

# Object - Oriented Programming & Design

## Part III: UML Core Diagrams

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

### **The Unified Modeling Language (UML)**

---

- The most widely used notation system for all aspects of OO development.
- *Not* a development method.
- Captures most aspects of a system.
- Automated support tools available.
- Has been accepted as a modeling standard by the OMG (Object Management Group).

UML documentation is available on the net:

- <http://www.rational.com/uml>
- <http://www.omg.org/>

The notation used in the *Design Patterns* book is adapted from the Object Modeling Technique (OMT) notation.

- The OMT notation is very similar to UML.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 2

## Modeling

---

Successful OO designs almost always begin with a visual “object model” of the problem domain, involving both the domain experts and software designers alike.

- If it is complicated and/or it needs to be understood by many people, make a model.
- A picture is worth a thousand words (1000 lines of code?)
- Not just for software.
- You don’t need to be a programmer to understand an object model.
- Modeling enhances everyone’s ability to understand the software, saving time and money.
- Models are used to visualize, specify and document a design, and they can even be used by some commercial tools to generate code (if the models are precise, unambiguous and complete).

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 3

## Modeling (cont.)

---

- Understand the domain.
- Understand the purpose of each model, its audience, the appropriate level of detail, and what information is therefore important or relevant.
- Model with interfaces before implementation.
- Anticipate implementation and maintenance difficulties.
- Discuss models in small groups, with a white board.
- Use various types of models, not just class diagrams.
- The domain vocabulary used by the designers should not differ from that of the users, business analysts and/or the documentation.
- Encourage your own internal complexity alarm to alert you to poor design. If it is neither clear, simple, nor intuitively satisfying, it can likely be designed better.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 4

## Modeling (cont.)

---

- Minimize inter-class dependencies.
- Plan for future extensions.
- Use artistic license, adding, deleting and *refactoring* classes as you see fit.
- Iterate, iterate, iterate...
- Foresight is not 20/20; that is why *iterative* approaches almost always result in higher quality designs.
- Capture essential abstractions and pertinent detail.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 5

## Static and Dynamic Models

---

- Static models describe things without regard to time.
- Dynamic models describe how things progress through time.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 6

## UML Models

		Analysis	Design	Implementation
Static	Use case	x		
	Object	x	x	
	Class	x	x	<b>x</b>
	Component		x	x
	Deployment			x
Dynamic	Interaction	x	x	<b>x</b>
	Activity	x		
	State	x	x	<b>x</b>

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 7

## There are many kinds of diagrams:

**Class** - What is the static structure of the system? (N.B. This is very often and incorrectly called an *object* model).

**Instance** - One instantiation of a class model (technically an object model).

**State** - One per class, showing when it can change state.

**Activity** - In what order must activities be sequenced (like a flowchart)?

**Use Case** - What does the system have to do, from the users' perspective?

**Sequence** - What are the patterns of object collaborations (per use case)?

**Functional** - How do data and storage get transformed?

**Deployment** - What software runs on what hardware (nodes), and how do these nodes communicate?

**Package** - What are the dependencies between implementation packages?

**User Interface** - What are the relationships between user interface screens?

**Data Model** - A.K.A. Entity-Relationship Diagrams (ERDs).

**And more...** Not all of these are part of the UML standard.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 8

## Detailed Modeling with the UML

- The Unified Modeling Language has very rich notational syntax and semantics. We will not cover it all. Nor should you feel compelled to use it all. UML models may be used for \*detailed\* specification of code, but often the code itself (with good comments) makes better \*detailed\* documentation. Use an appropriate (pertinent) level of detail in your models, depending on the target audience and intended use.

