Object - Oriented Programming & Design

Part V: Object- Oriented Design

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster.   All rights reserved.

Object- Oriented Design

- Domain analysis done ... what is the best way to code it?
- Make UML models.
- Code incrementally & iteratively w/ refactoring.
- Reduce complexity.
- Study Design Patterns - review GRASP patterns (from section II).
- More patterns: Pattern-Oriented Software Architecture 2, for example.
- Experience helps – promote mentoring.
- Pay attention to the design ***process.***
- More art than science.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster.   All rights reserved.

Good Object-Oriented Design

- No model that solves the problem at hand is intrinsically wrong, but some models are better than others because they prove to be more flexible, extensible, easier to understand, less complex...
- Where is the ***best*** place to draw boundaries between parts of a system?
- The first take on a design model is seldom the best one.
- Be architecture-centric, rather than feature-centric.
- Strive to generalize service-oriented infrastructure for reuse, so that the code can easily evolve with the inevitable changes in requirements.
- Predictable extensions should be designed to be accomplished additively and non-invasively; two different designs might yield identical functionality and yet be very different in this respect.
- Good design is as simple as possible, for humans.

Copyright © 1996 - 2008

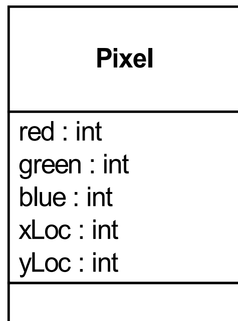
David Leberknight & Ron LeMaster. All rights reserved.

Attributes vs. Classes

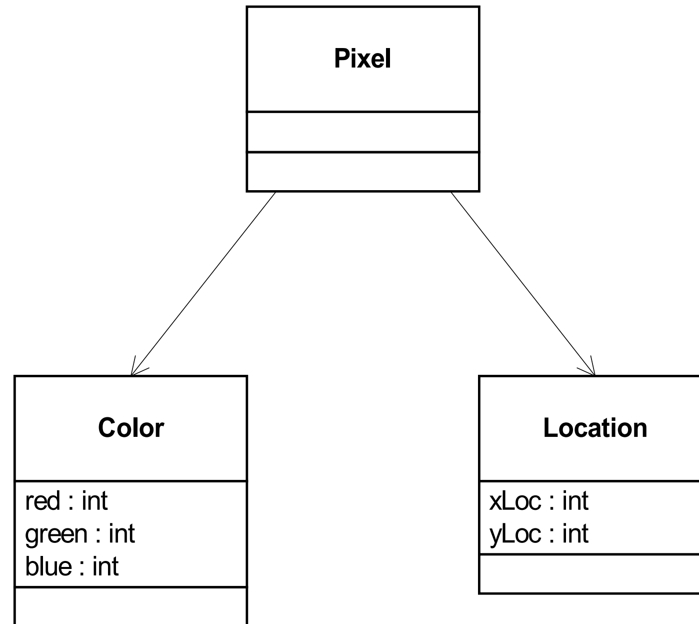
- Question: When should something be modeled as an attribute, and when should it be modeled as a class?
- Consider the following:
  - Class Person
    - » Phone number
    - » Social security number
    - » Age
  - Class Pixel
    - » Color
    - » Location

Copyright © 1996 - 2008
David Leberknight & Ron LeMaster. All rights reserved.

Example: Pixel



or



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Attributes vs. Classes

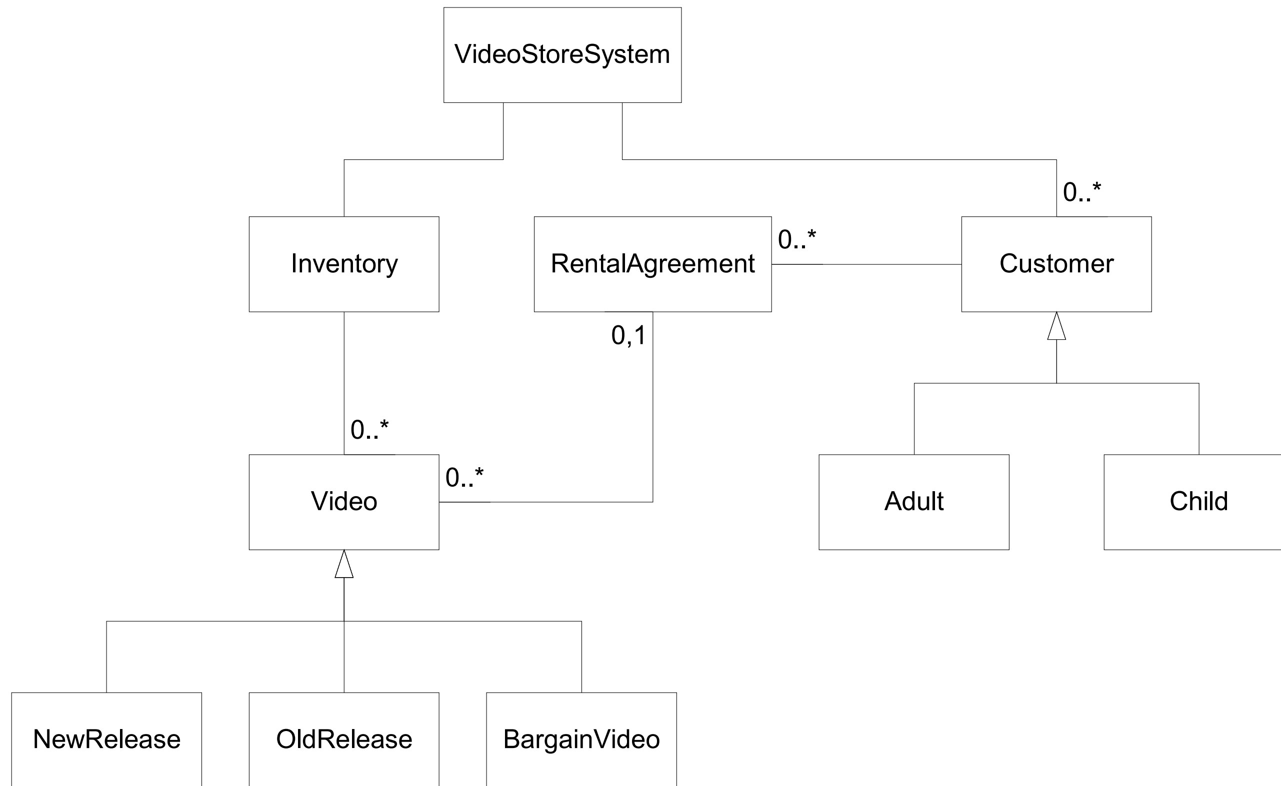
- Considerations:
  - Can the entity be represented by a primitive data type?
  - Will the entity be compared ***by value*** or ***by identity***?
  - Is the entity's value encoded?
  - Is there any behavior associated with the entity?
  - Do parts of the entity have their own meaning (e.g., a phone number has an area code)?
- When in doubt, create an associated class rather than an attribute.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

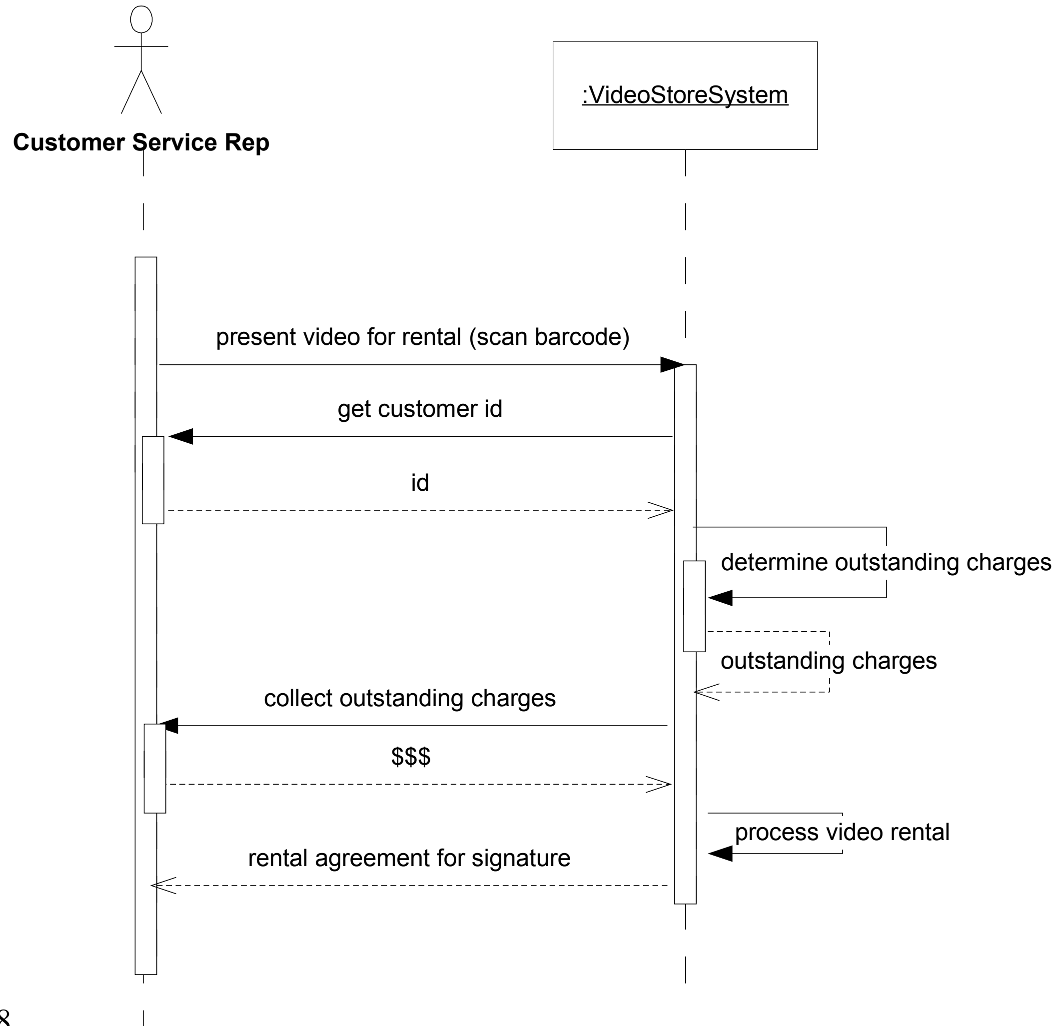
Video Store Class Model (Analysis)

First attempt at a domain model...



Copyright © 1996 - 2008
David Leberknight & Ron LeMaster. All rights reserved.

