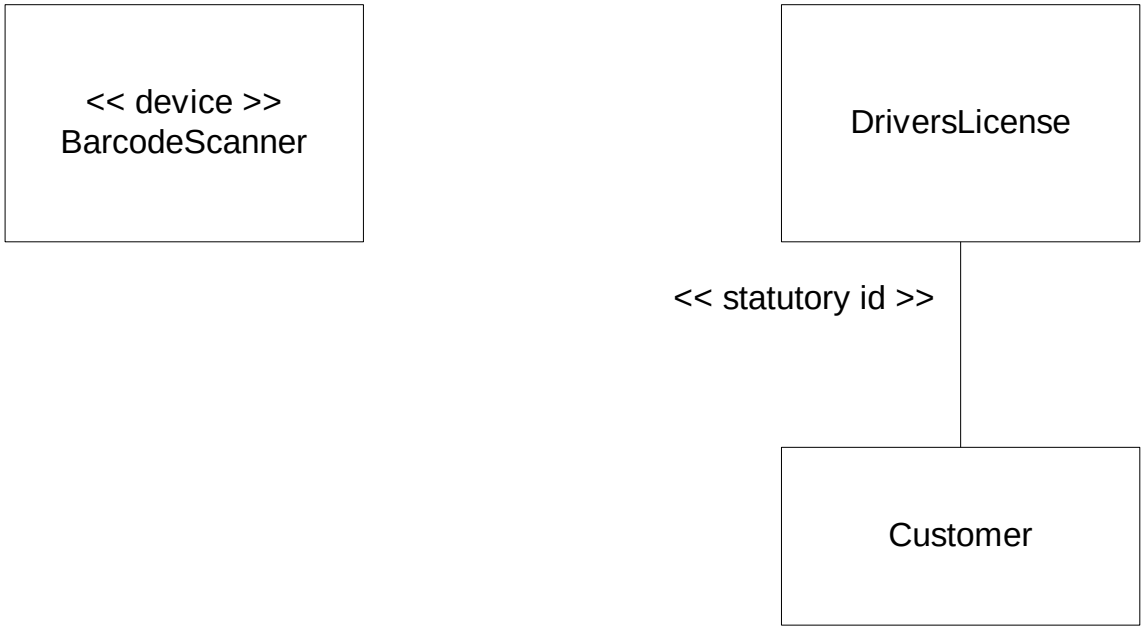
A categorization of modeling entities.

- Often applied to classes, associations, and methods.
- A way of extending the UML; for defining your own modeling elements, specific to your problem.
- Some stereotypes are recognized by CASE tool code generators.

<< abstract >>, << interface >>, << exception >>,
<< instantiates >>, << subsystem >>, << extends >>,
<< instance of >>, << friend >>, << JavaBean >>,
<< constructor >>, << thread >>, << uses >>,
<< global >>, << creates >>, << invent your own >>

Stereotype Examples

- Modelers are free to invent their own stereotypes.

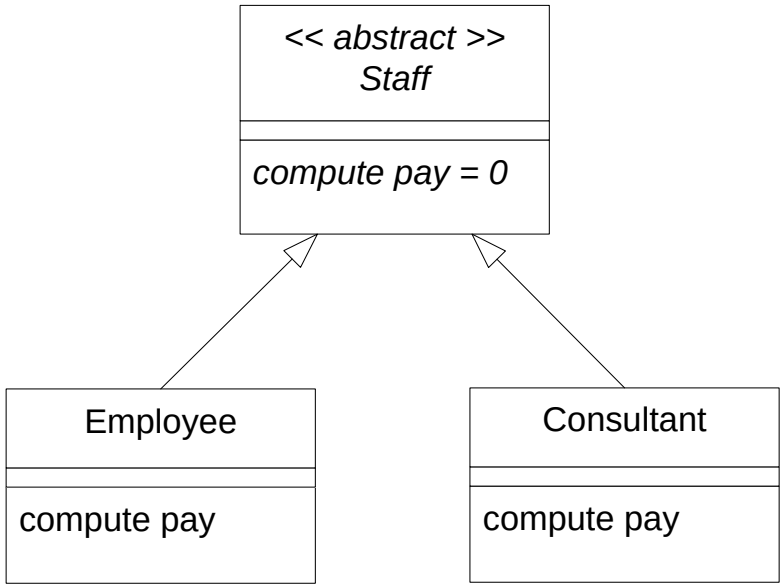


Abstract Classes & Methods

Indicated in UML with *italics*.

Italics are not whiteboard friendly.

- We use **`=0`** for abstract methods (derived from C++)
- We use **`<< abstract >>`** for abstract classes.

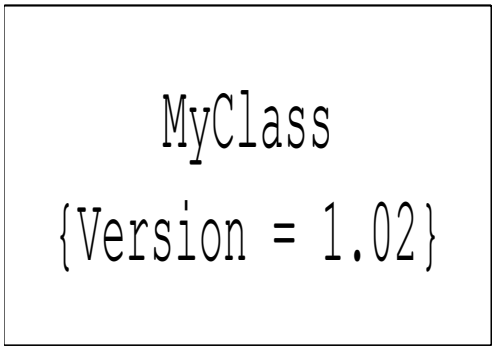


Tagged Values

- Another UML extensibility mechanism, allowing you to add **`{name = value}`** properties to your model.

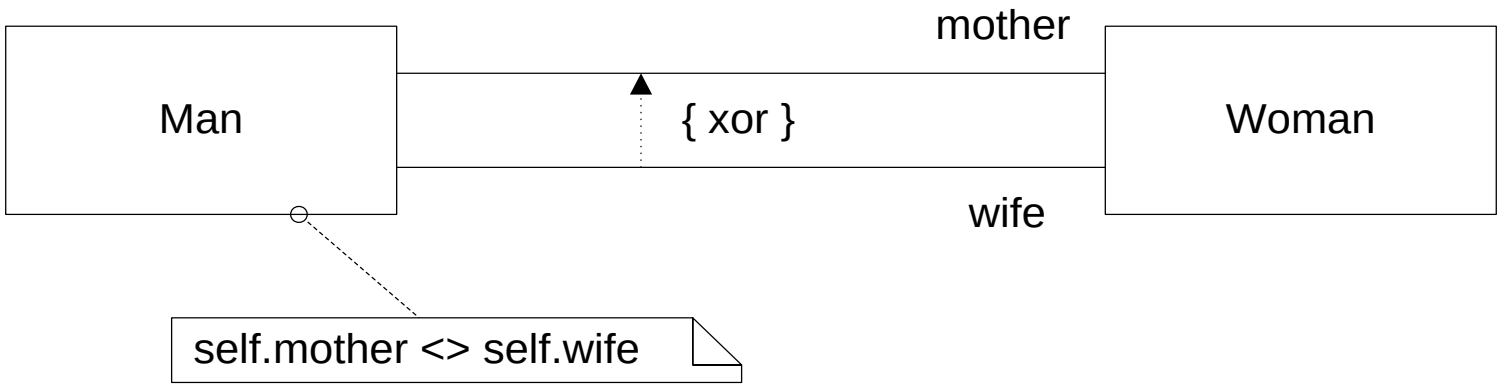
Examples:

```
{Author = (Dave,Ron)}
{Version Number = 3.1}
{Location = d:\uml\examples}
{Location = Node:Middle Tier}
```



Object Constraint Language

- Used to model business rule semantics and to make unambiguous assertions (with no side effects).
- Constraints make models more precise.
- OCL is used below to model the invariant on Man.
- Note: this is redundant with the **{ xor }** constraint on the relationships.
- OCL has an easy syntax.
- Google ‘OCL’ to find out more.



OCL Examples

- Invariant:

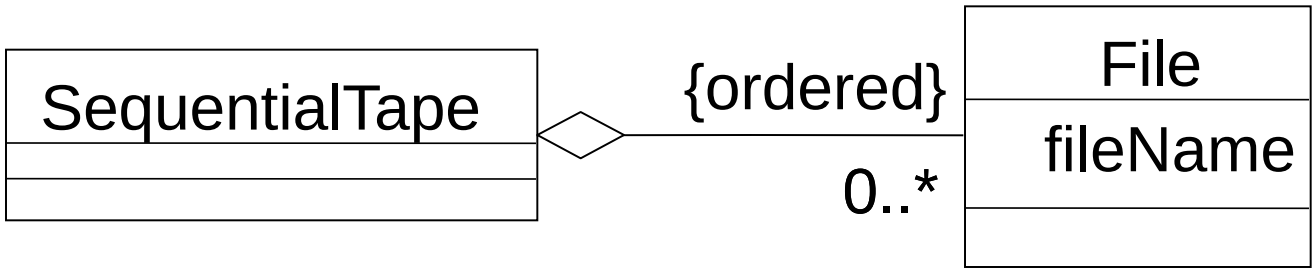
```
context m : Man
inv: m.mother <> m.wife
```

- Pre & post condition:

```
context Factory::processOrder( o : Order ) : boolean
pre orderPaidFor: o.outstandingBalance <= 0
post orderProcessed: processed = true
```

Ordering

- **`{ordered}`** is an example of a common ***constraint***.



