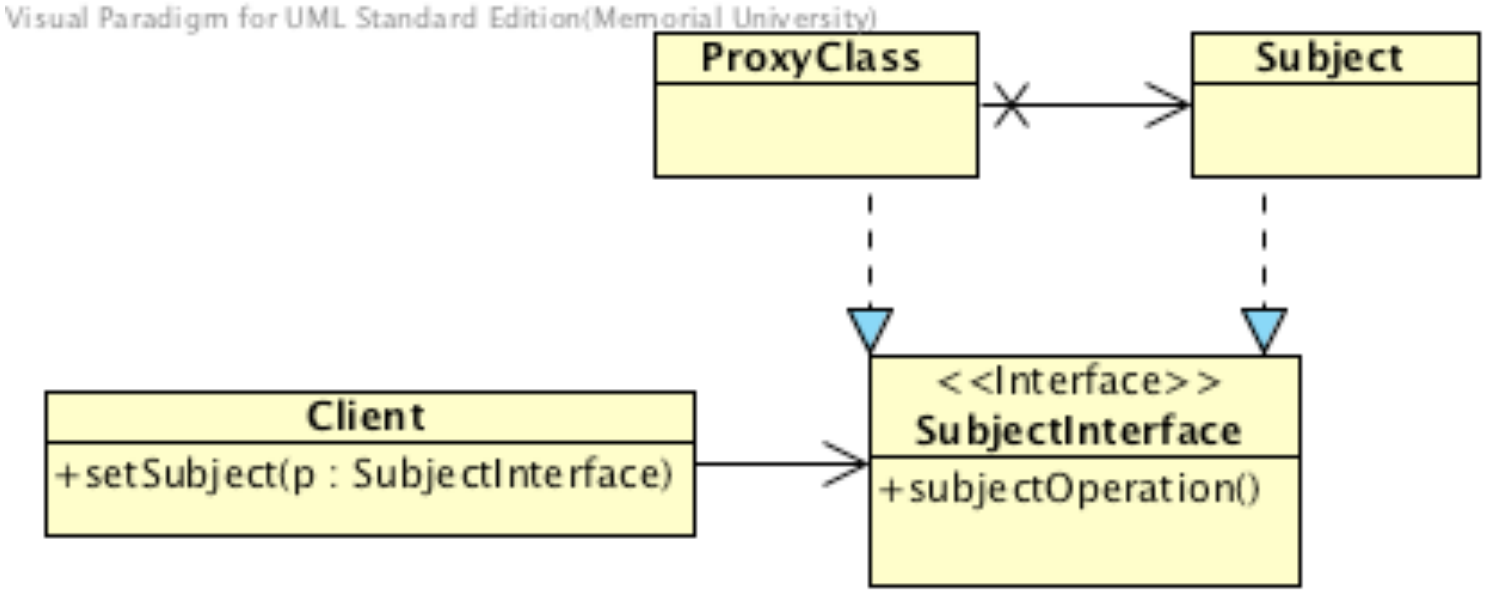
# Remote Method Invocation (RMI) and Distributed Observers in Java