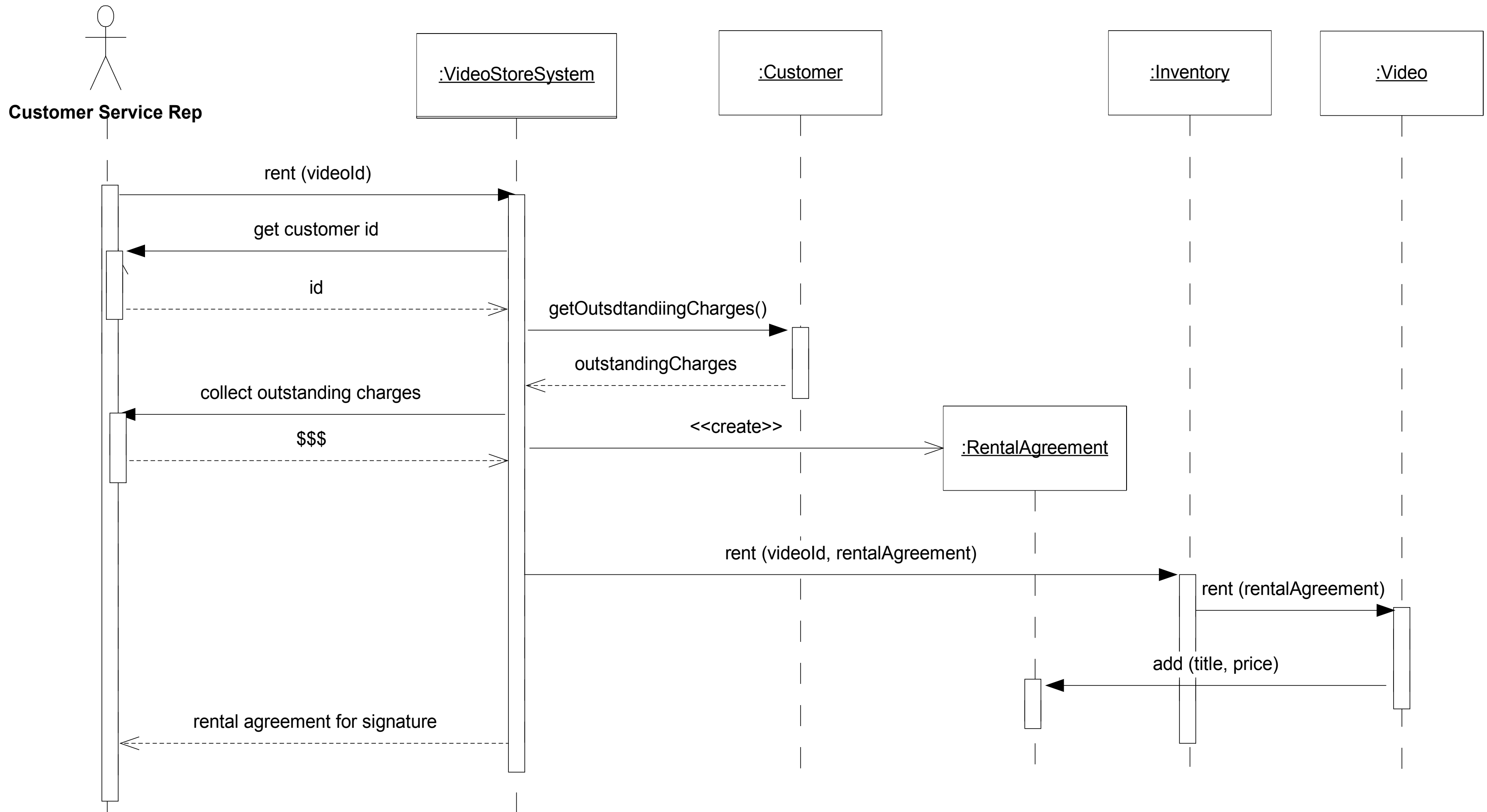
Scenario: Rent a Video (analysis)



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

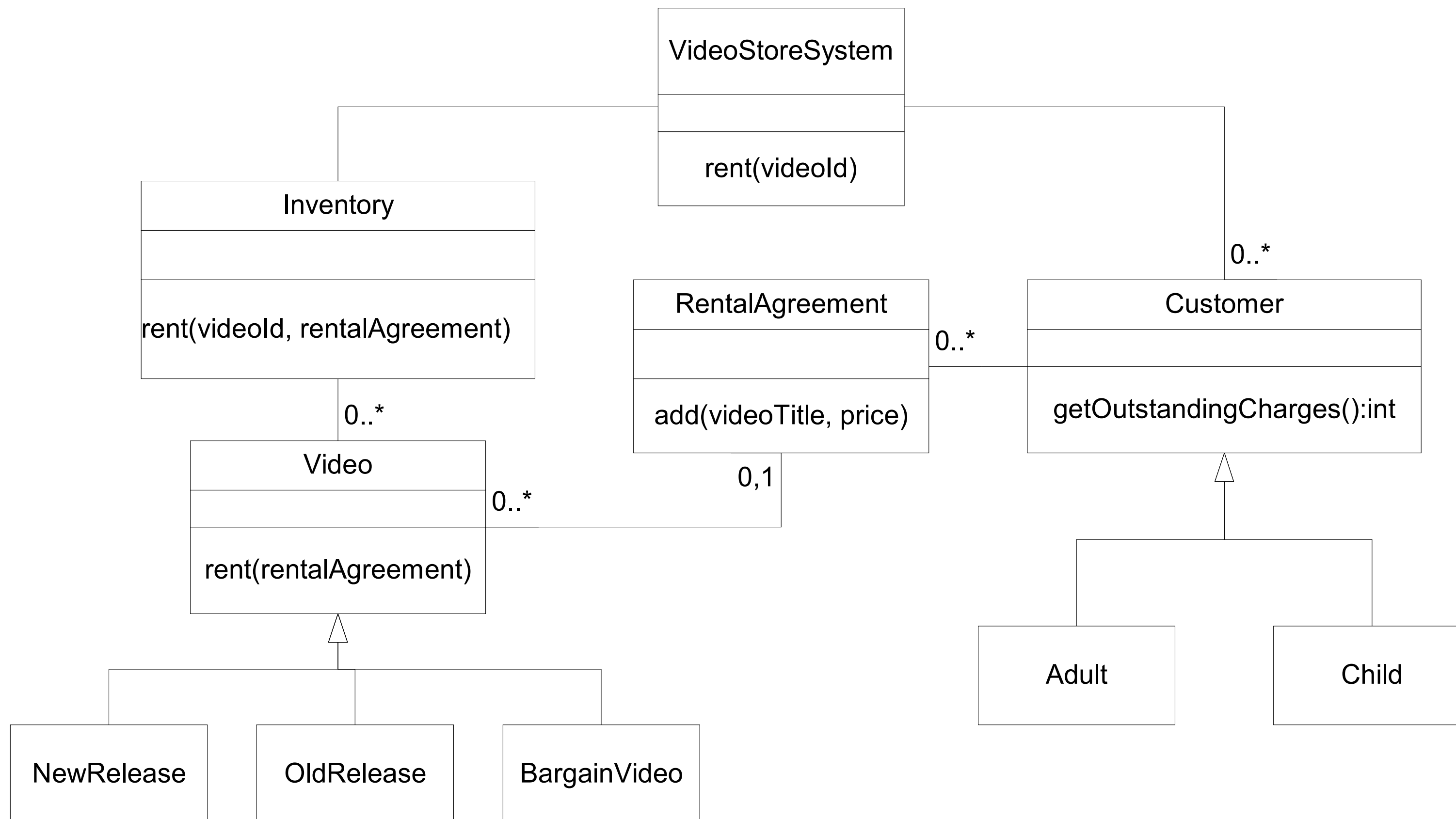
Scenario: Rent a Video (take 2)



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Video Store Class Model (begin design)



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

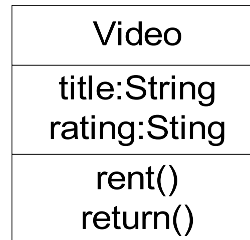
Specification Classes

- It is often advantageous to separate out certain object attributes.
- Useful when a domain has numerous objects that share identical fields.
- Useful when specifications exist independently of the objects to which they apply.
- Example:
  - Does the Video Store have a copy of *Titanic* in stock?
  - Separate the Title from the Video.
  - This way the VideoCatalog can reference Titles without referencing individual Videos / DVDs.

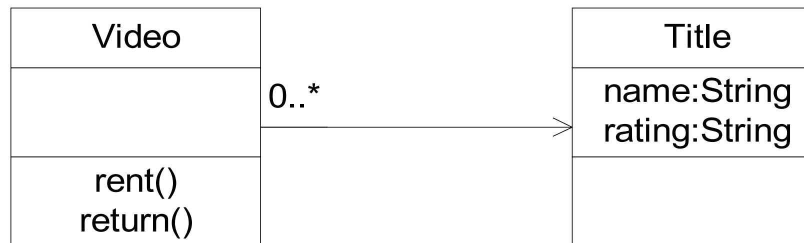
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Specification Class Example



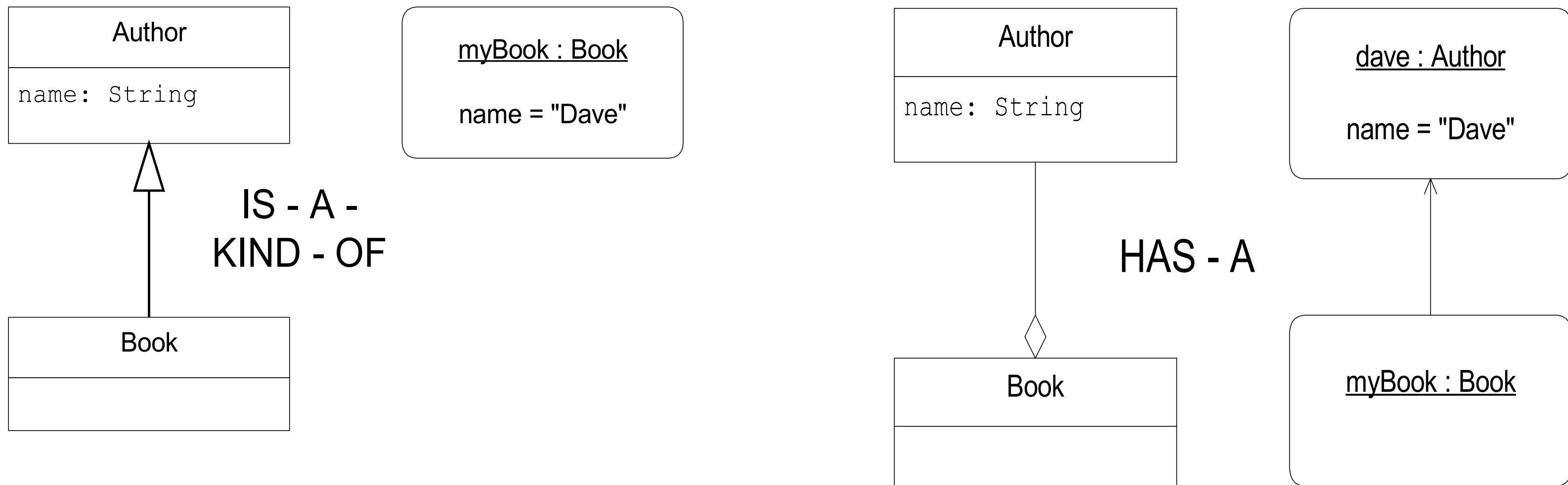
vs



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Inheritance vs. Delegation



Consider two ways to model ***"Books have Authors"*** ...