Interfaces

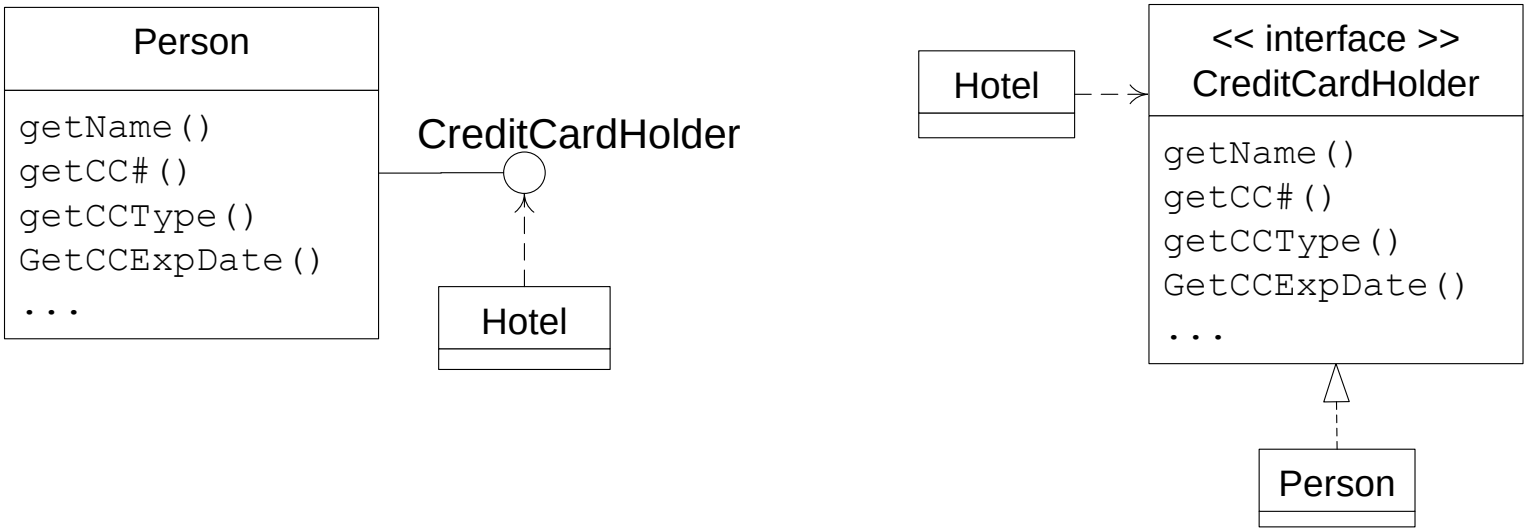
The client object sends a message to an object with a known interface; any class that implements the given interface will do.

Example: Class `Person` can implement a `CreditCardInfo` interface, used by an airline reservation program.

- The program doesn't know or care about `Person`, only about objects that implement the `CreditCardInfo` interface.
- There might also be a `Corporation` class that implements the `CreditCardInfo` interface.
- The `Person` class can change dramatically without the reservation program having to be changed at all.

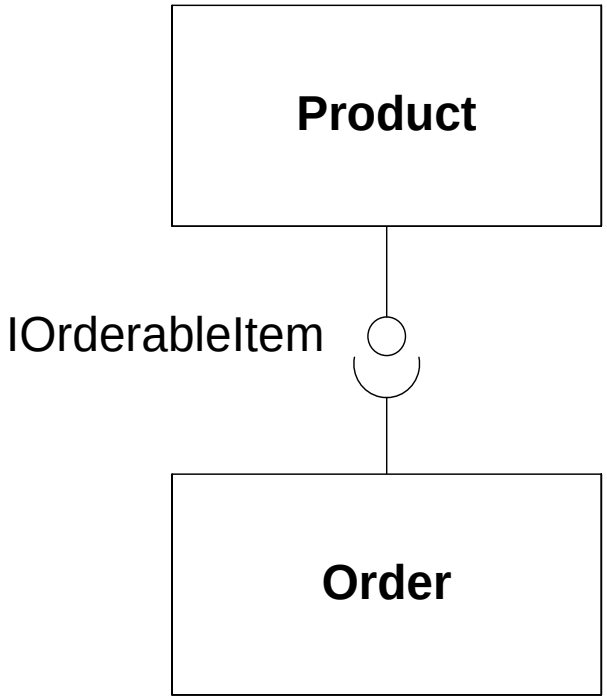
Interfaces

- Java interfaces may define ***no implementation***.
- C++ interfaces are built with ***purely abstract*** classes.
- The use of this so-called ***lollipop notation*** is optional.



Interfaces and Sockets

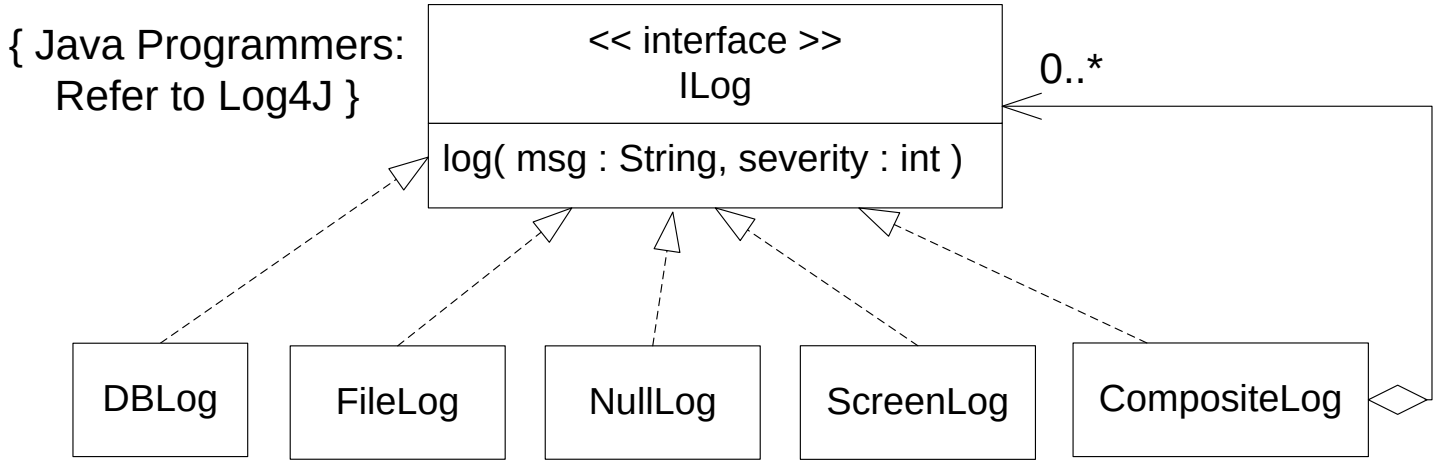
- Product is-a-kind-of IOrderableItem (it implements the interface).
- Order requires an IOrderableItem (this is called a ***socket***).



Interface Example

Interfaces may be represented...

- Using the “lollipop” notation, as in the previous slide.
- As a class adorned with the <<interface>> stereotype.
- By naming convention, IWhatever.
- Note the dotted line on the inheritance relationship.