Theodore Norvell

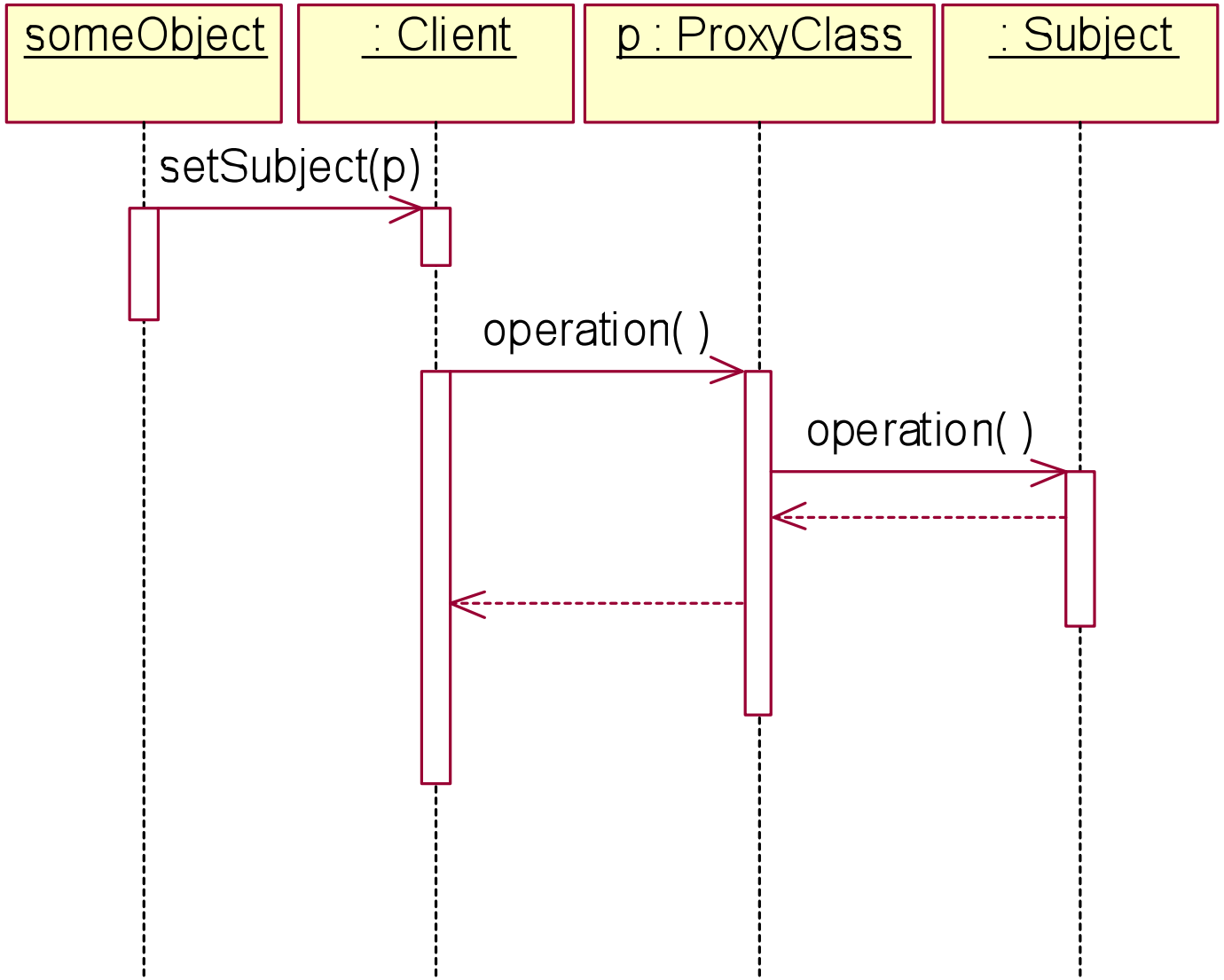
# The Proxy Pattern




- The Client object uses its subject via an interface
- Thus it may be used with a real subject or with a proxy object which represents the subject.

# The Proxy Pattern

- The client calls the proxy, which
- forwards the call (somehow) to the actual subject.



# RMI and the Proxy pattern

- RMI uses the Proxy pattern to distribute objects across a network.
- Recall that in the Proxy pattern a proxy and a subject share a common interface.
- In RMI, objects call methods in a proxy (aka stub) on its own machine
  - The proxy sends messages across the network to a “skeleton” object
  - The skeleton calls the subject object.