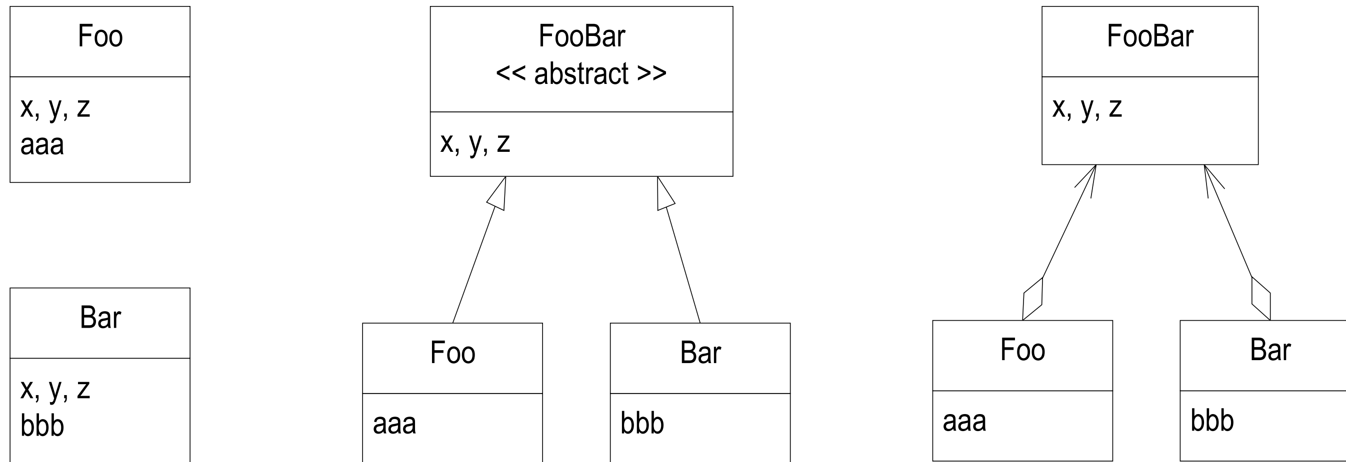
The inheritance configuration (on the left) logically works because the Book inherits the Author's name, but it is a **BAD design**; the problem is that a Book *is-NOT-a-kind-of* Author.

We would say instead that a Book *has - a* Author.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Inheritance vs. Delegation



It can sometimes make sense to model a situation with either inheritance or delegation; you have to choose.

- Compare two design approaches for `Foo` & `Bar` to share the `x,y,z` functionality…

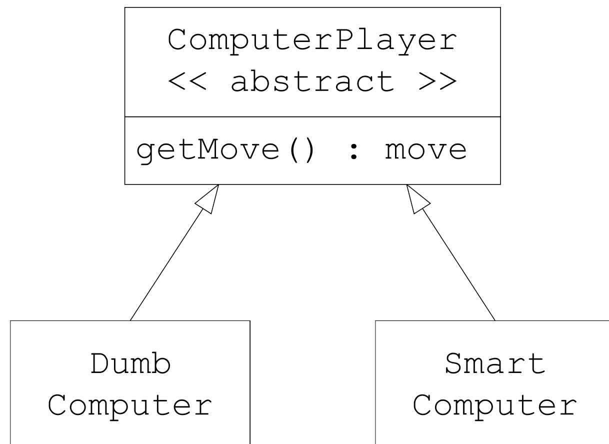
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

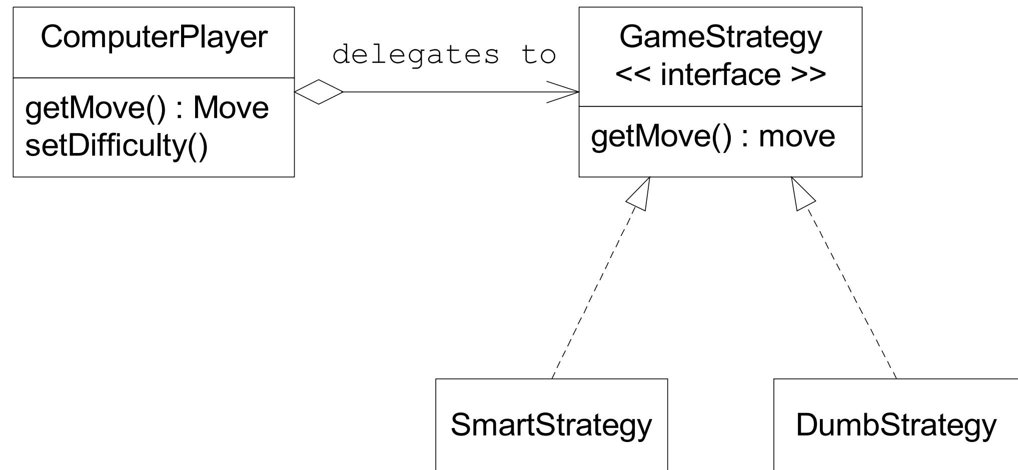
The Strategy Design Pattern

- Consider a computer game with the requirement that it is possible to change the difficulty level at any time.
- Solution: Encapsulate the algorithms while allowing them to vary independently from the client.
- How does the client of Model A change the difficulty level?