### CSCI-4448 Students:

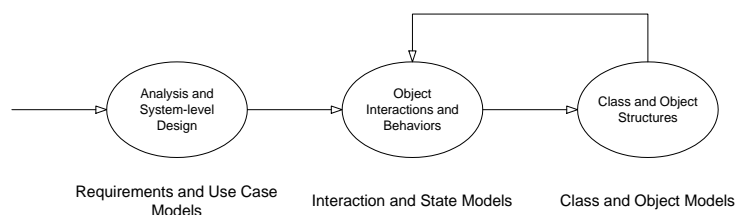
- You should learn at least the UML syntax and semantics contained in these lecture notes (all chapters).
- For the purposes of this course (ie: tests) if it isn't in these notes, you don't have to learn it.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 9

## General Flow of Modeling and Development



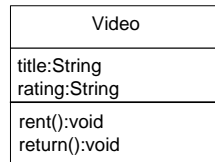
- Note: this is a very simplified view of the world. There is iteration and refactoring of the requirements, design, and code...

Copyright © 1996 - 2008

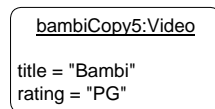
David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 10

## Classes & Objects



A Class



An Object (Instance)

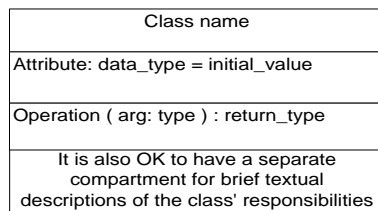
- Note: The UML uses rectangular boxes for both objects and classes; we will use rounded corners on objects to help visually distinguish between the two. This approach is more whiteboard friendly.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 11

## Class Adornments



You can put a description of the class, its reason for being, or other cogent information in a note box like this

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

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 12

## Class Adornments (cont.)

### Class Name

Normal font = concrete class (no adornment needed)

*Italic font* or << *abstract* >> = abstract class

(*italics* are not whiteboard friendly - use the *stereotype* notation)

### Attribute and Method Scope

Normal font = instance scope (no adornment needed)

Underlined or **\$** = class scope (\$ is not standard UML)

For abstract methods, use =**0** (not standard UML)

### Attribute and Method Visibility (*degree of encapsulation*)

+ public

- private

# protected

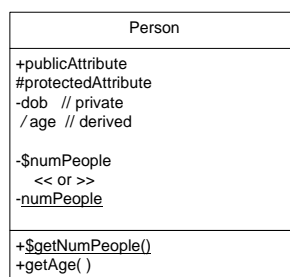
% (Java's package visibility - not standard UML)

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 13

## Class Adornments (cont.)



- The age attribute is *derived*.
- The number of instances of class Person, numPeople, is an attribute of the class Person itself, and not of any one instance of the class. This is called a *class* (“static”) member variable; it is like a global variable for the class. Some people use \$ as an alternate notation for class attributes and behaviors.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 14

## Class Adornments (cont.)

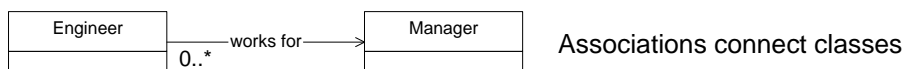
```
public class Person // Java
{
    public Person( Date dob ) { // constructor
        this.dob = dob;
        numPeople++;
    }
    public void finalize() { // not exactly a destructor
        numPeople--;
    }
    public int getAge() {
        return TimeSevices.diffInYears( TimeServices.today(), dob );
    }
    public static int getNumPeople() {
        return numPeople;
    }
    private Date dob;
    private static int numPeople = 0;
}
```

Copyright © 1996 - 2008

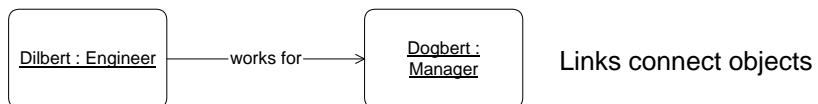
David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 15

## Links and Associations



Associations connect classes



Links connect objects

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 16

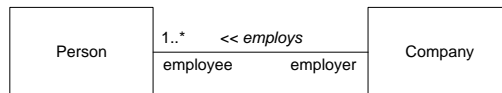
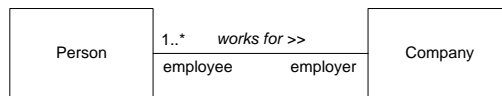
## Roles and Association Names

- **Role**

- Name instances of a class as they appear to the class at the other end of the association. Usually a noun.