# One Remote Method Call.

(0) Client calls stub

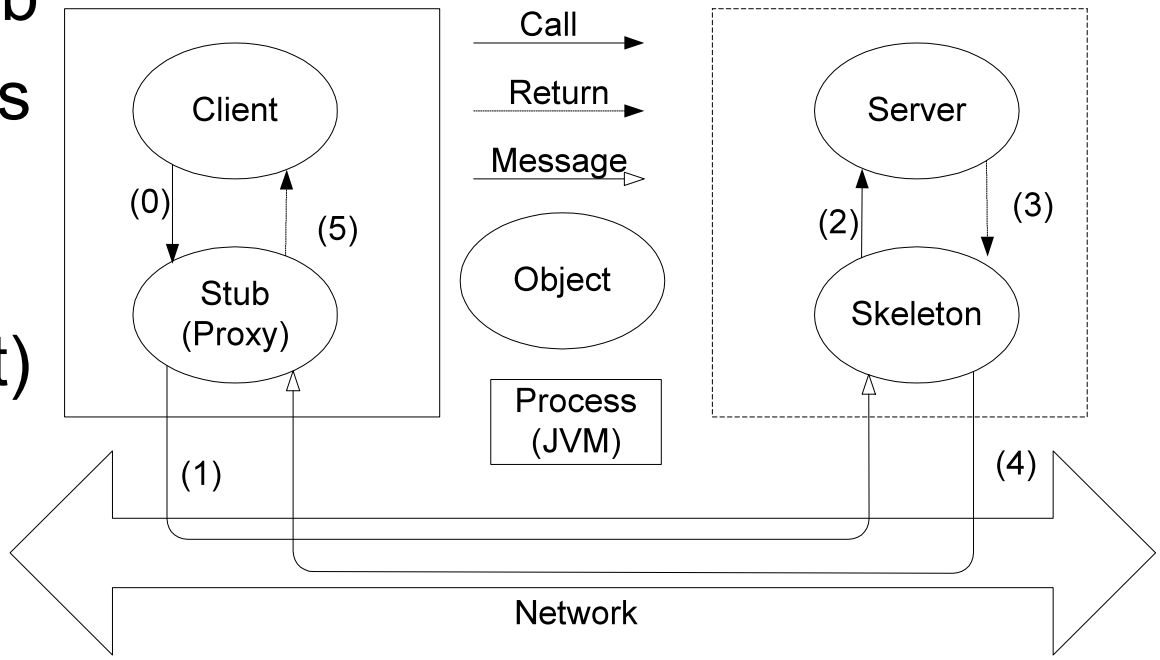
(1) Stub messages skeleton

(2) Skeleton calls server (subject)

(3) Call returns

(4) Skeleton messages proxy

(5) Call returns



# Full disclosure

- Java no longer uses skeleton objects that are paired one-to-one with the server objects.
- I'll include them in the pictures to represent whatever set of objects is listening for net traffic and capable of sending messages to the true server object.

# Issues

- Concurrency
  - If there are multiple clients, the server may field multiple calls at the same time.
    - So use *synchronization* as appropriate.
- Argument passing
  - Arguments are passed by value or “by proxy” not by reference.
- Proxy generation
  - Proxy classes are automatically derived from the server class’s interface.
- Lookup
  - Objects are usually found via a registry (program *rmiregistry*)

# Nitty-Gritty

- The proxy and the server share an interface.
  - This interface must extend java.rmi.Remote.
  - Every method in the interface should be declared to throw java.rmi.RemoteException
  - RemoteExceptions are thrown when network problems are encountered,
  - or when server objects no longer exist.
- The server typically extends class java.rmi.server.UnicastRemoteObject
  - The constructor of this class throws a RemoteException
  - Therefore, so should the constructor of any specialization.