<u>Model A</u>



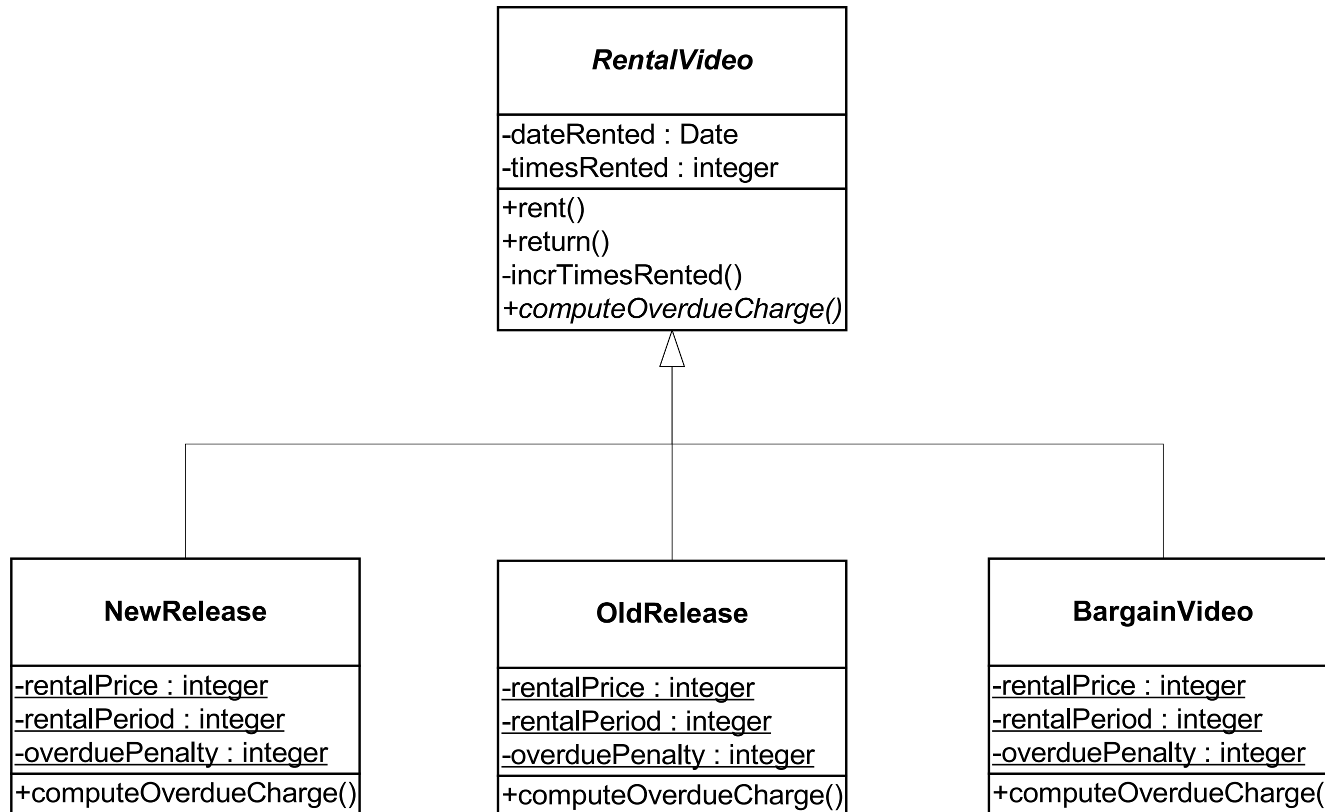
<u>Model B</u>



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Example: RentalVideo

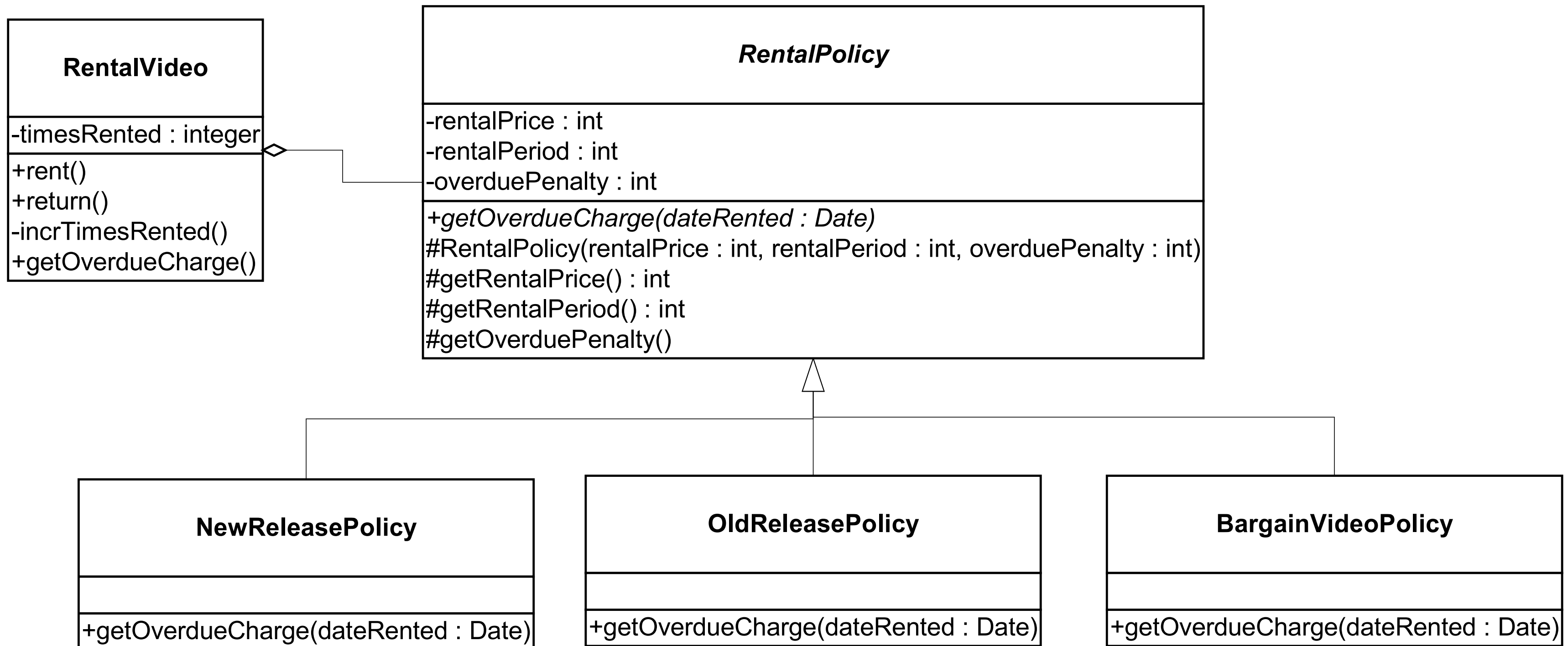


Remember: an object, once instantiated, cannot change its class.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Another Strategy Example

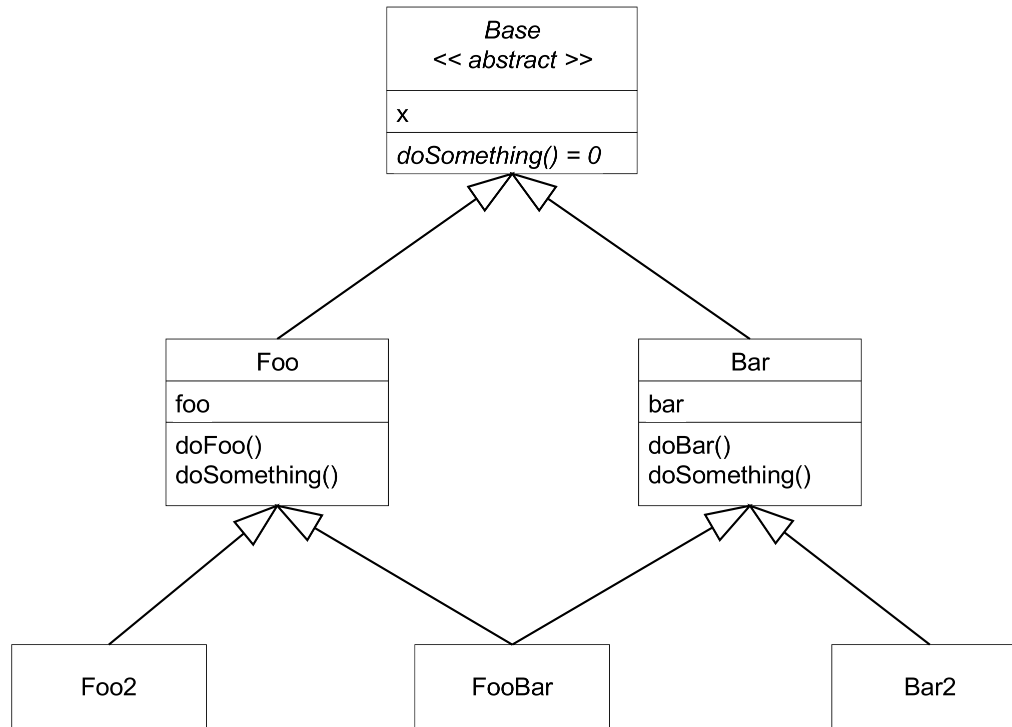


Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Multiple Implementation Inheritance

- Multiple ***implementation*** inheritance (**possible in C++)** is tricky.
- For FooBar, should there be one or two copies of the data, x?
- Which implementation of doSomething() should FooBar inherit?



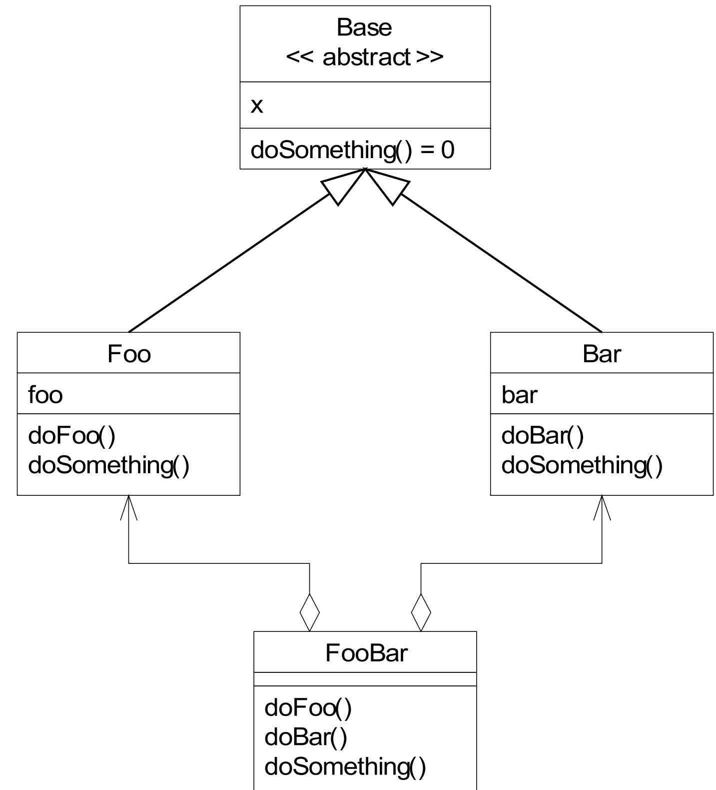
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Multiple Inheritance

Alternative:

- Use *aggregation* and *delegation*:
  - Reduces class hierarchy size.
  - More loosely coupled.
  - Run-time flexibility for FooBar's delegation to Foo or Bar.



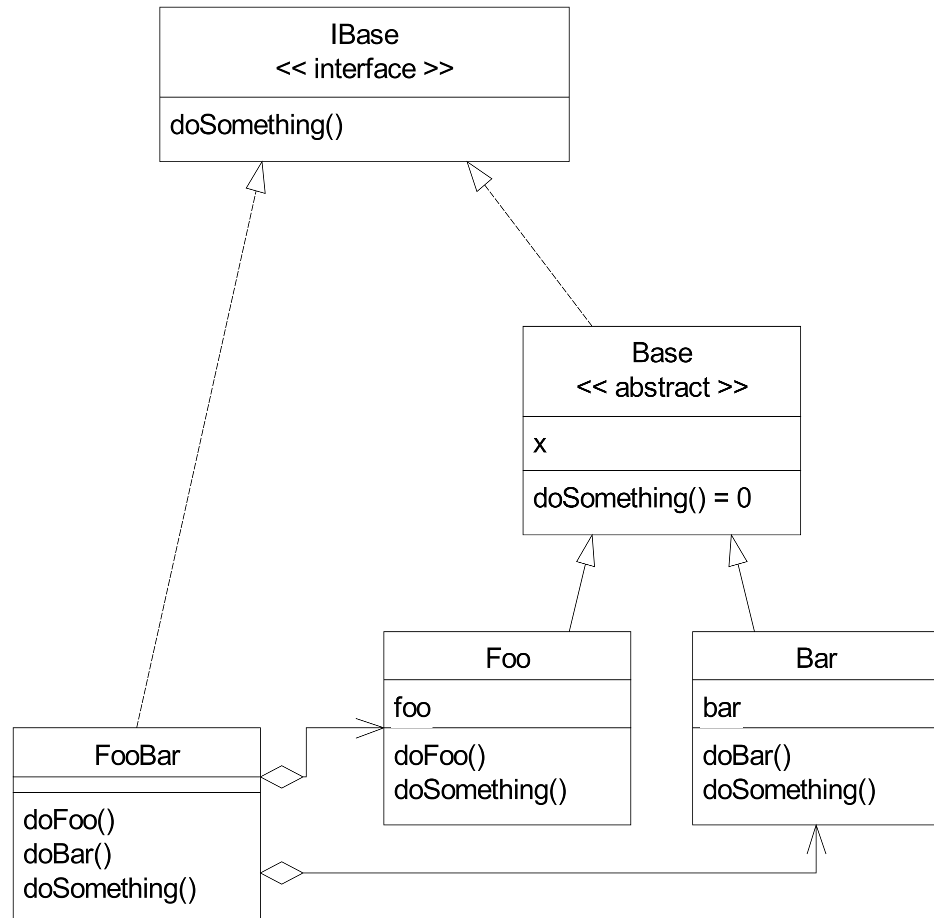
The problem here is that FooBar is no longer a subclass of Base; polymorphism has been disabled.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Multiple Inheritance

- Use an ***interface***.