- **Association name**

- Names the association itself, and may require an arrow to specify the direction of the association. Usually a verb or verb phrase.



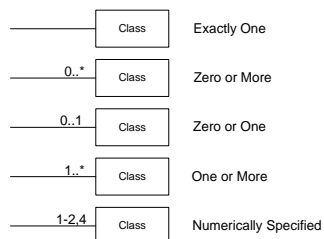
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 17

## Multiplicity (Cardinality)

- Class association adornments, to specify constraints on the number of instances (objects) on either end of the association.
- Not needed for links in instance diagrams. Why not?



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

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 18

## Stereotypes

A *Stereotype* is a conventional categorization of modeling entities.

- They are often applied to classes, associations, and methods.
- They provide 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.

Two ways of designating:

- Use a normal UML element, with your stereotype name written between guillemets.
- Use your own, unique icon.

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

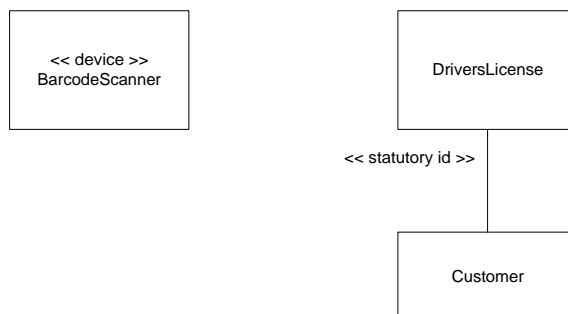
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 19

## Stereotypes (cont.)

- Modelers are free to invent their own stereotypes



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

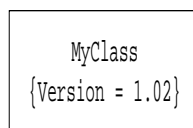
( III ) UML Core Diagrams - 20

## Tagged Values

- A *Tagged Value* is another UML extensibility mechanism, allowing you to add new {name = value} property specifications to your model.

Common examples of tagged values are:

- {Author = (Dave,Ron)}
- {Version Number = 3}
- {Location = d:\java\uml\examples}
- {Location = Node: Middle Tier}



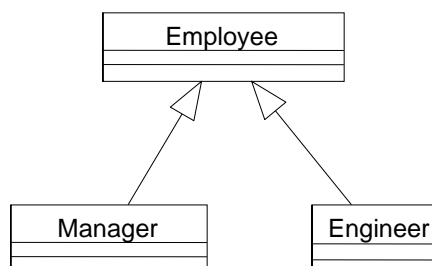
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 21

## Generalization, Specialization & Inheritance

- Employee *is a generalization of* Engineers and Managers.
- Engineer *is a specialization of* Employee.
- Manager *is a kind of* Employee.
- Engineers and Managers *inherit* the Employee interface (and in this case, some implementation too).



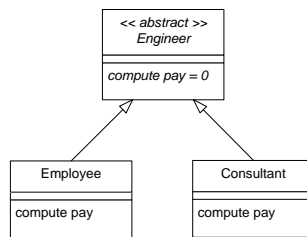
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 22

## Abstract Classes

- A generalization, with a mechanism for specifying specializations.
- Indicated in UML with the << abstract >> stereotype.
- Indicated in C++ by declaring all “pure virtual” methods to be = 0.
- Indicated in Java with the abstract keyword.
- An *interface* is like an abstract class, but with no implementation whatsoever.



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 23

## Interfaces

For polymorphism to work in a typed OO language like C++ or Java, the client object sends a message to an object with a known interface; any class that implements the given interface will do.

- An interface is the most basic declaration you can make about an object.
- Use of interfaces promotes code flexibility.

**Example:** A Person class can implement a CreditCardInfo interface, used by an airline reservation program.

- The program doesn't know or care about the Person per se, only that (s)he can implement the CreditCardInfo interface.
- There might also be a Corporation class that implements the CreditCardInfo interface; the rest of the program wouldn't know or care.
- The Person class can change dramatically without the reservation program having to be changed at all.