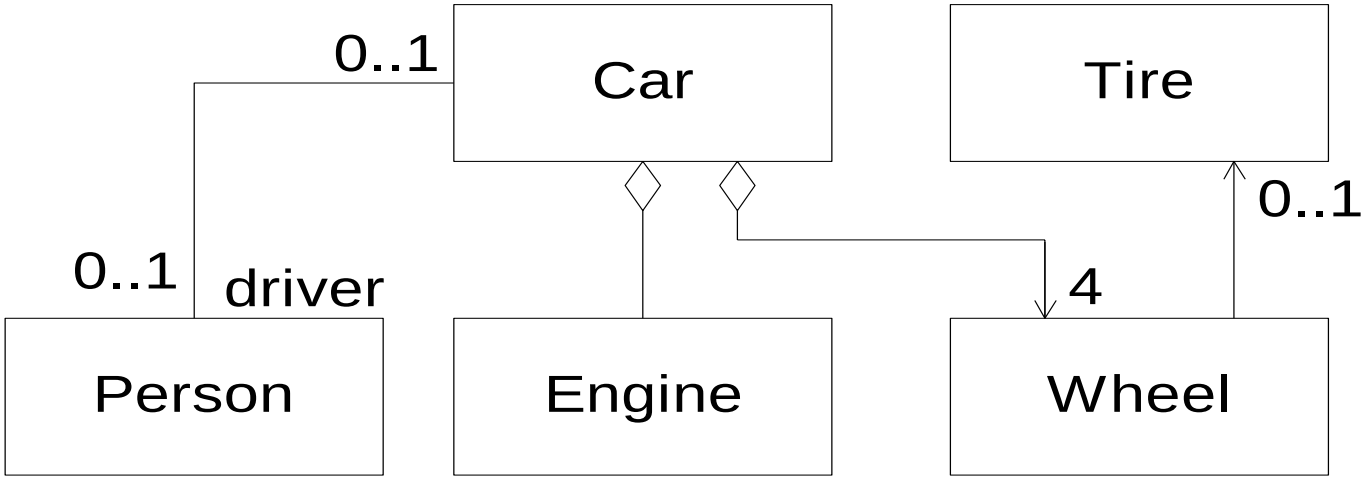
Interface Example

- You decide that your Video Store system could be used to manage other businesses that rent things (e.g., ski shops & libraries).
- You make a new abstract class called RentableObject with rent() and return() methods.
- Make Video extend RentableObject.
- This sounds easy, but… core system classes like Video may already be in a different inheritance hierarchy**.**
- In C++ you can use multiple implementation inheritance, but in Java you can't.
- Instead, create an interface: IRentableObject.

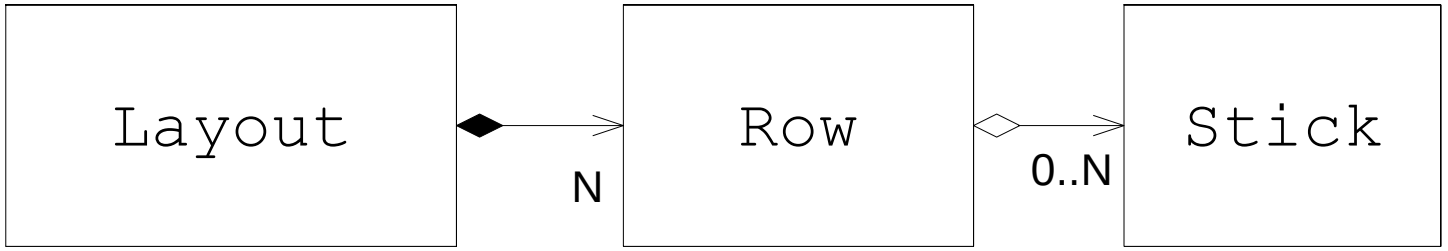
Composition / Aggregation

The diamond symbol can represent more than one concept:

- Part / whole relationships (most common)
- Has - a
- Has - a - collection - of
- Is - composed - of



Composition & Aggregation



Composition:

- UML blackens the ***composition*** diamond.
- The hollow diamond is used for ***aggregation***.
- Composition is a stronger association than aggregation. The difference is that with composition, the part never has more than one whole, and the part and the whole always have a shared lifetime.

Composition, Aggregation, & Associations

Composition:

- A book is composed of its pages and cover.

Aggregation:

- A bookshelf holds a collection of books that changes over time.

Association:

- A book has an associated author.

Dependency:

- A person reads a book, then gives it to a friend.

Composition & Associations Example

```
class Person {
  private Life        vida   = null;                    // Composition
  private Array<Cell> cells  = new ArrayList<Cell>();    // Aggregation
  private Person      mother = null;                     // Association
  private Person      father = null;
  public Person( Person mom, Person dad ) {
    mother = mom;
    father = dad;
    vida = new Life();
    cells.add( new Cell( this, mom, dad ) );
  }
  public void read( Book b ) { b.read(); }               // Dependency
}
```

- N.B: This slide is not intended to provide commentary on religion.

Association Semantics

For Composition / Aggregation:

- Can the containee be contained within more than one container?
- Are the lifetimes of the two objects exactly the same?
- Does one object own/control the other's memory?
- Can the association be labeled ***part of*** or ***composed of***?
- Or would it be better labeled ***collection of***?

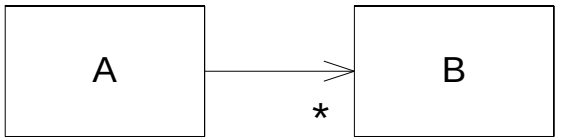
For Associations / Dependencies:

- Is the association transient, permanent, or somewhere in between?

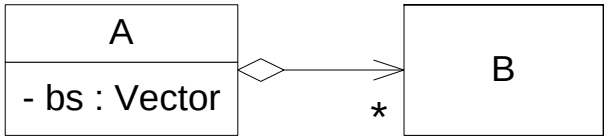
Sometimes these distinctions are not black and white.

Consider the memory management implications (especially in C++).

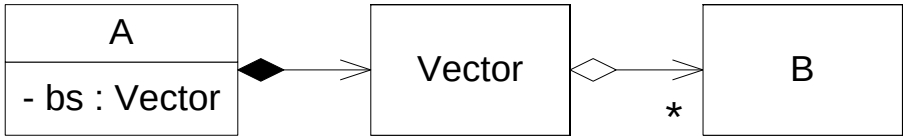
Level of Detail



Manager / Client / Analysis / High-level Design



Programmer / Detailed Design



Pedantic / CASE Tool

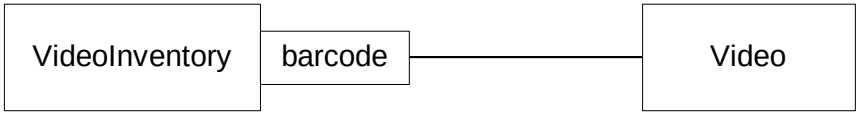
- The level of detail depends on the audience.
- Notice that ***Collection Classes*** (such as Vector) are usually not shown.

Qualified Associations

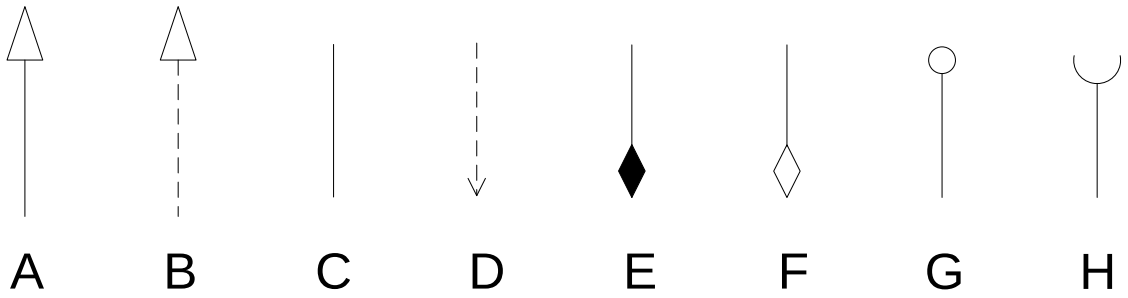
- Qualified associations are implemented with a Dictionary / HashTable / Map.
- The unqualified model can be read, “The VideoInventory has a collection of zero or more Videos.”
- The qualified model can be read, “The VideoInventory, given a barcode, uniquely references a Video.”

Unqualified:

Qualified:



UML Association Review



A) Implementation Inheritance (Generalization)

B) Interface Inheritance (Realization)

C) Bidirectional Association

D) Unidirectional Dependency

E) Composition

F) Aggregation

G) Provided Interface (Lollypop)

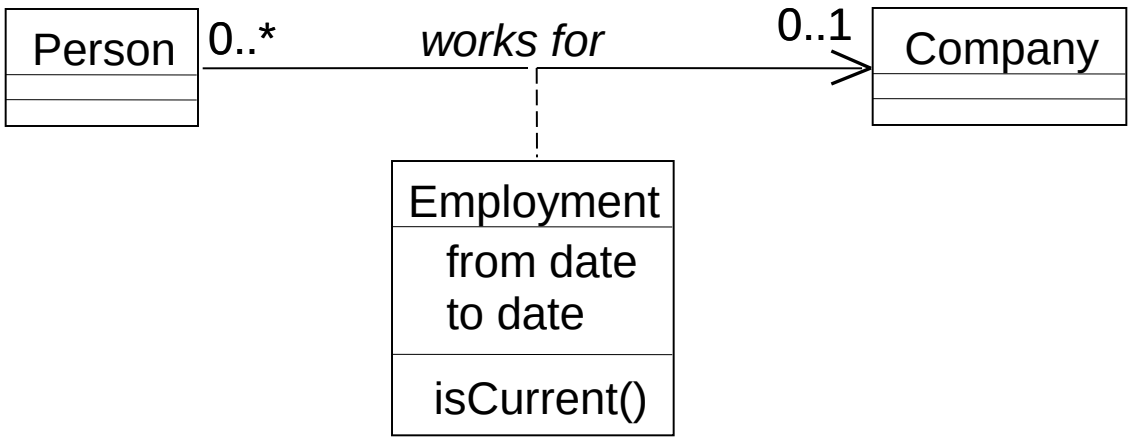
H) Required Interface (Socket)