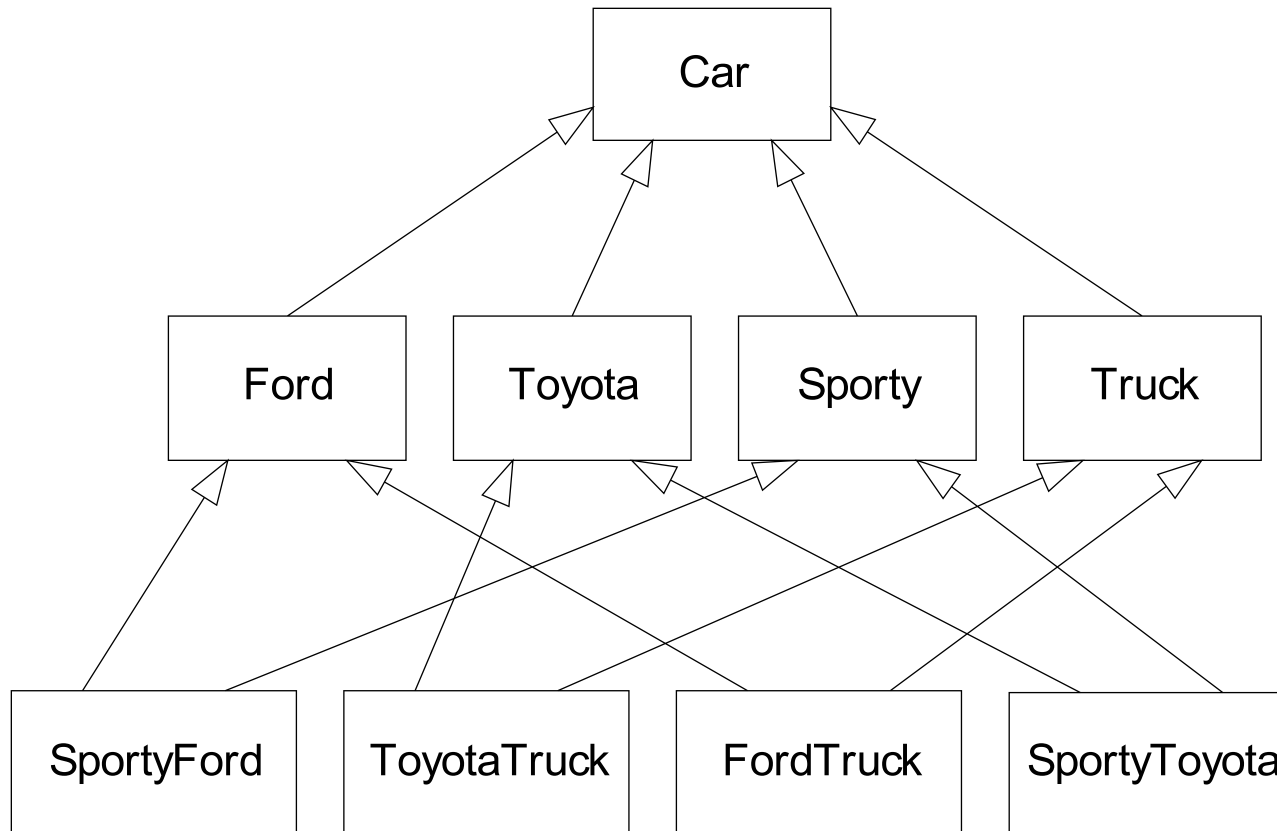


Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Multiple Inheritance Silly Example

- How can we simplify this design?



Copyright © 1996 - 2008

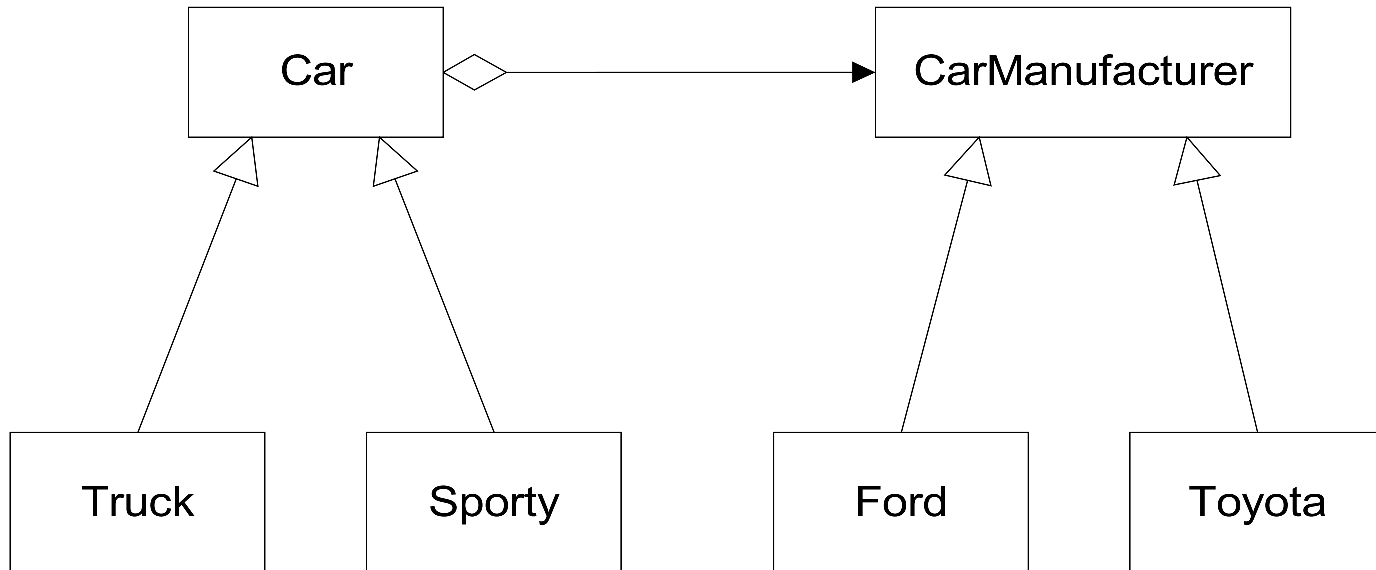
David Leberknight & Ron LeMaster. All rights reserved.

The Bridge Design Pattern

Apply the ***Bridge*** Design Pattern

Intent: Decouple a class abstraction from its implementation.

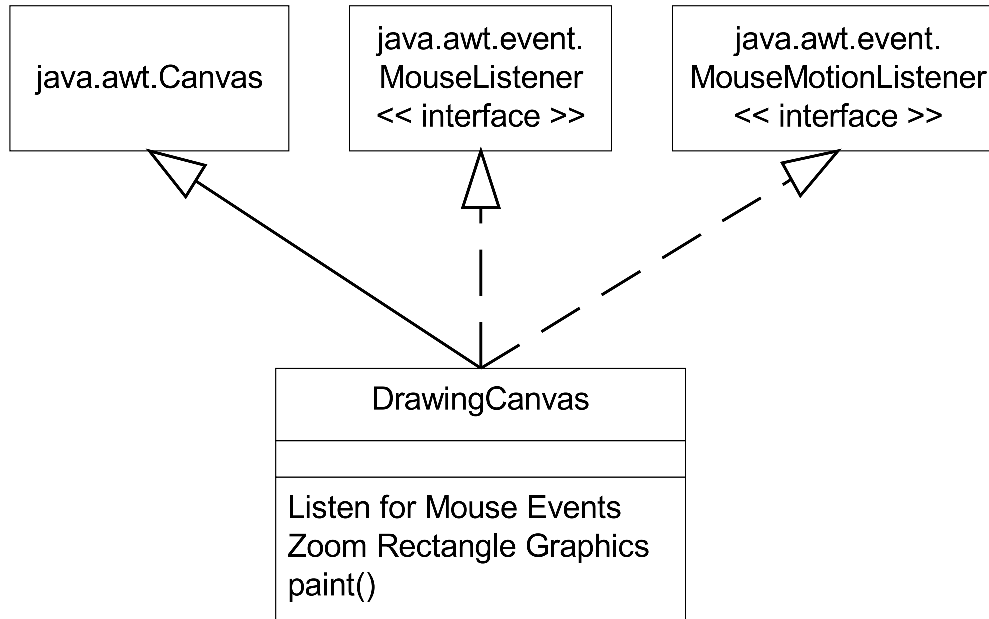
- You might use Bridge when you might otherwise be tempted to use multiple inheritance...



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Multiple Interface Inheritance



- Note that `DrawingCanvas` inherits the implementation of `awt.Canvas`, and additionally implements two other 2 interfaces.
- Java rules prohibit multiple ***implementation*** inheritance, but this is OK.

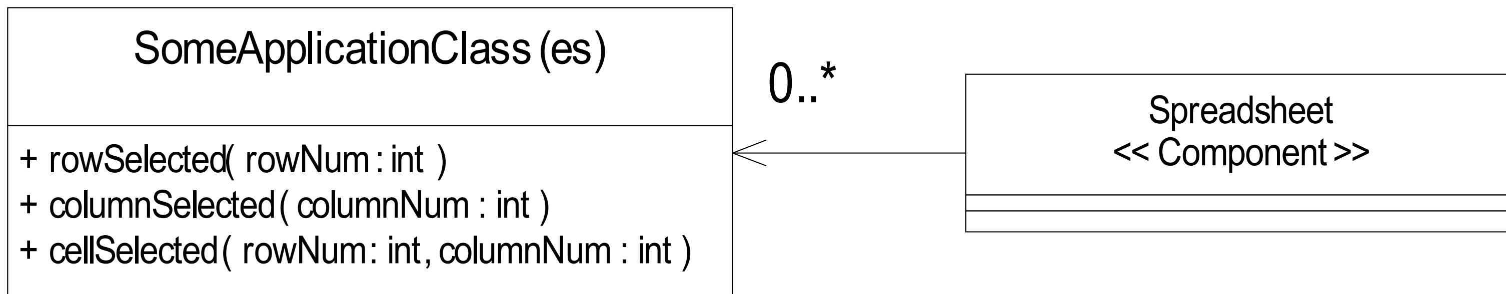
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Reusable Components

A reusable class must know nothing about its context.

- Example: A Spreadsheet widget has to notify application classes whenever a new row, column, and/or cell gets selected.
- How can it do this without detailed knowledge of the application classes?