Copyright © 1996 - 2008

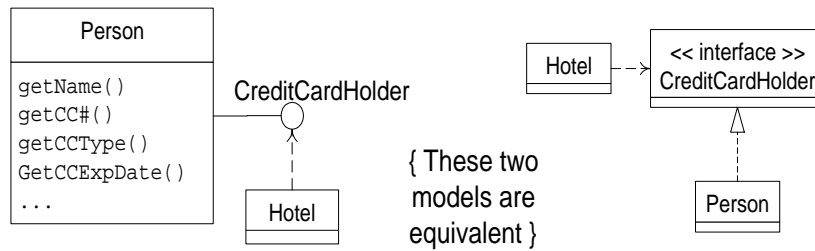
David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 24

# Interfaces

- In Java, interfaces may not define any implementation.
- In C++, interfaces may be built with purely abstract classes.

The use of this so-called “lollipop” notation is optional.



Copyright © 1996 - 2008

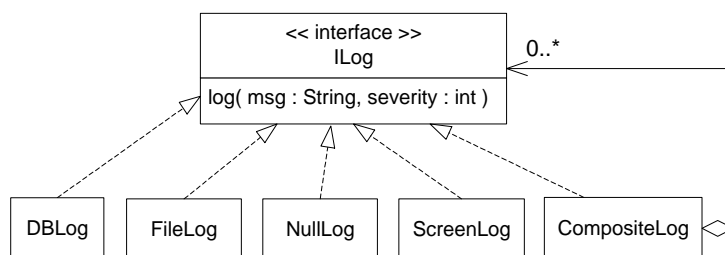
David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 25

# Interfaces (cont.)

Interfaces may be represented...

- using the “lollipop” notation, as in the previous slide.
- as a class adorned with the <<interface>> stereotype, accompanied by a dotted line on the inheritance relationship.



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 26

## Interface Example

- Suppose that, after having written your video store system, you decide that your core system could be used to manage other businesses that rent things out. (E.g., ski shops, libraries, equipment rental stores, etc.)
- You could abstract from the Video class a RentableObject abstract class, with rent() and return() methods. This would be somewhat problematic though, because you would want other applications to reuse your core system classes for things that may already be in a different inheritance hierarchy.
- In C++, you could use multiple implementation inheritance, but in Java you can't. Instead, you create an interface.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

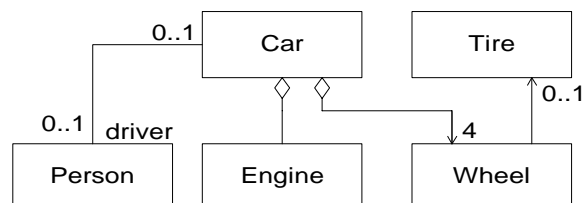
( III ) UML Core Diagrams - 27

## 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

Note how time influences the cardinalities: the car could have many drivers over time, but at any given time, it can have at most one.

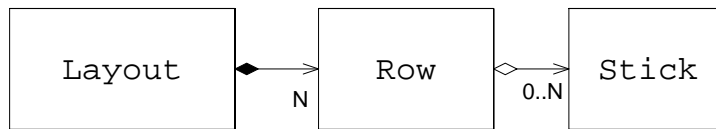


Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 28

## Composition & Aggregation (cont.)



### 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. It can be thought of as if the Row rectangles were drawn completely inside of the Layout rectangle.
- In this example, the rows are a fixed and permanent part of the layout, but the number of sticks within a row changes as the game progresses.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 29

## Composition, Aggregation, & Associations

### Composition:

- Object B is part of object A, such that B is created when A is created, and B is destroyed when A is destroyed. B has no existence other than as a part of A.
- Example: A book is composed of its pages and cover.

### Aggregation:

- Instances of class B exist independently of object A, but object A maintains knowledge of specific instance(s) of class B; used for collections and weak part/whole relations.
- Examples: A bookshelf holds a collection of books.