Association Attributes

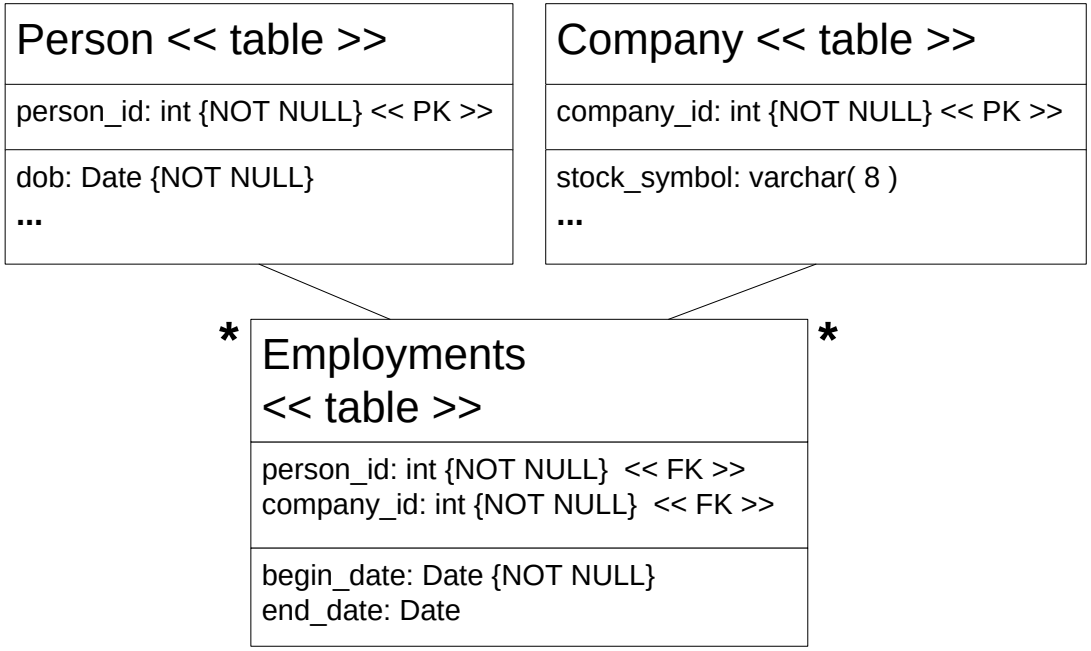
Attributes sometimes depend on two objects.

- Complex attributes may be modeled as a class.

For every Person / Company pair, there is one Employment instance, an attribute of the ***works for*** association.



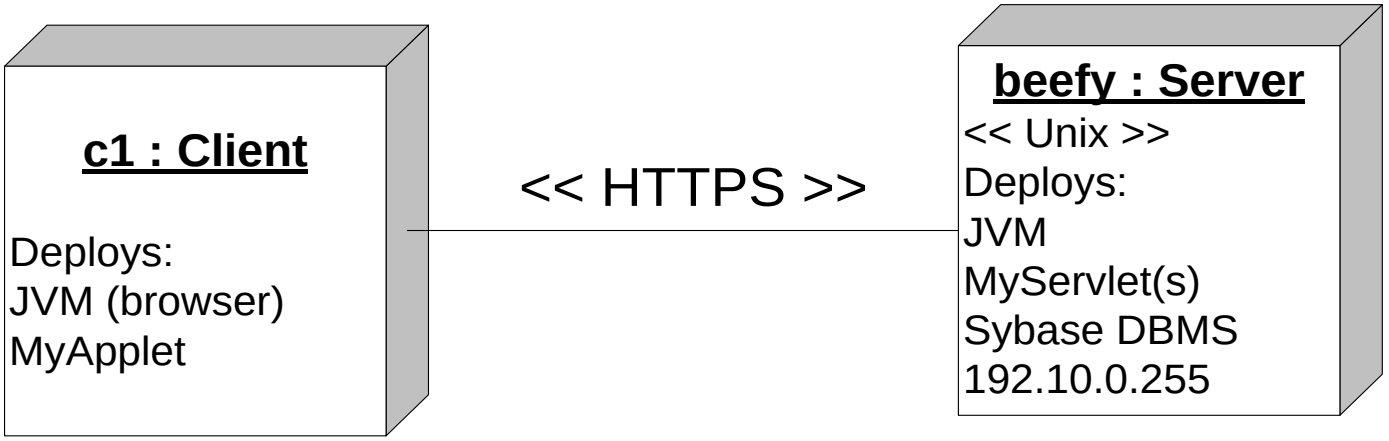
Mapping Many-to-Many Associations



- Whenever two classes have a many-many relationship, a ***relational*** database requires a third ***table*** to represent the mapping.
- More on this in section XVIII.

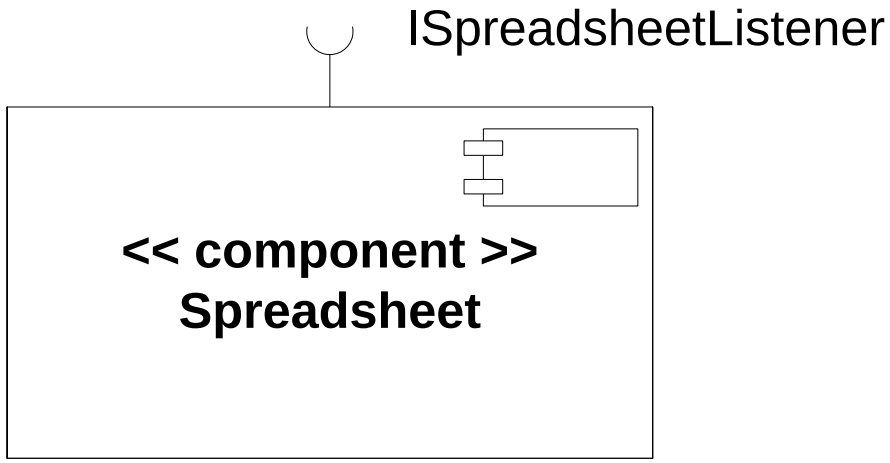
Deployment Diagrams

- ***Nodes*** represent the system hardware.
- ***Components*** represent software things.
- Components are deployed on Nodes.
- An association between 2+ Nodes is a ***Connection***.

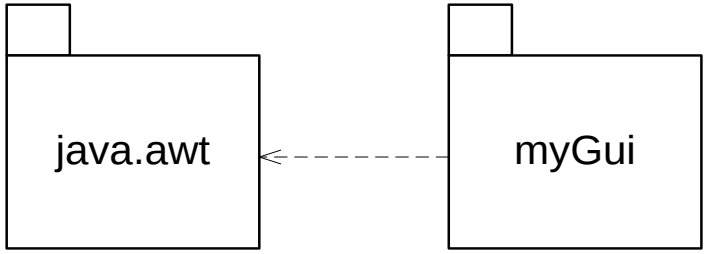


Components

Components should be designed to be reused, with high cohesion, disciplined encapsulation, and dependencies only on external interfaces.