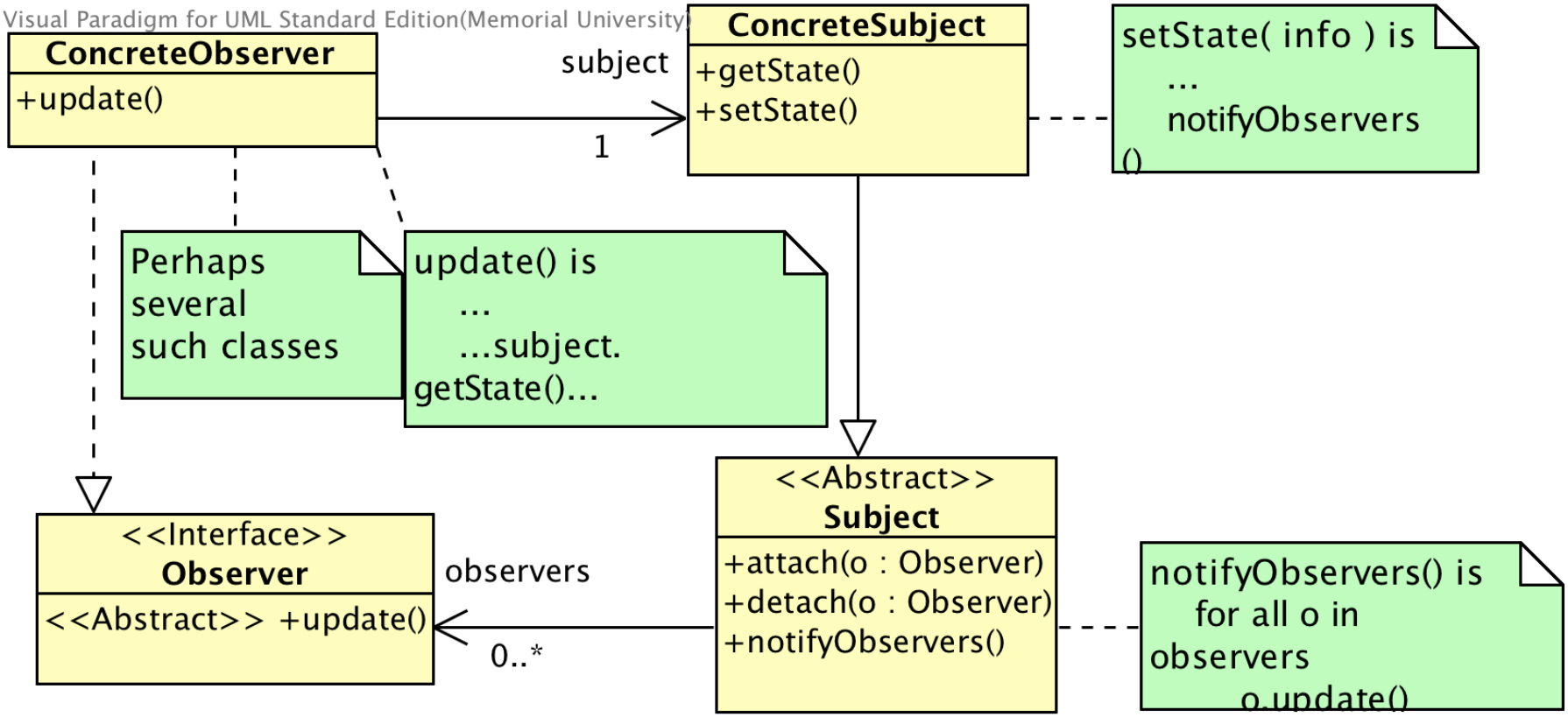
# Argument Passing Revisited

- Most arguments and results
  - are converted to a sequence of bytes sent over the net
  - therefore the class should implement the java.io.Serializable interface
  - a clone of the argument/result is built on the other side.
  - Pass by object value, rather than by object reference.
- But
  - objects that extend java.rmi.server.UnicastRemoteObject
  - instead have a proxy constructed for them on the other side
  - I call this *"pass by proxy"*. *Essentially* pass/return by reference
- So each argument, result, exception type should be
  - a primitive type, implement Serializable, or extend UnicastRemoteObject