### Association:

- An object of class A holds a semi-permanent reference to an object of class B, with no containment semantics.
- Example: Book(s) have author(s).

### Dependency:

- Instances of class A have transient relationships with instances of class B.
- Example: A person reads a book, then gives it to a friend.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 30

## Composition & Associations (cont.)

```
class Person {
    private Life vida = null;           // Composition
    private Vector cells = new Vector(); // Aggregation
    private Person mother = null;      // Association
    private Person father = null;
    public Person( Person mom, Person dad ) {
        mother = mom;
        father = dad;
        vida = new Life();
        cells.addElement( 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 or philosophy...
- *Class exercise*: draw a UML class diagram for this example...

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 31

## Association Semantics

For Composition / Aggregation:

- Can the containee be contained within more than one container?
- Can the containee be accessed only via the container's interface?
- Are some operations on the containing class always applied to its parts?
- Does the existence of one object depend on another's?
- Are the lifetimes of the two objects always exactly the same?
- Does one object own/control the other's memory?
- Can the association be labeled *part of* or *is 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.

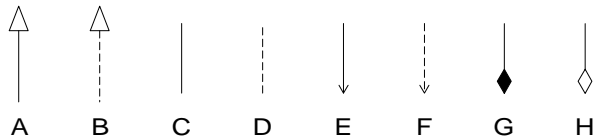
Consider the memory management implications (especially in C++)

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 32

## UML association review



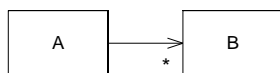
- A) Implementation Inheritance (Generalization)
- B) Interface Inheritance (also called Realization)
- C) Bidirectional Association
- D) Dependency (weak, transient Association)
- E) Unidirectional Association
- F) Unidirectional Dependency
- G) Composition
- H) Aggregation

Copyright © 1996 - 2008

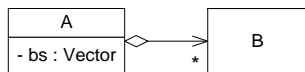
David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 33

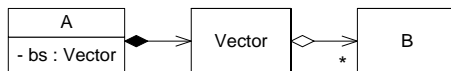
## Level of Detail



Manager / Client / Analysis /  
High-level Design



Programmer /  
Detailed Design



Pedantic / CASE Tool

- The level of detail in a UML model depends on the audience for which the model is intended.