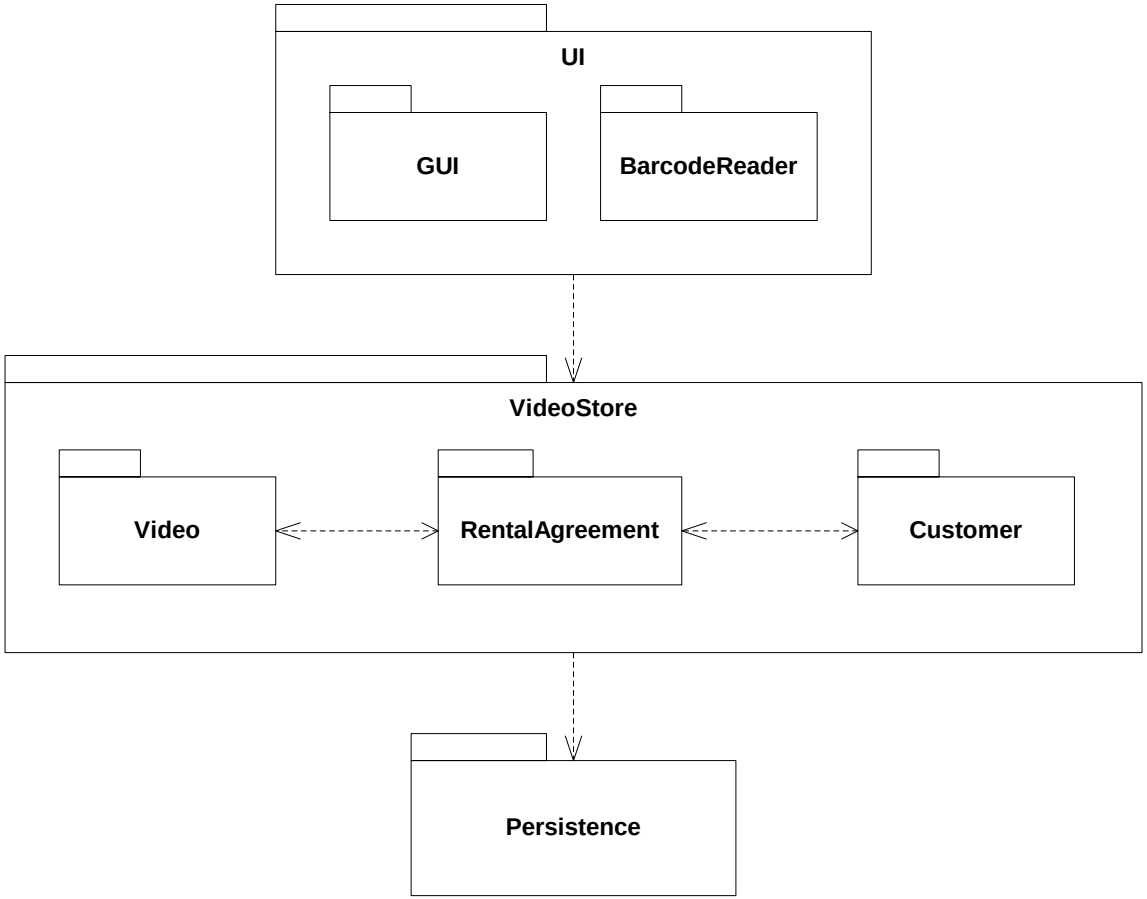


Packages

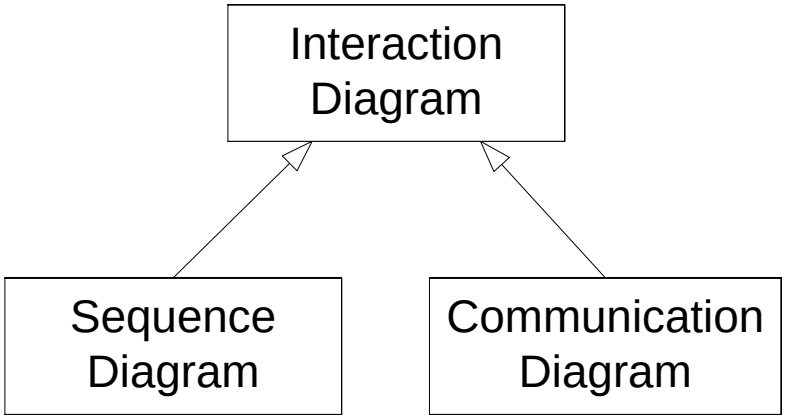


- A ***package*** is a way to organize code into semantically related groups.
- Packages can be nested.
- At the highest level, a package contains an architectural entity (e.g., business domain or subsystem). Or a package may represent a single person's work.
- Java packages (like C# ***namespaces***) solve class naming problems.
- For example, both `java.awt` & `myGui` have a class called `Event`. Use `java.awt.Event` to disambiguate.
- This diagram is useful for visualizing ***dependencies***.

Package Example



Interaction Diagrams

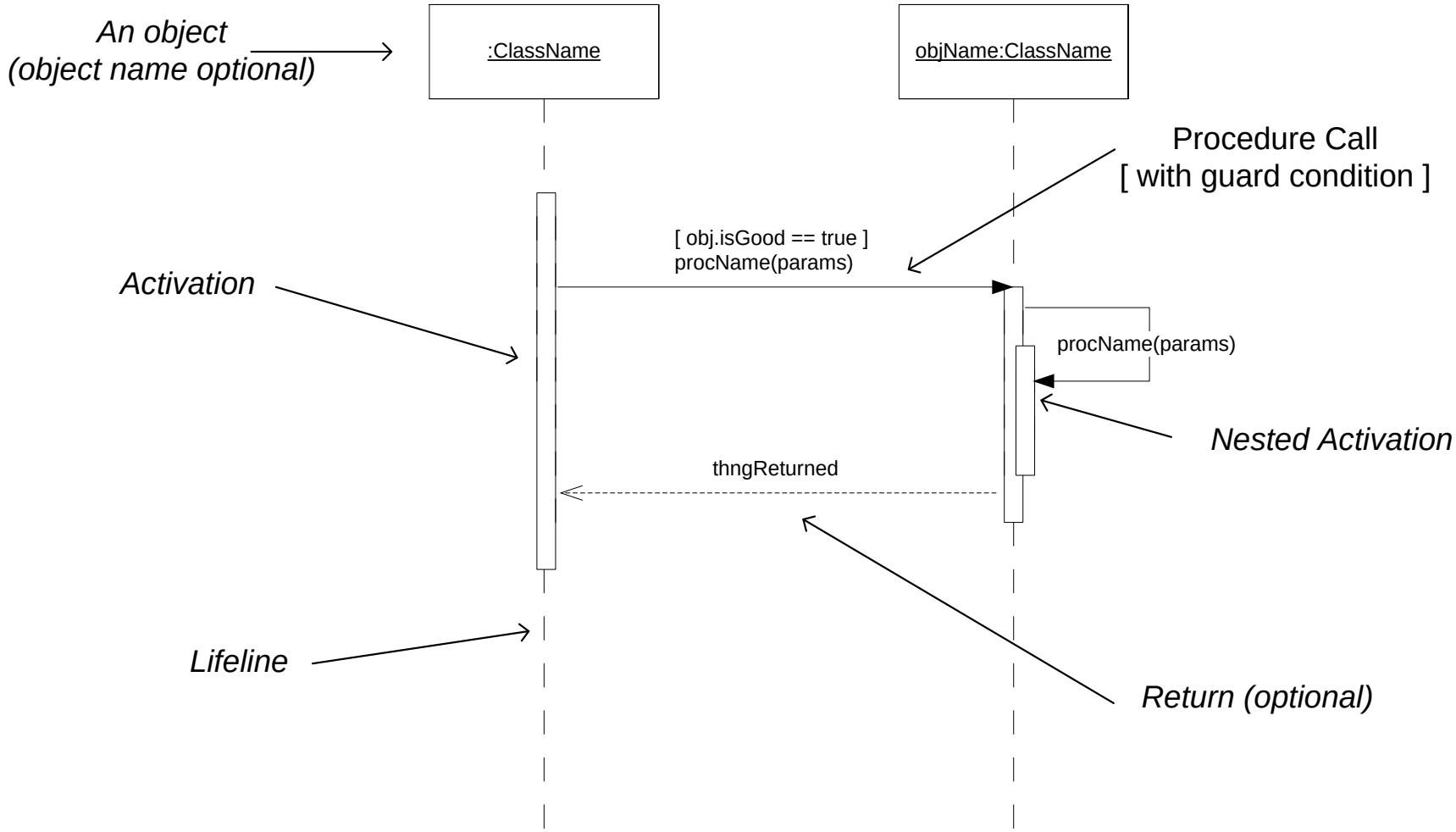


- Communication Diagrams (formerly known as Collaboration Diagrams) are roughly equivalent to Sequence Diagrams semantically; they are just laid out differently, with Sequence Diagrams placing more emphasis on the time-flow aspect of the situation.