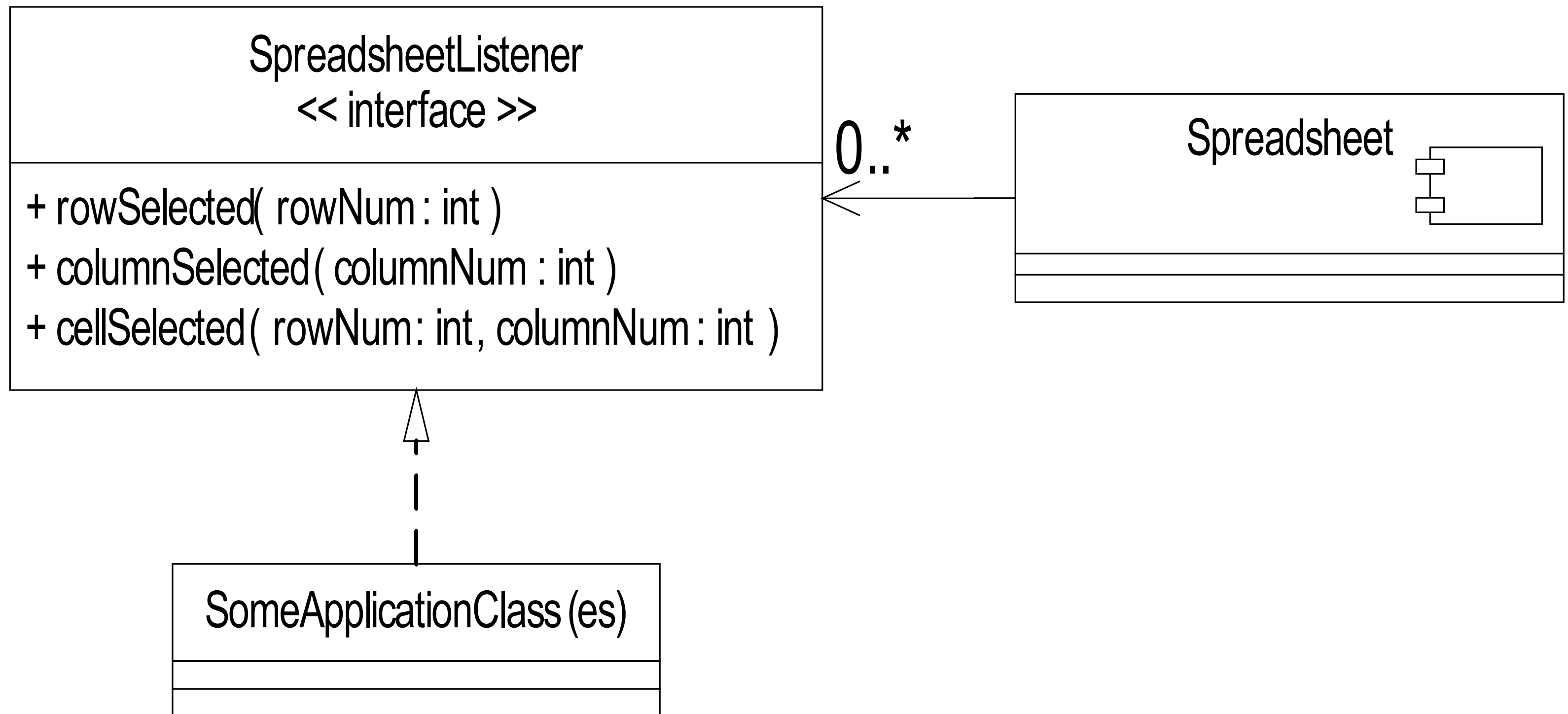


Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Reusable Components

- Use an ***interface***.



Copyright © 1996 - 2008
David Leberknight & Ron LeMaster. All rights reserved.

Inheritance versus Delegation

- Inheritance allows generalization and abstraction.
- Delegation allows dynamic run-time flexibility.
- Reuse via inheritance is tightly coupled to the base class (breaking encapsulation).
- Interfaces encourage loose coupling.
- Inheritance allows functionality to be extended.
- Delegation allows functionality to evolve independently.
- is-a-kind-of ... *or* ... has-a ?
- Bonus question: *When might you use an interface that defines no methods?*

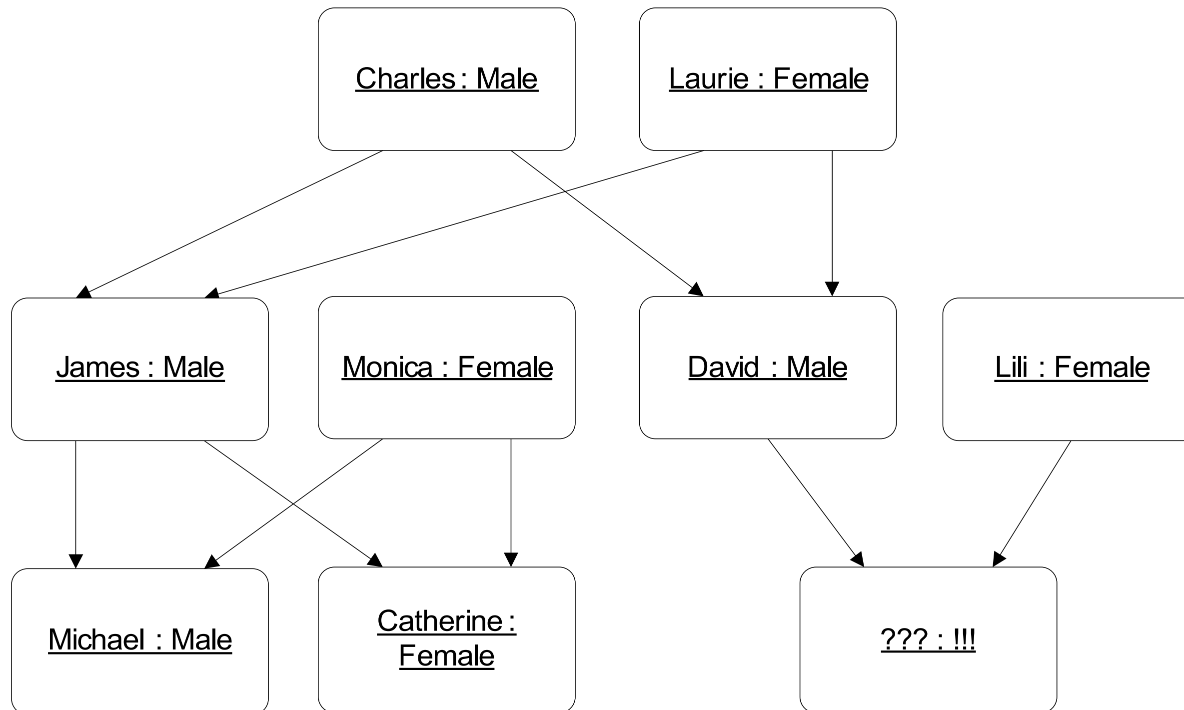
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Modeling Example: Biological Families

Here is an example object hierarchy...

What would the class diagram look like?



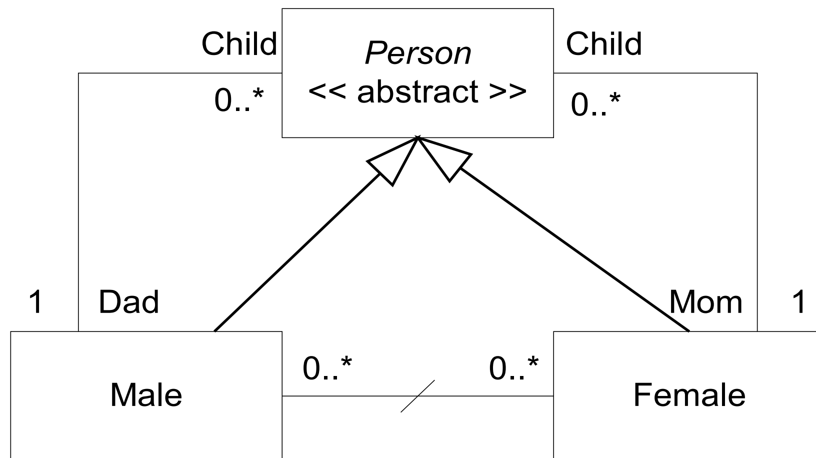
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

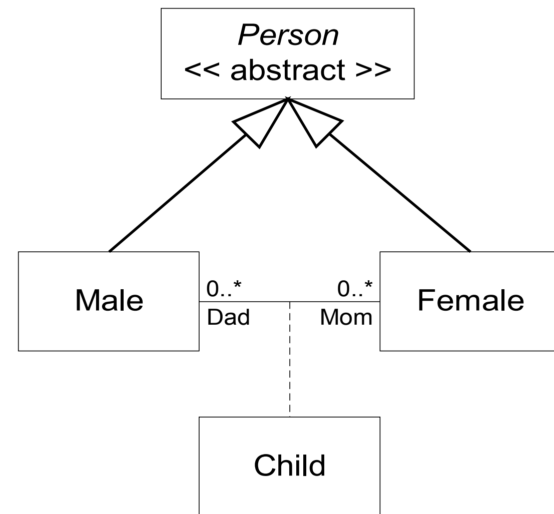
Modeling Example

1. Child is a kind of Person, so Model B shouldn’t show it as a distinct class.
2. A given Male/Female combination can have more than one Child; Model B specifies one association between a given male and a given female.

Model A

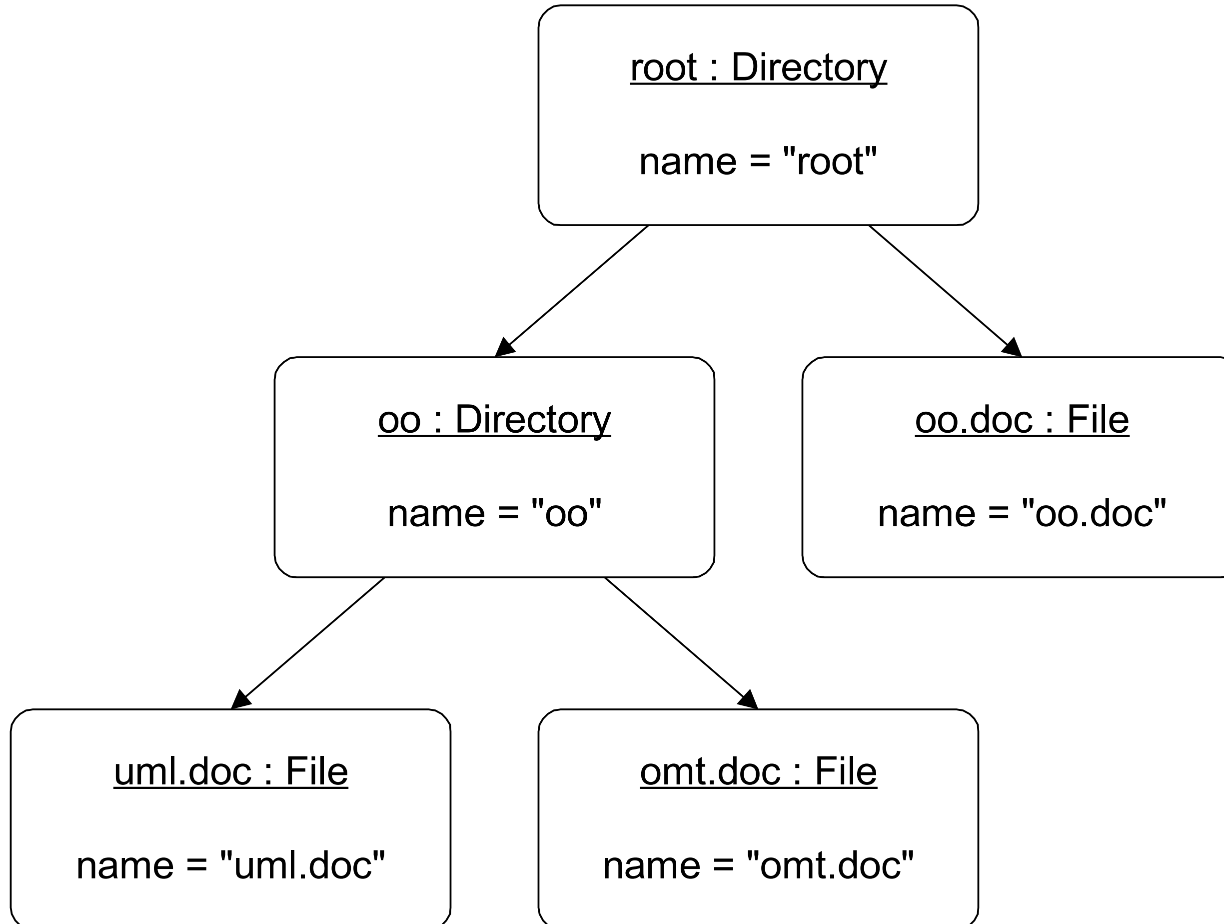


Model B



Copyright © 1996 - 2008
David Leberknight & Ron LeMaster. All rights reserved.