- Notice that Java Collection Classes are often not explicitly shown.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

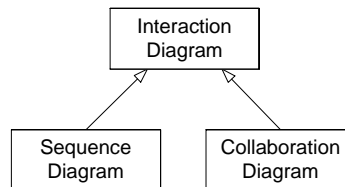
( III ) UML Core Diagrams - 34

## Interaction Diagrams

---

UML Interaction Diagrams:

- **Sequence**
- **Collaboration**



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.

So we will only present the more commonly used variation here: the Sequence Diagram...

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 35

## Sequence Diagram

---

- Shows the object collaborations over time for one scenario.
- Useful for capturing scenarios and use cases.
- Use one diagram per scenario.
- 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 possible thread of control within the scenario (if..else).
- A type of UML *Interaction Diagram*
  - Also called an Event Trace.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 36

# Interactions

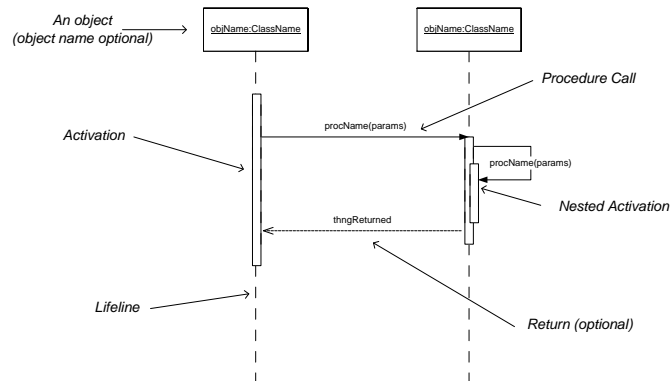
- Call
  - One object invoking a method on another, or on itself.
- Return
  - An object that was called returning a value as a result of the call.
- Send
  - One object sending an asynchronous signal to another, or itself.
- Create
  - One object instantiates another.
- Destroy
  - One object destroys another, or itself.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 37

# Sequence Diagram Notation

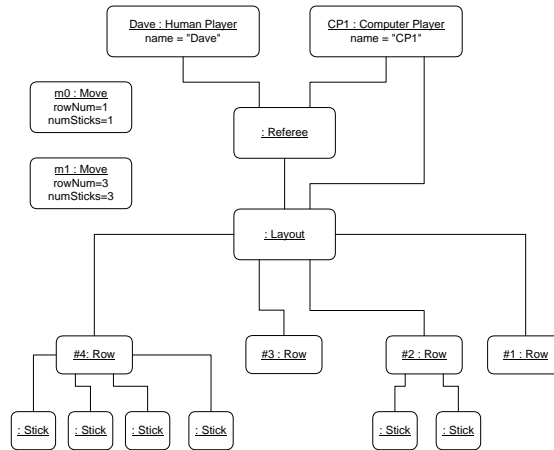


Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 38

## Example: Sticks Game Objects

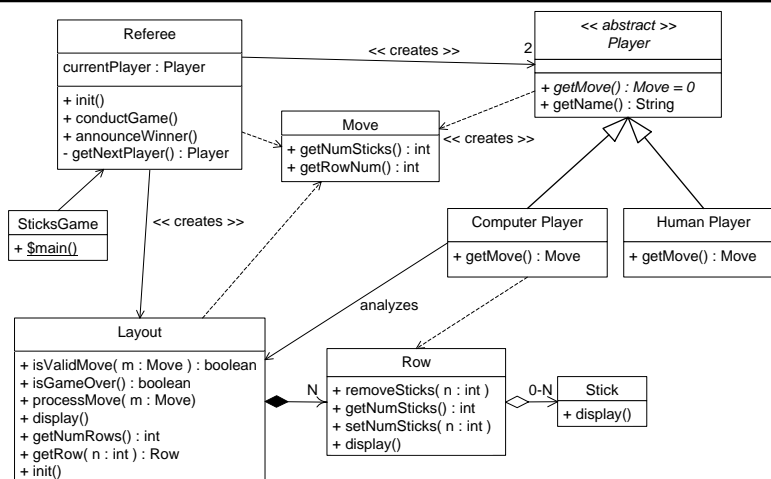


Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 39

## Example: Sticks Game Class Diagram



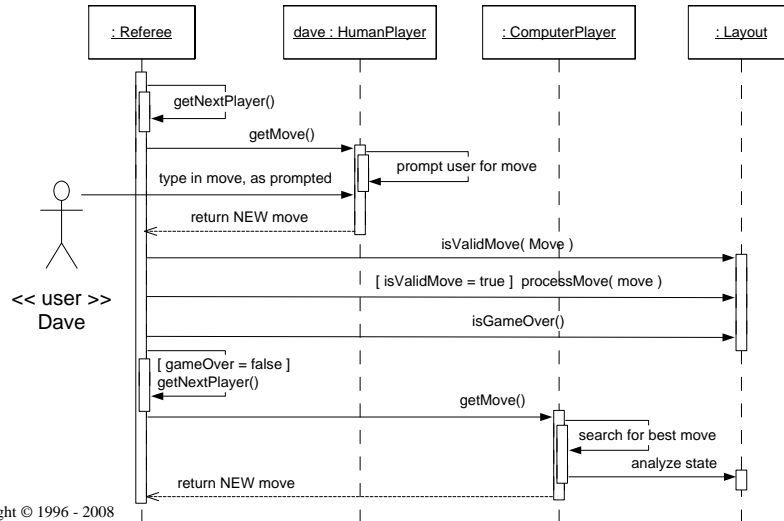
• Identify the GRASP patterns in this design.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 40

## Example: Sticks Game Sequence Diagram



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 41

## 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();
        } } }
    
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 42

## Example: Sticks Game Java (cont.)

- The Sticks Game design can be prototyped easily as a *console program* with no graphics and a dumb ComputerPlayer. Then iterate... refactoring and adding features such as a Minimax computer player search, and a *GUI*.
- But first, to get started coding easily, begin with a class called SticksTest ...

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

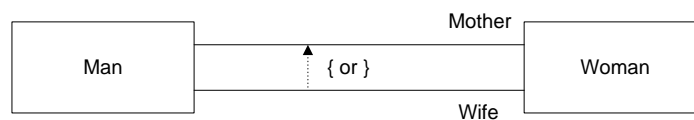
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 43