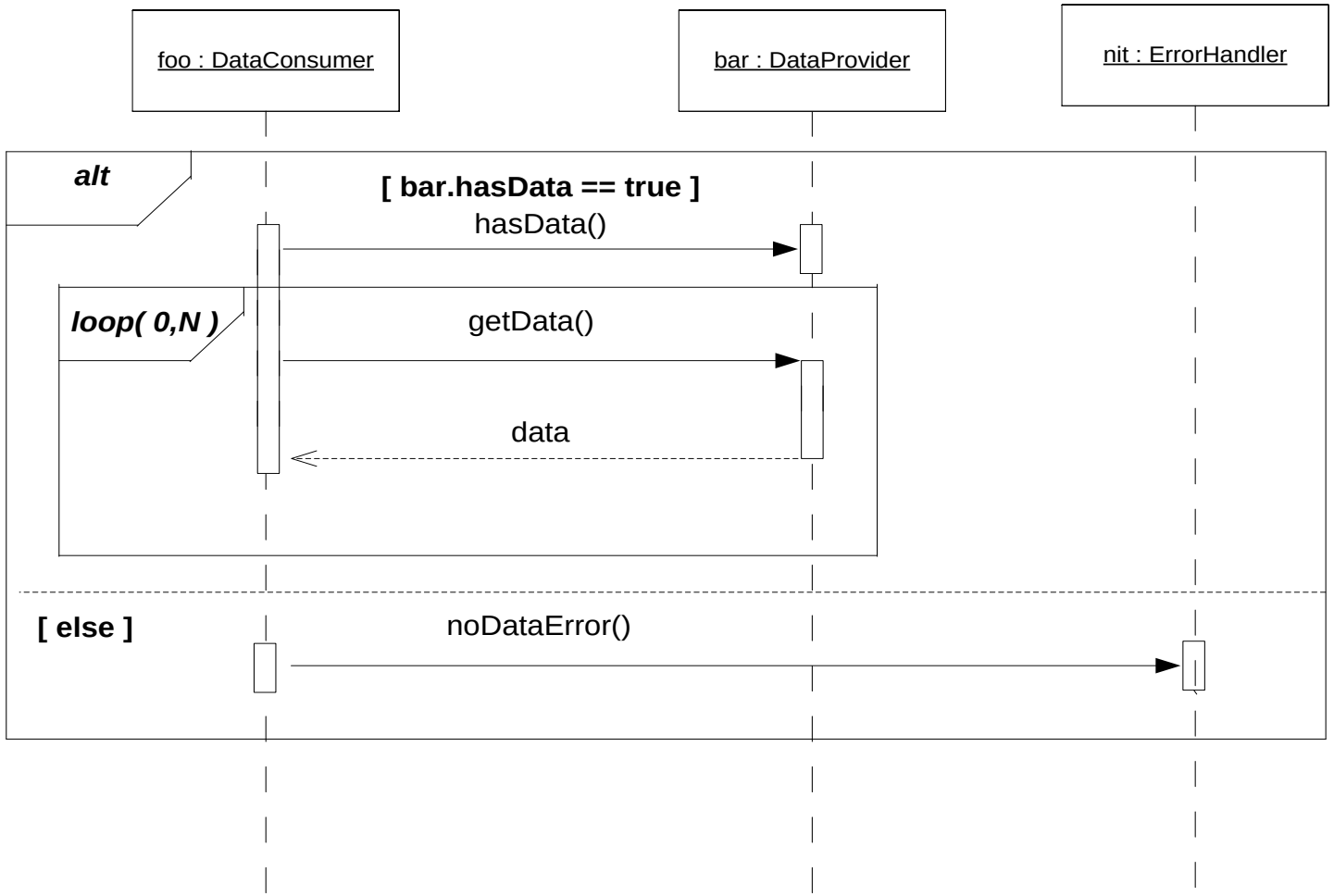
Sequence Diagram

- Shows the object collaborations over time for one scenario.
- Useful for understanding use cases.
- Useful for determining which object and classes should have which responsibilities.
- Start drawing these diagrams as soon as you have candidate classes, and before you spend too much time refining them.
- Can get messy when there is more than one thread of control within the scenario (if..else, looping). Simplify, don't clutter.