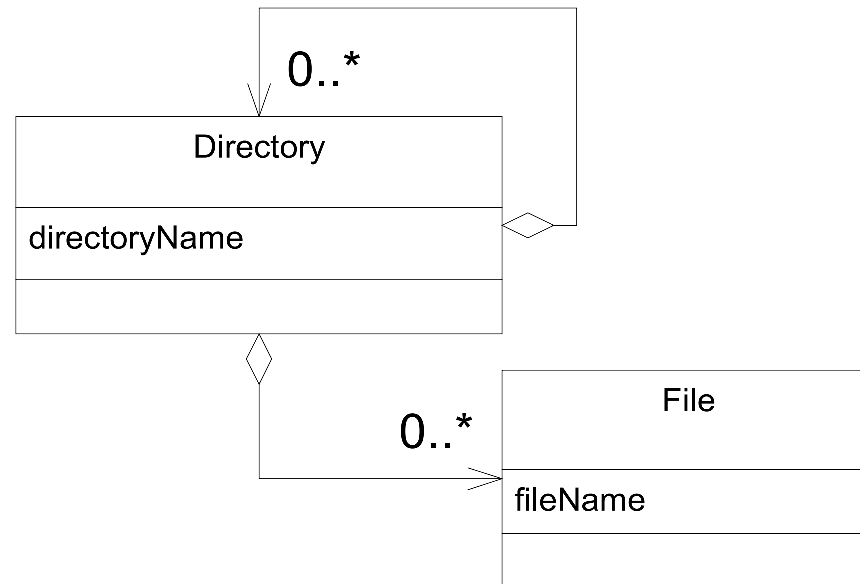
Recursive Aggregation



Copyright © 1996 - 2008
David Leberknight & Ron LeMaster. All rights reserved.

Recursive Aggregation (cont.)

- This is how we often model *object hierarchies*, such as a the directory hierarchy for a file system.
- Sometimes we use the ***Composite*** Design Pattern instead...



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Design Pattern: *Composite*

Intent: Model a tree-like hierarchy, such that branches and leaves can be manipulated uniformly.

Implementation: Usually *recursive*.

Applicability:

- Any hierarchical organization of objects & compositions of objects.

Pros:

- Simplifies client code whenever clients don't know or care whether a given node (component) is a leaf or a branch.
- It is easy to add new component types.

Cons:

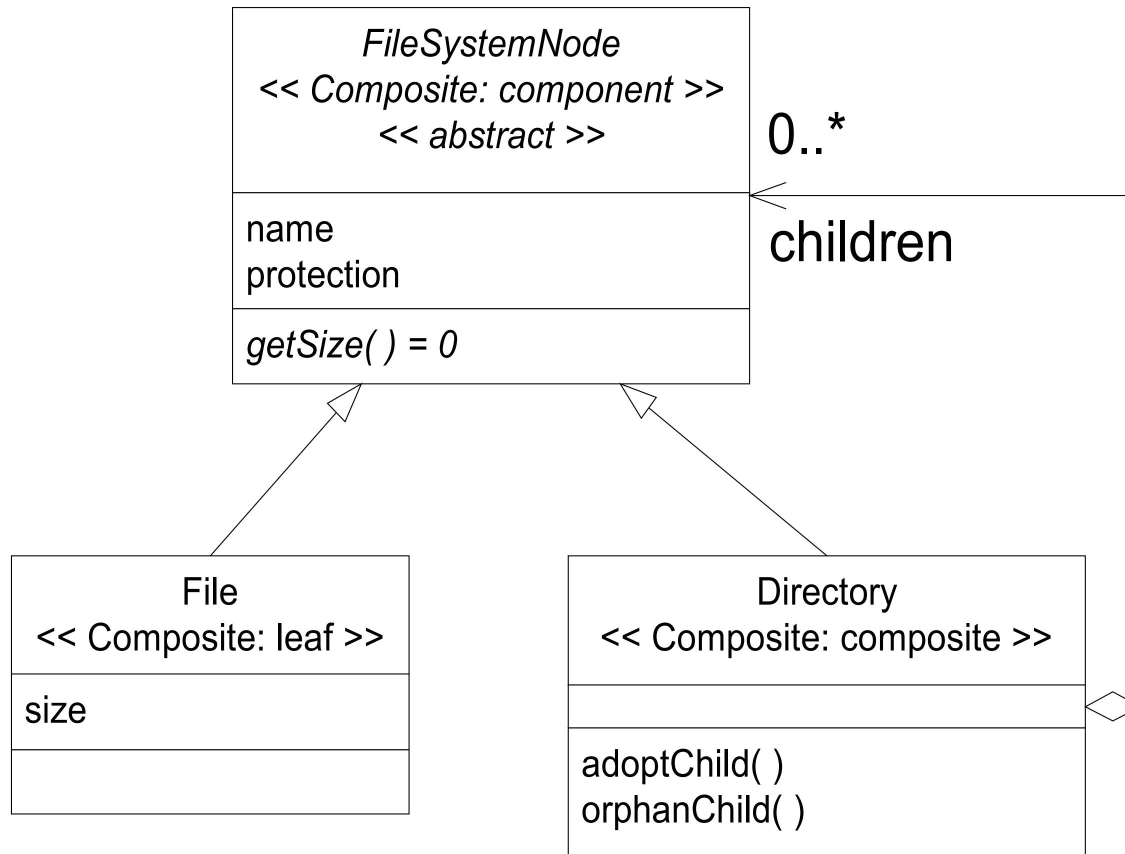
- Whenever the client *does* care about the type of a given node, it must implement run-time type-checks and down-casts; otherwise, compile-time checks would have been sufficient.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Composite Example

- Write pseudo-code for the getSize() method(s).



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

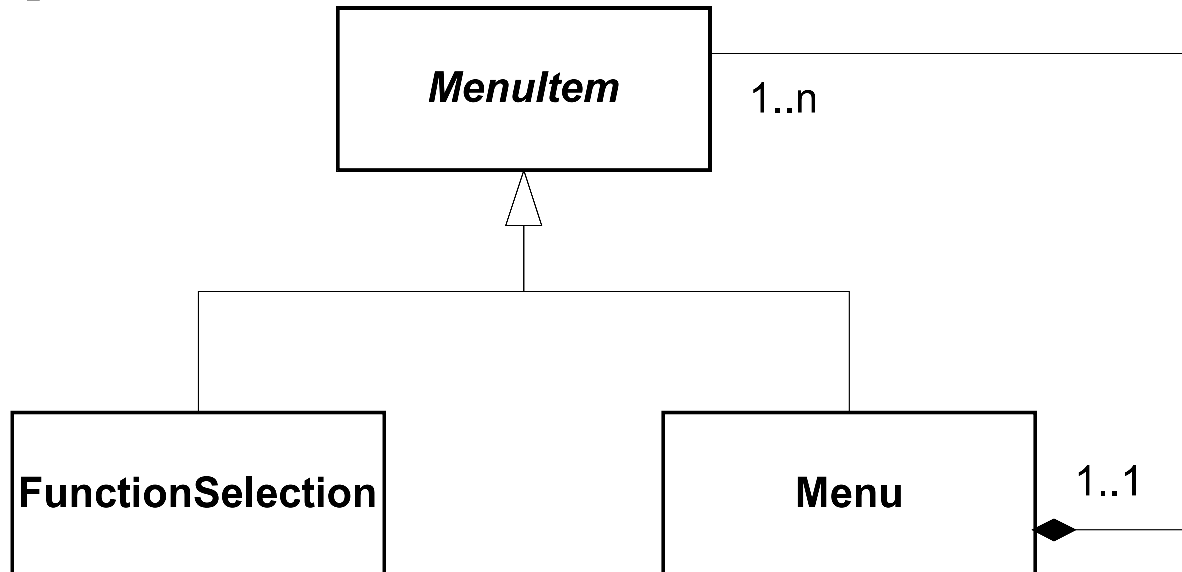
Composite Example Code

```
class Directory extends FileSystemNode {
 public int getSize() {
    int size = 0;
    FileSystemNode child = getFirstChild();
    while( child ) {
       size += child.getSize(); // Recursive & Polymorphic
       child = getNextChild();
    }
    return size;
}...}
class File extends FileSystemNode {
public int getSize() {
    return size;
}...}
```

Copyright © 1996 - 2008
David Leberknight & Ron LeMaster.   All rights reserved.