## Constraints

- Used between two associations.
- Used to model business rule semantics.
- The language used within the {} is not specified, although there are some conventions.



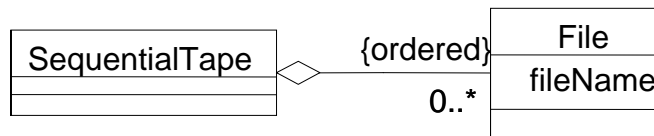
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 44

## Ordering

- Used where the objects at one end of an association have an explicit ordering.
- {ordered} is an example of a common constraint.

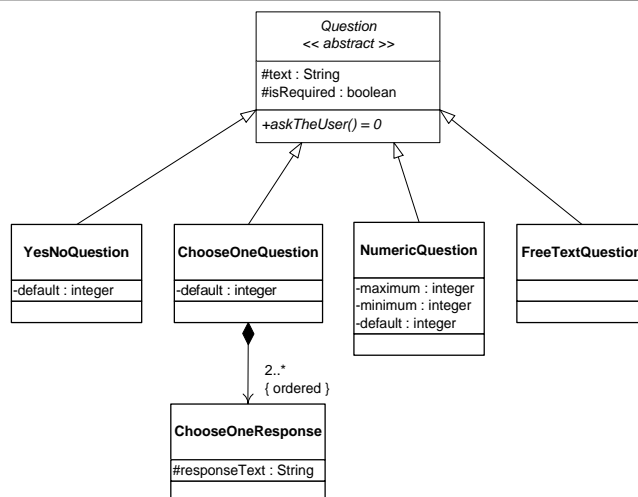


Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 45

## Example: The Question Model



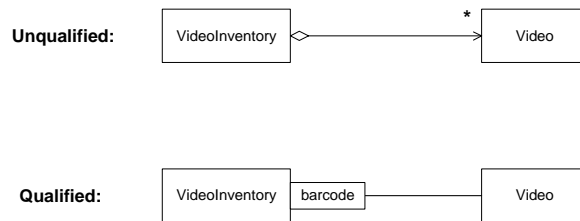
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 46

## Qualified Associations

- Used where the instances of the class at one end of a one to many association have unique identifiers to specify a single instance from the ‘many’ side.
- Qualified associations are usually implemented with some kind of Dictionary (also known as an Associative Array), such as a Hash Table or Tree Map.
- Why is the qualified association a better model?



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 47

## Packages

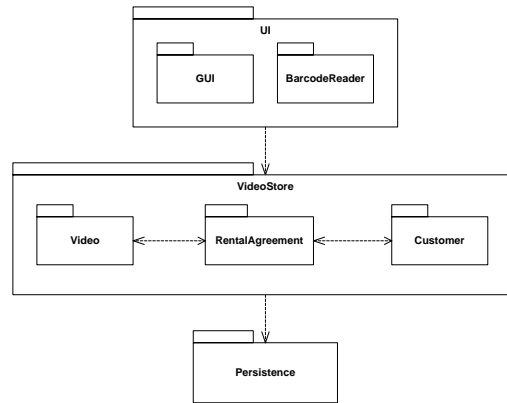
- “A general-purpose mechanism for organizing elements into semantically related groups.”
- Logical groupings of entities.
- Packages can be nested.
- At the highest level, a package contains an architectural entity (e.g., User Interface, Business Domain, subsystem).
- At the lowest level, a package may represent a single person’s work.
- Packages are part of the Java programming language.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 48

## Example: Video Store



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 49

## Packages (cont.)

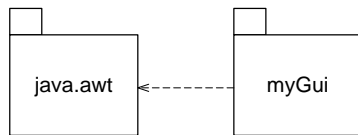
- Help control namespace
  - Names are not visible outside of package unless they are declared to be visible.
  - Java supports packages directly, and anything that does not have an explicitly declared visibility (i.e., public, private, or protected) is visible throughout the package, but not outside.
  - C++ provides *namespaces*, which provide similar, but not identical, name scoping.
- In UML, packages provide a mechanism for encapsulation.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 50

## Packages (cont.)



Packages help to conceptually organize classes by providing a higher level abstraction.

- There are many classes that comprise the Java AWT, but it is simpler to think of them all together, as one package.