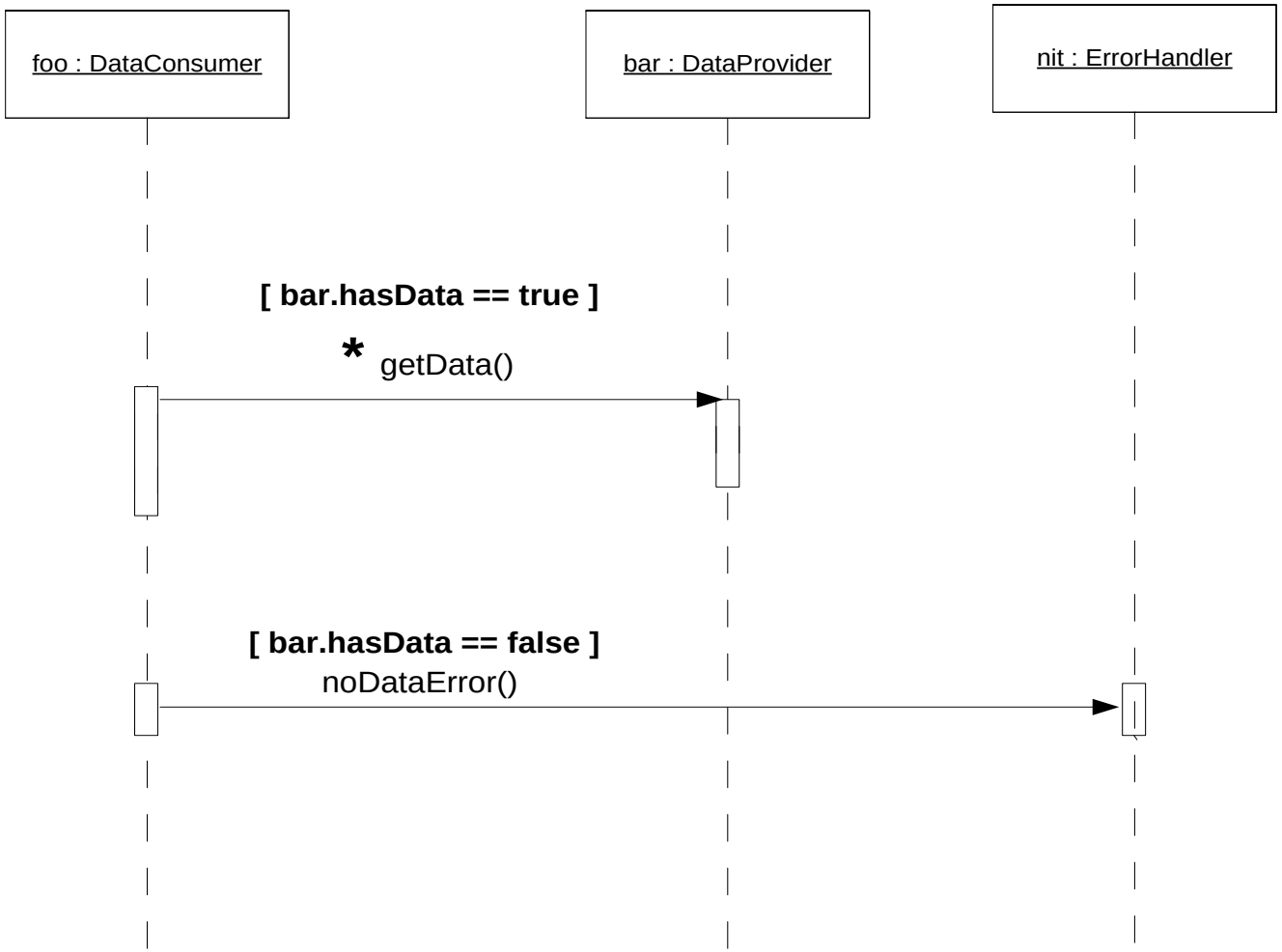
Sequence Diagram Notation



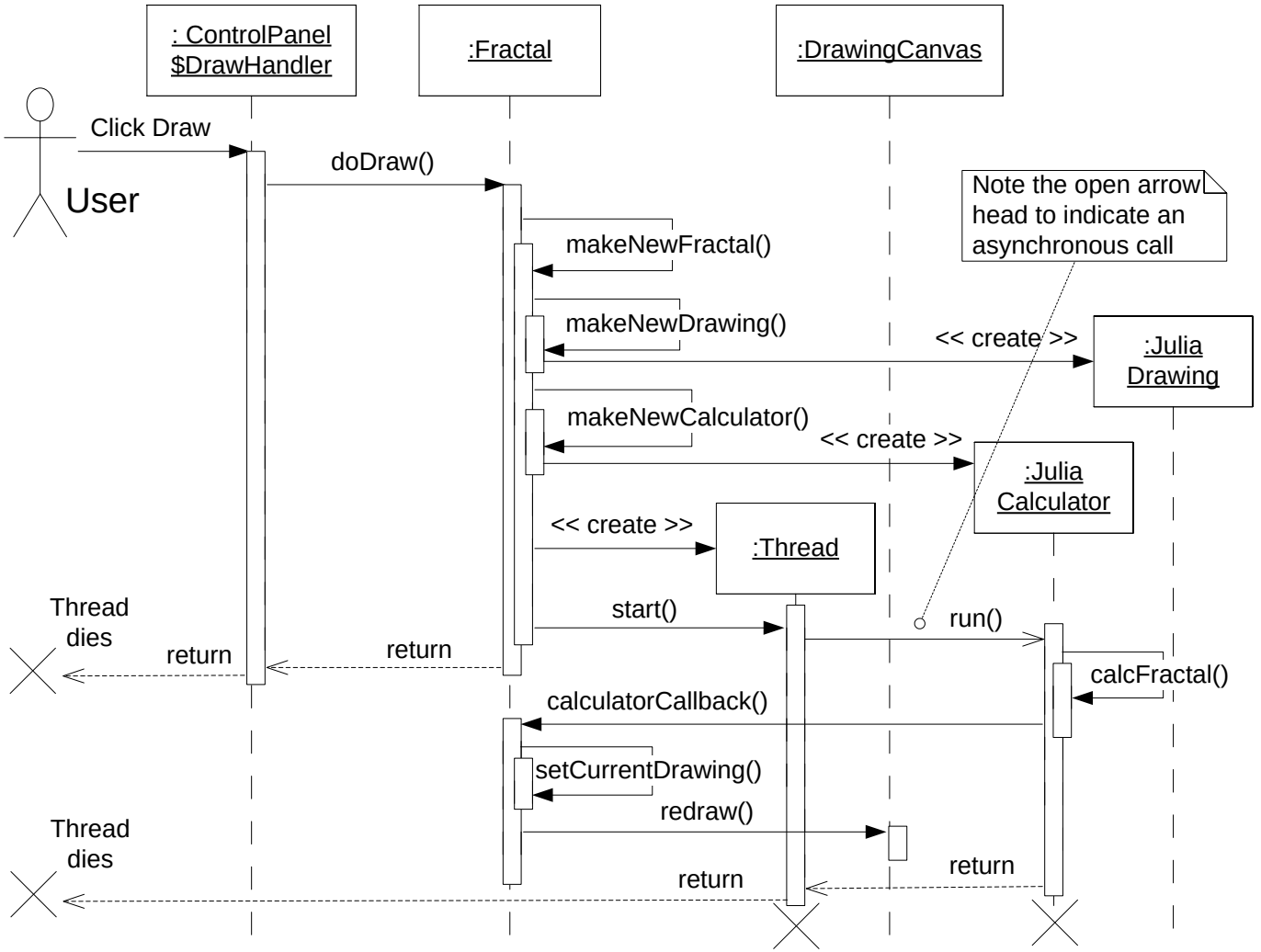
Another Sequence Diagram Example



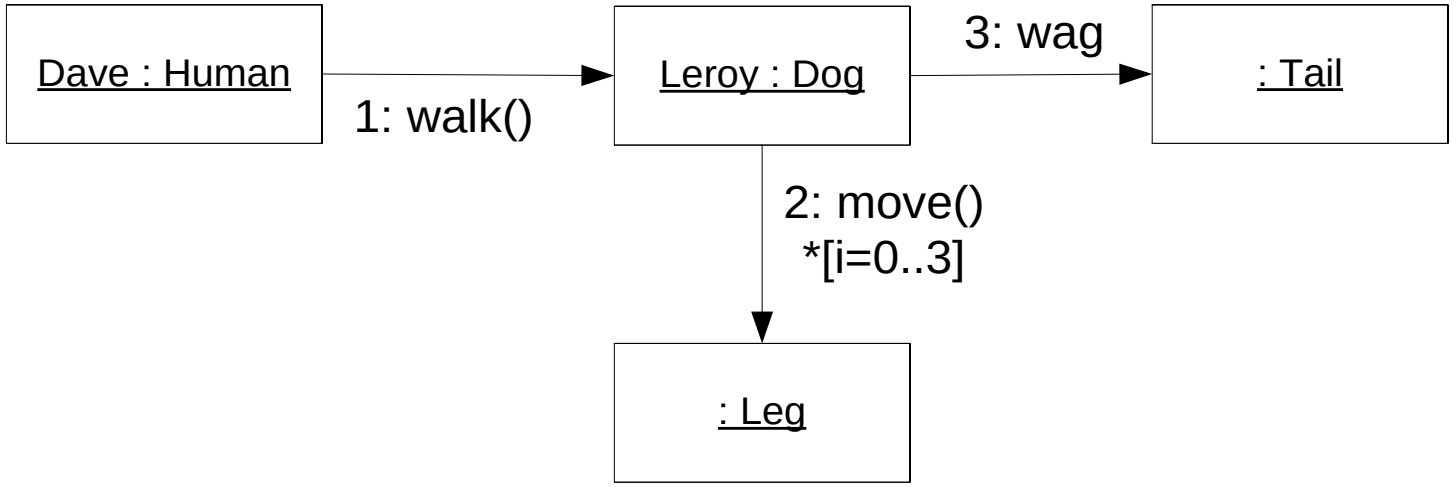
Another Whiteboard-Friendly Alternative



Example Sequence Diagram

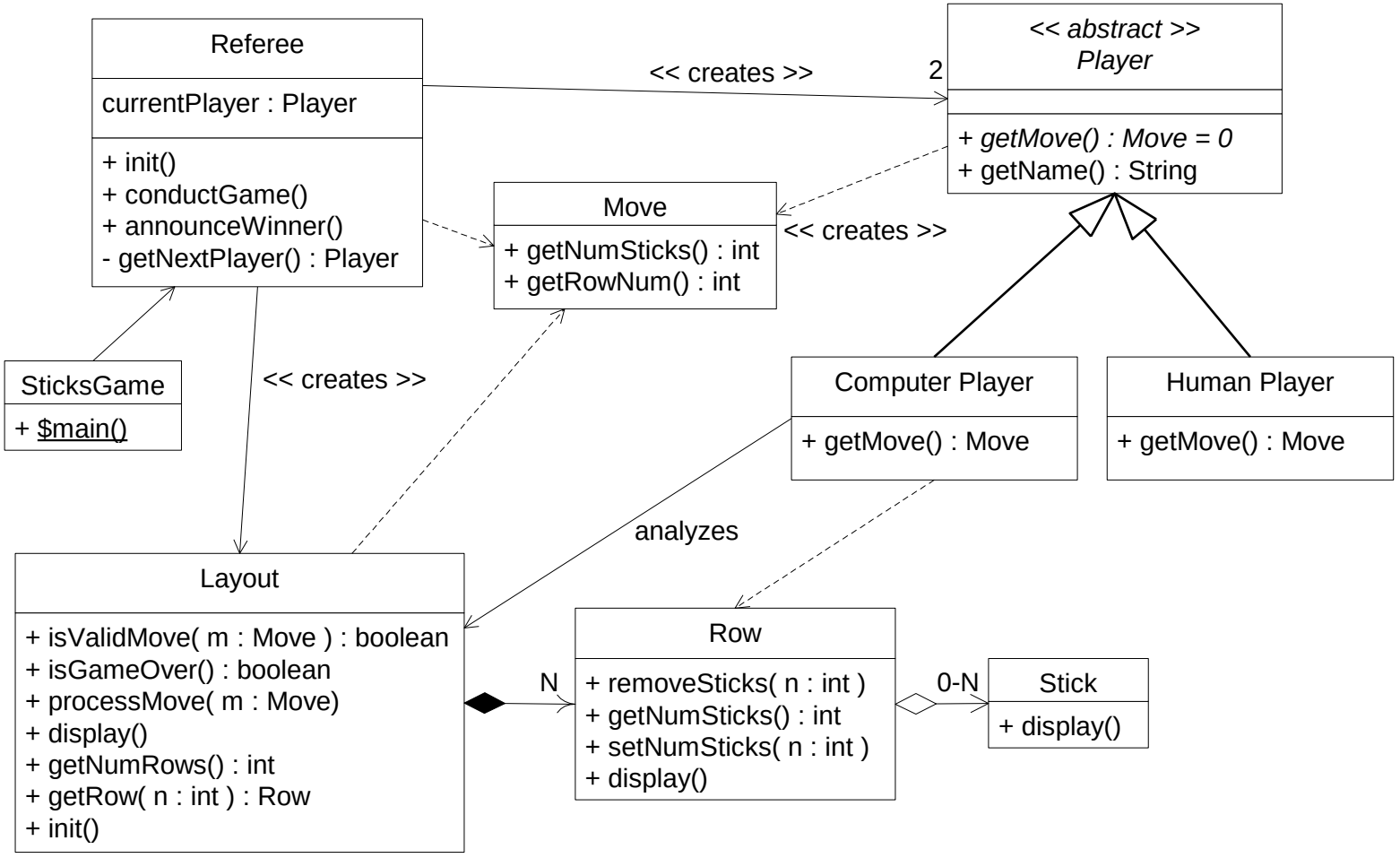


Communication Diagram

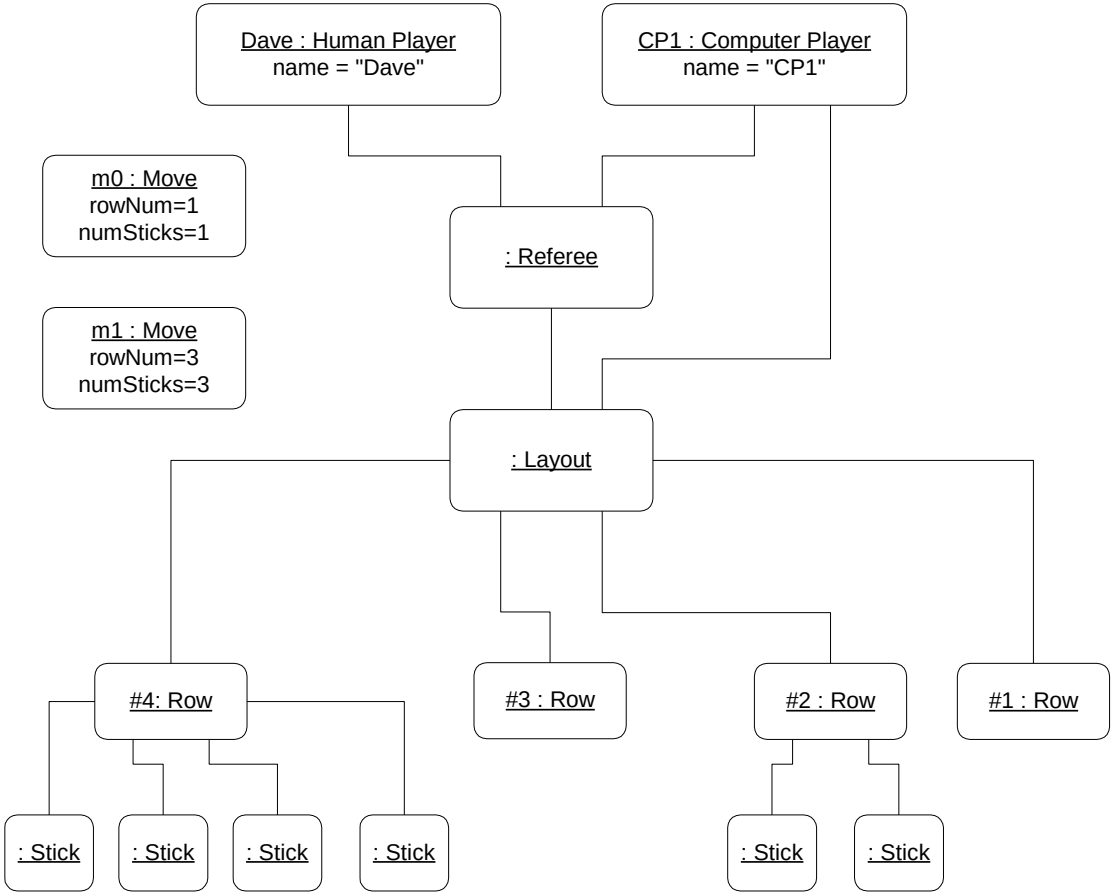


- Objects communicate, not classes.
- *[i=0..3] is UML standard syntax for a looping constraint.
- An alternative model could have the 4 leg objects shown with sequence numbers 2a, 2b, 2c, and 2d.

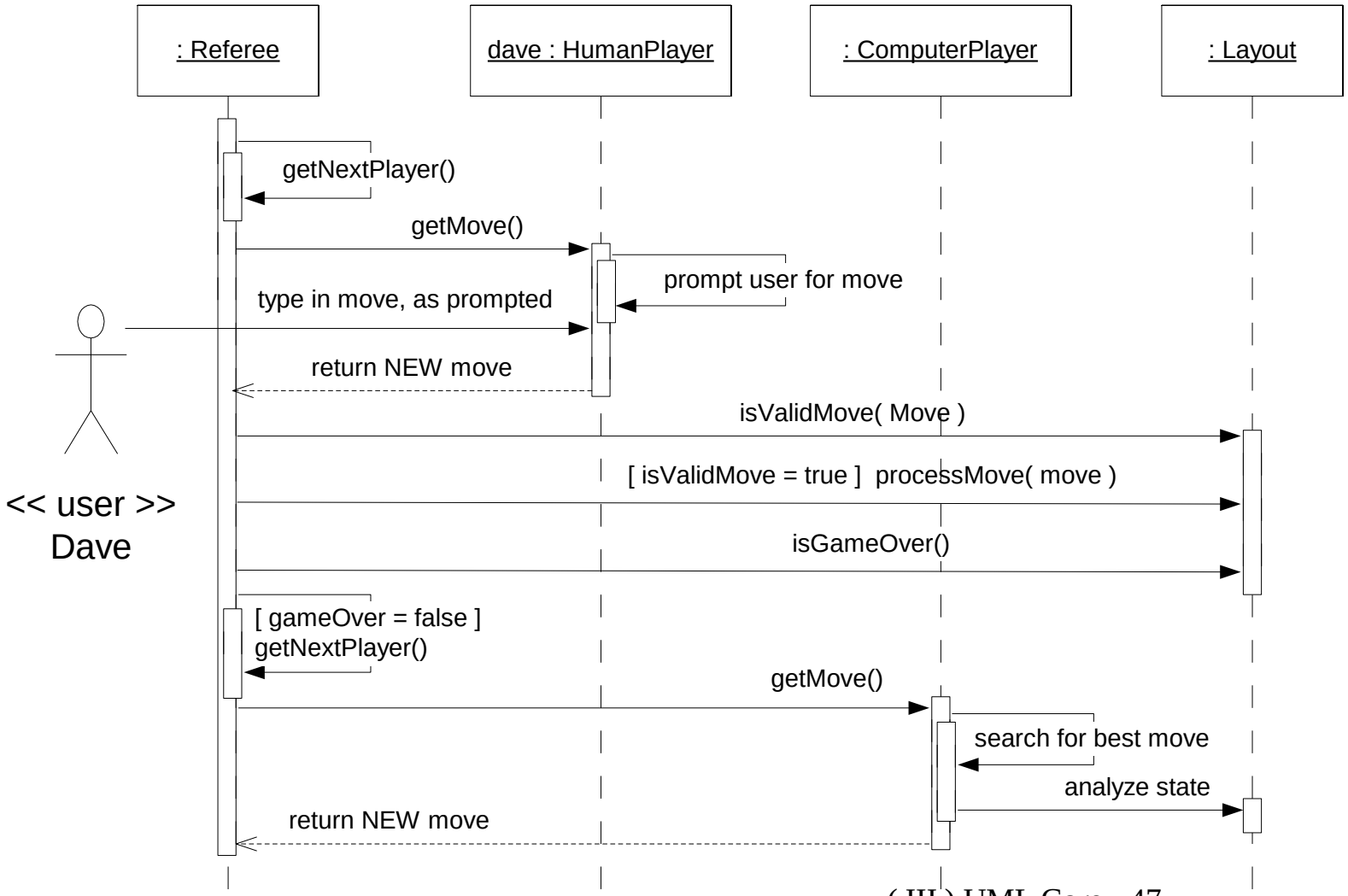
Example: Sticks Game Class Diagram



Example: Sticks Game Objects



Example: Sticks Game Sequence Diagram



Example: Sticks Game Java

- Refer to **sticksgame.zip** (complete source code) & **minimax.pdf** (design of the computer player's search algorithm) on the course web site.

```
package oop.sticks; // File: oop/sticks/SticksGame.java
public class SticksGame {
  public static void main( String[] args ) {
    try {
      Referee ref = new Referee();
      ref.init( args );
      ref.conductGame();
      ref.announceWinner();
    }
    catch( Throwable t ) {
      t.printStackTrace();
} } }
```

Sticks Game Java – Getting Started

- The Sticks Game can be prototyped as a ***console program*** with no graphics and a dumb ComputerPlayer. Then iterate, refactoring and adding features such as a Minimax computer player and a ***GUI***.

```
package oop.sticks;
public class SticksTest {
  public static void main( String[] args ) {
    try {
      Row row = new Row( 3 ); // This is easy to code!
      row.display(); // Next: create a Layout...
    } // Then, ask the Layout if the game is over...
    catch( Throwable t ) {
      t.printStackTrace(); // Improve exception handling
} } }
```

Not all useful diagrams use UML