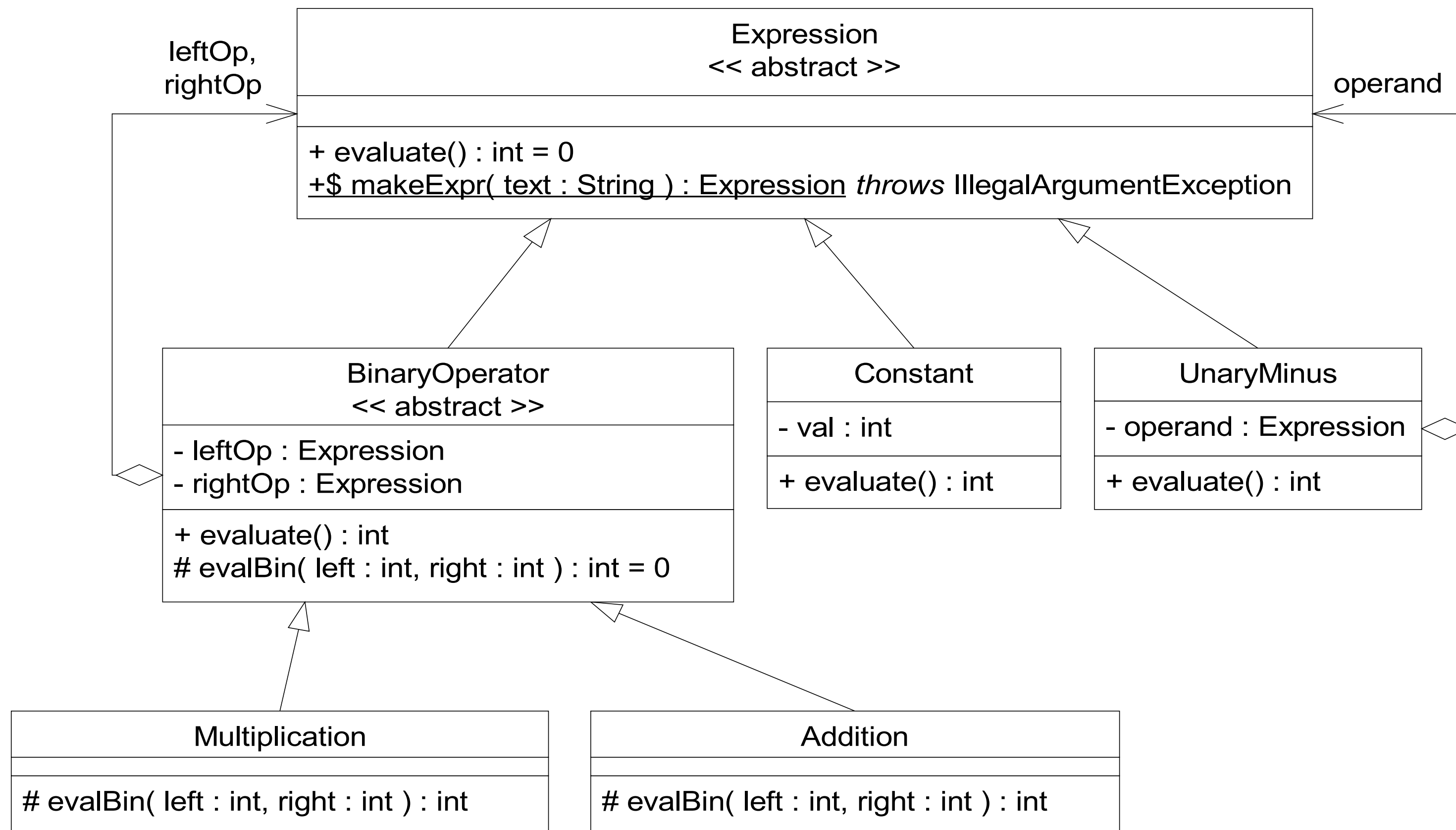
Another Composite Example

- UI Menus are composed of menu selections.
- A selection can:
  - Invoke any program function.
  - Bring up another menu.



Copyright © 1996 - 2008
David Leberknight & Ron LeMaster. All rights reserved.

Composite Example: Expressions

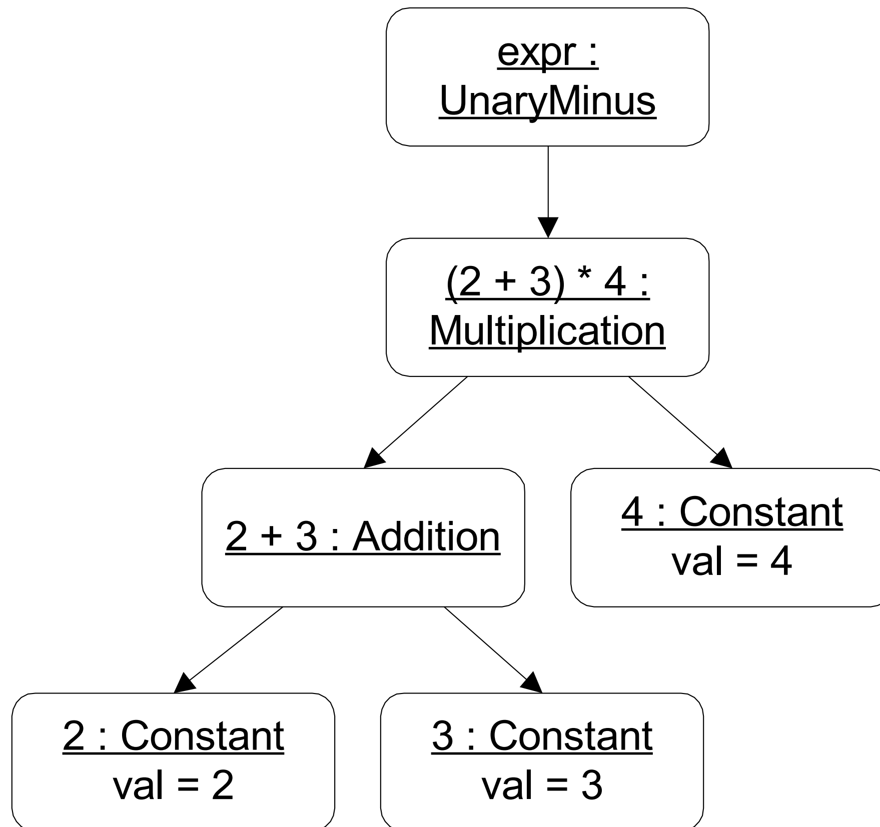


Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Composite Example

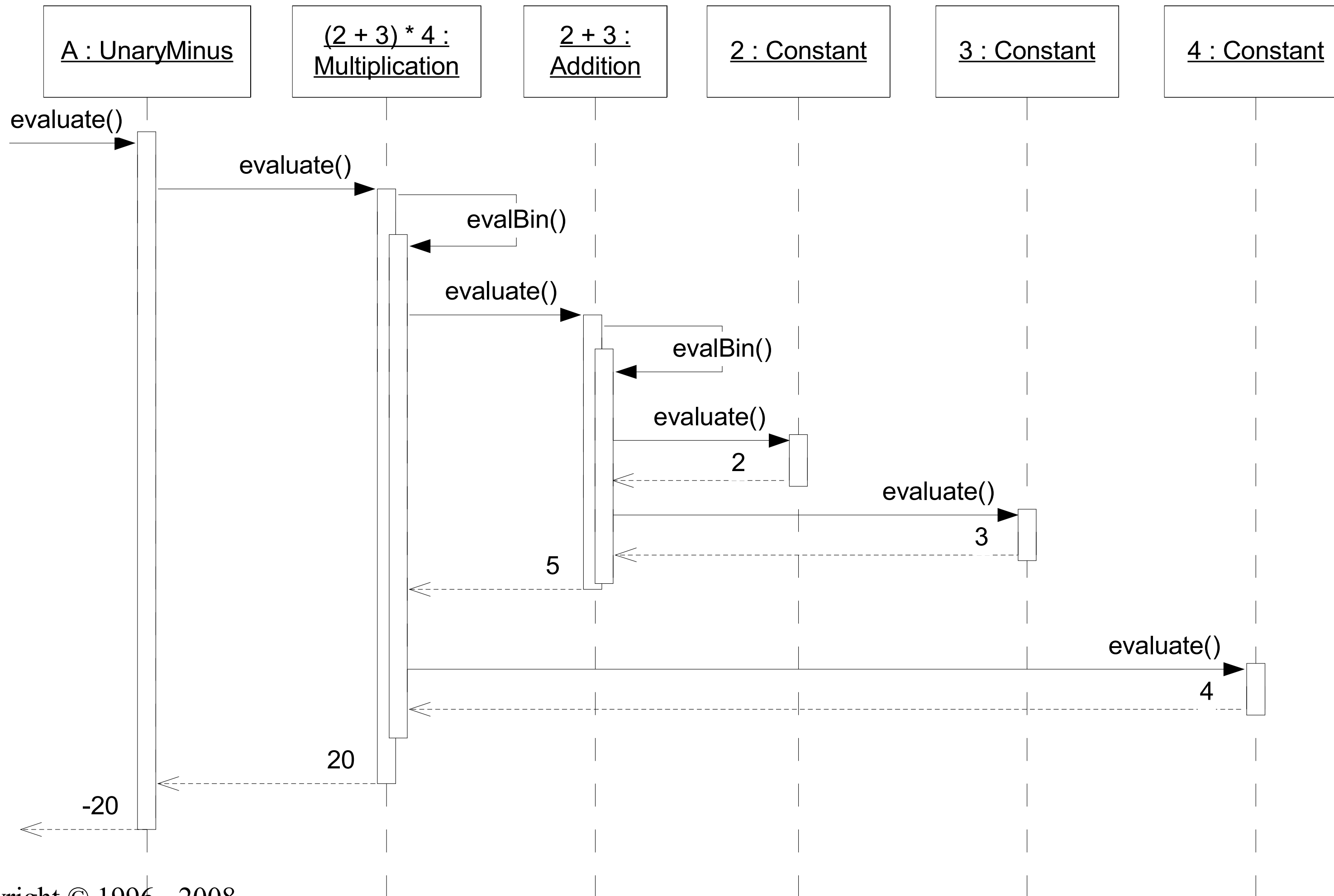
`expr = -((2+3)*4)` *evaluates to* `-20`



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Composite Expressions



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Composite Expressions (cont.)

```
// Ignore Constructors. . .
abstract class Expression {
  public abstract int evaluate();
  public Expression makeExpr( String text )
    throws IllegalArgumentException { // Virtual Constructor
} }
class Constant extends Expression {
  private int val;
  public int evaluate() {
    return val;
} }
class UnaryMinus extends Expression {
  private Expression operand;
  public int evaluate() {
    return - operand.evaluate();
} }
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Composite Expressions (cont.)

```
abstract class BinaryOperator extends Expression {
  private Expression leftOp;
  private Expression rightOp;
  protected abstract int evalBin( int left, int right );
  public int evaluate() {
    // Template Method
    return evalBin( leftOp.evaluate(), rightOp.evaluate() );
} }
class Addition extends BinaryOperator {
  protected int evalBin( int left, int right ) {
    return left + right;
} }
class Multiplication extends BinaryOperator {
  protected int evalBin( int left, int right ) {
    return left * right;
} }
```

Copyright © 1996 - 2008
David Leberknight & Ron LeMaster. All rights reserved.

Summary

- Solve the problem the simplest way possible, while planning for future extensions / design iterations …
- Sometimes focus effort on reusable infrastructure.
- Use UML to model visually before coding, sometimes with throw-away white-board sketches, and sometimes with precise diagrams created inside a modeling tool.
- Master the tools at your disposal: polymorphism, encapsulation, delegation, generalization, UML, ...
- Study design patterns.
- The art of good design is hard to define, but involves creating the simplest solution that works, while being as flexible as possible for predictable future extensions (sometimes planning ahead for future refactorings without actually doing them right away) and always trying to balance the various design, personnel and economic forces.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.