Packages solve class naming problems by providing an additional naming context.

- Both the Java AWT and myGui packages might have a class called Event, but this is not a problem; in Java, say: java.awt.Event; or myGui.Event to be clear.

UML Package diagrams are often used in the design process to visualize the interdependencies between packages; such dependencies should be minimized.

- In the above example, the package myGui is *dependent on* the package java.awt.

Copyright © 1996 - 2008

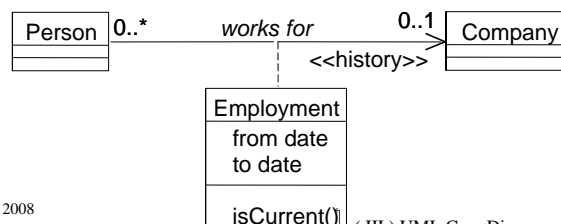
David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 51

## Association Attributes

Attributes sometime depend on the identities of two objects. If such an attribute is more complex than a scalar value, it may be modeled as a class. This effectively allows attributes & methods to be modeled as associations.

- In this example, an **Employment** is an attribute of the *works for* association.
- Note: The semantics of association classes (as modeled) indicate that for every Person / Company pair, there is exactly one Employment instance. Thus the model dictates that a Person cannot work for the same Company two different times!
- Note: The << history >> stereotype (from *Analysis Patterns*, by Fowler: ISBN 0-201-89542-0) clarifies the time aspect of the relationship... It says that a Person over time may work for many Companies, but at any one time, may work for 0 or 1 Company.



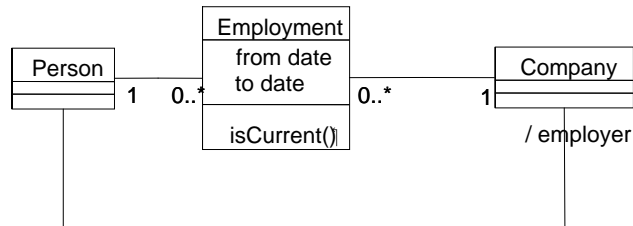
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 52

## Association Attributes (cont.)

- The semantics of association attributes works well for relational database design, where, in order to model a many-many relationship, a third “association table” is required. But it does not work so well for object modeling; in fact, neither Java nor C++ directly supports the idea.
- In the implementation, the Person might maintain a list of Employments; each Employment would know about one Person and one Company.
- Note the changes in the association cardinalities, and the fact that the employer relationship is now *derived* (indicated with a “/”).



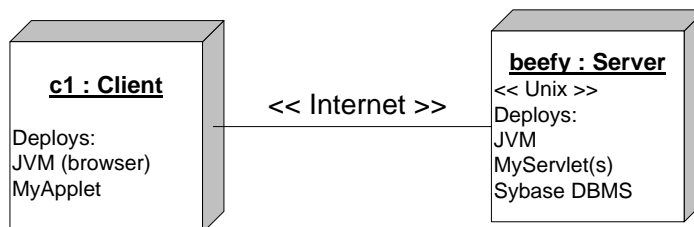
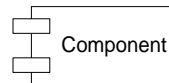
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 53

## Deployment Diagrams

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



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

( III ) UML Core Diagrams - 54