The meaning of OO, part 2?

Comp-304 : The meaning of OO, part 2
Lecture 6

Alexandre Denault
Original notes by Hans Vangheluwe
Computer Science
McGill University
Fall 2006
Changes to Assignment 1

■ Task 2
  • Fix any bug I might have inserted in the code. These bugs are typos, usually involving one or two characters. If a method does something mathematically impossible (division by zero, for example), it should throw an ArithmeticError exception. You will need to add those checks and test for them.

■ Task 3
  • Implement the following functions in the Vector class: dotProduct, unit. Also, implement the equals function in the Force and Mass classes. You can use the unit tests from Task 1 to help you implement these functions (as done in XP Programming). You don't need to implement the crossProduct method in the Vector class.
v1 = vector(2,2)
f1 = force(v1, 5, 10)

f1.getMagnitudeAtTime(0) -> (0,0)
f1.getMagnitudeAtTime(1) -> (0,0)
f1.getMagnitudeAtTime(4) -> (0,0)
f1.getMagnitudeAtTime(5) -> (2,2)
f1.getMagnitudeAtTime(6) -> (2,2)
f1.getMagnitudeAtTime(9) -> (2,2)
f1.getMagnitudeAtTime(10) -> (2,2)
f1.getMagnitudeAtTime(11) -> (0,0)
Assignment 1 : Force Object

\[ v_1 = \text{vector}(2,2) \]
\[ v_2 = \text{vector}(1,1) \]
\[ f_1 = \text{force}(v_1, 5, 10).\text{add}(\text{force}(v_2, 1, 6)) \]

\[ f_1.\text{getMagnitudeAtTime}(0) \rightarrow (0,0) \]
\[ f_1.\text{getMagnitudeAtTime}(1) \rightarrow (1,1) \]
\[ f_1.\text{getMagnitudeAtTime}(4) \rightarrow (1,1) \]
\[ f_1.\text{getMagnitudeAtTime}(5) \rightarrow (3,3) \]
\[ f_1.\text{getMagnitudeAtTime}(6) \rightarrow (3,3) \]
\[ f_1.\text{getMagnitudeAtTime}(7) \rightarrow (2,2) \]
\[ f_1.\text{getMagnitudeAtTime}(9) \rightarrow (2,2) \]
\[ f_1.\text{getMagnitudeAtTime}(10) \rightarrow (2,2) \]
\[ f_1.\text{getMagnitudeAtTime}(11) \rightarrow (0,0) \]
1) Encapsulated
2) State Retention
3) Implementation / Information Hiding
4) Object Identity
5) Messages
6) Classes
7) Inheritance
8) Polymorphism
9) Generacity
<table>
<thead>
<tr>
<th></th>
<th>Replication Strategy</th>
<th>Interest Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zone-based</td>
<td>Zone-based Replication Strategy</td>
<td>Zone-based Interest Management</td>
</tr>
<tr>
<td>Tile-based</td>
<td>Tile-based Replication Strategy</td>
<td>Tile-based Interest Management</td>
</tr>
</tbody>
</table>
When observing an encapsulation, we can have two point of view:
  • From the outside (public view)
  • From the inside (private view)

The advantages of a good encapsulation is the separation of the private and public views.

To access elements in the private view, users must go through the public interface.
  • Use of encapsulation to restrict internal workings of software from external user view
Information vs Implementation

Information Hiding

- We restrict user from seeing information
  - variables, attributes, data, etc.

- To access information, users must use a set of public methods.

Implementation Hiding

- We restrict user from seeing implementation
  - code, operations, methods, etc.

- Users can use the method without knowledge of their working.
Why should we do this?

- Designer and user must agree on some interface, and nothing else. They are independent. They do not need to speak the same language.
- Software evolution is easier. Suppose user knows about implementation and relies on it. Later, if the designer changes the implementation, the software will break.
- Code re-use is high.
- Abstraction from user is high, user need not worry about how it works!
Never allow other class to directly access your attribute.
Once an attribute is public, it can never be changed.
  * Ex: `img.pixeldData`
Make your attributes available using get/set methods.
  * `this.connectionStatus` Bad!
  * `this.getConnectionStatus()` Good!
public interface Point {
    public set(int x, int y);
    public int getX();
    public int getY();
}

Inside, point could be using Cartesian or Polar coordinates.

- Cartesian coordinates are more efficient when dealing with lots of translations.
- Polar coordinates are more efficient when dealing with lots of rotations.
public interface NetworkClient {
    public connect(String address);
    public void send(Object obj);
    public Object receive();
    public void close();
}

- This kind of network interface can be implemented using multiple protocol.
- The user doesn't even need to know which underlying protocol is used.
- Each object can be identified and treated as a distinct entity.
- Use **unique** names, labels, handles, references and / or object identifiers to distinguish objects. This unique identifier remains with the object for it's **whole life**.
- We cannot use objects' states to distinguish objects, since two distinct objects may have the same state (i.e. same attribute values).
Distinct Identity

Memory Heap

- Player
  - Loc: 4,5
  - 3897894

- Ghost
  - Color: Blue
  - 678567

- Square
  - Type: Wall
  - 984323

- Square
  - Type: Wall
  - 4224534

Variable
Player pacman
Mutable vs Immutable Objects

- An Immutable object is an object that is created once and is never changed.
  - String, Long, etc.
  - Two Immutable objects are considered the same if they have the same state.

- A Mutable object is an object who's state can change.
  - Vector, Array, etc.
  - Two different Mutable objects are never considered the same (different identity).
• **Sender** object (o1) uses messages to demand **target** object (o2) to apply one of o2's methods

• For o1 to send a meaningful message to o2, it must adhere to some **message structure**
  • o1 must know o2's unique identifier
  • o1 must know name of o2's method it wants to call
  • o1 must supply any arguments to o2 so that the method may execute properly

• i.e. in Java, we write o2.method(args)
In “pre-OO” language, we might have written `method(o2, args)`. Why is this not good?

- This doesn't allow polymorphism!
- For o1's message to properly execute o2's method, o1 must
  - know the signature of o2's method
  - pass the proper arguments (inputs)
  - know if the method will return any values (outputs) and be ready to store them accordingly
Three types of messages:

- Informative: supplies target object with information to update its attribute(s) [i.e. o2.setx(5)]
- Interrogative: asks target object to supply information about its attribute(s) [i.e. o2.getx()]
- Imperative: tells target object to do some action [i.e. o2.moveNorth()]
Informative, Interrogative or Imperative?

- `ghost.up()`?
- `grid.insertPlayer(pacman, square)`
- `square.isWall()`?
- `pacman.collectPellet()`
- `ghost.isScared()`?
- `square.addItem(pellet)`
Synchronous vs Asynchronous

Synchronous Messaging
- An object receiving a request executes it immediately and returns the result.

Asynchronous Messaging
- A object receiving a request acknowledges it.
- The request is executed latter and the return value is eventually returned (often through the use of a call-back method)