# An Example – Distributed Othello

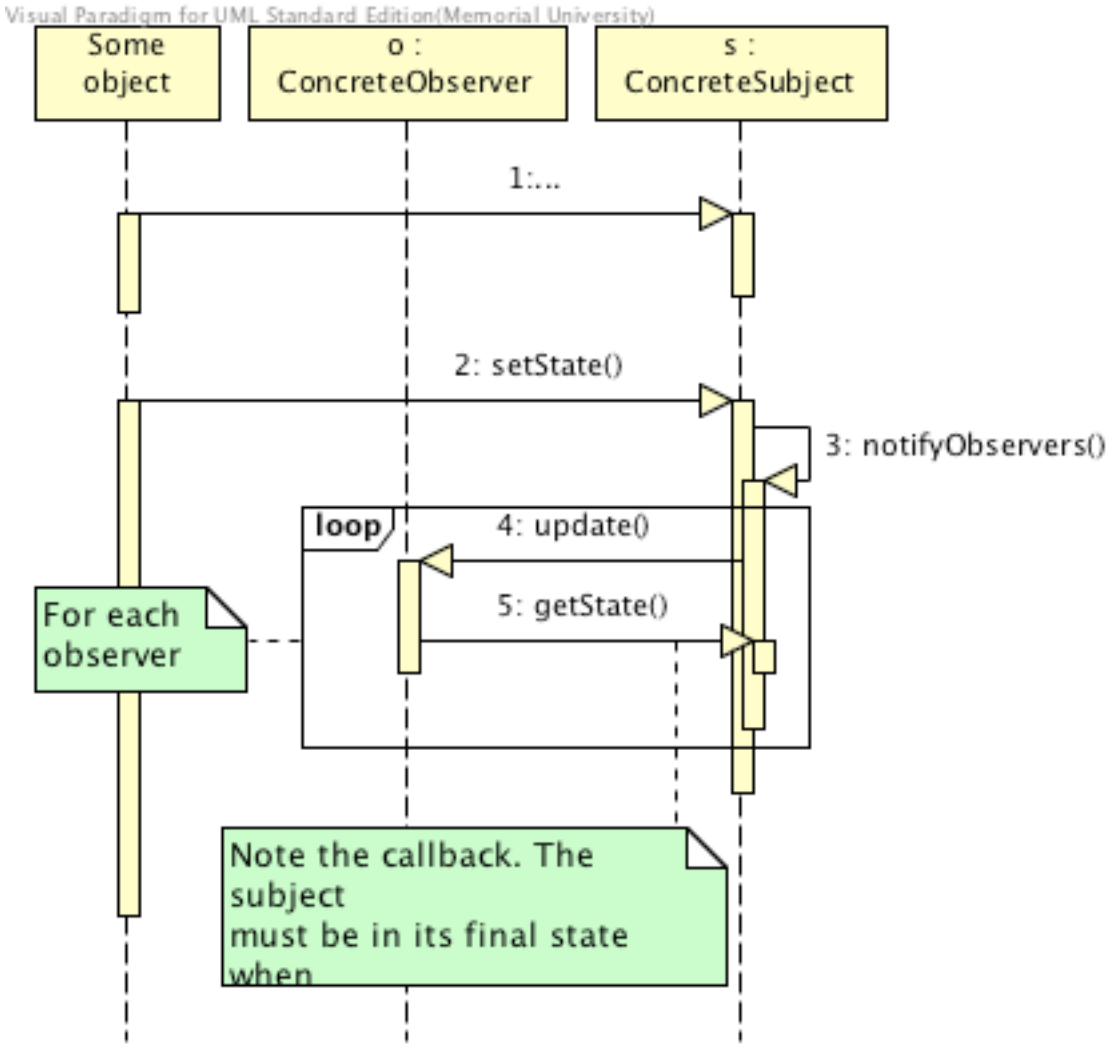
- Othello is a two person board game
- My implementation uses the *observer pattern* so that,
  - when the (game state) model changes,
  - all observers are informed of the change.
- I wanted to put the model on one machine and the observers on other machines.
- Hence I implemented the observer pattern with RMI.
- This is example othello-2 on the website.

# The Observer Pattern

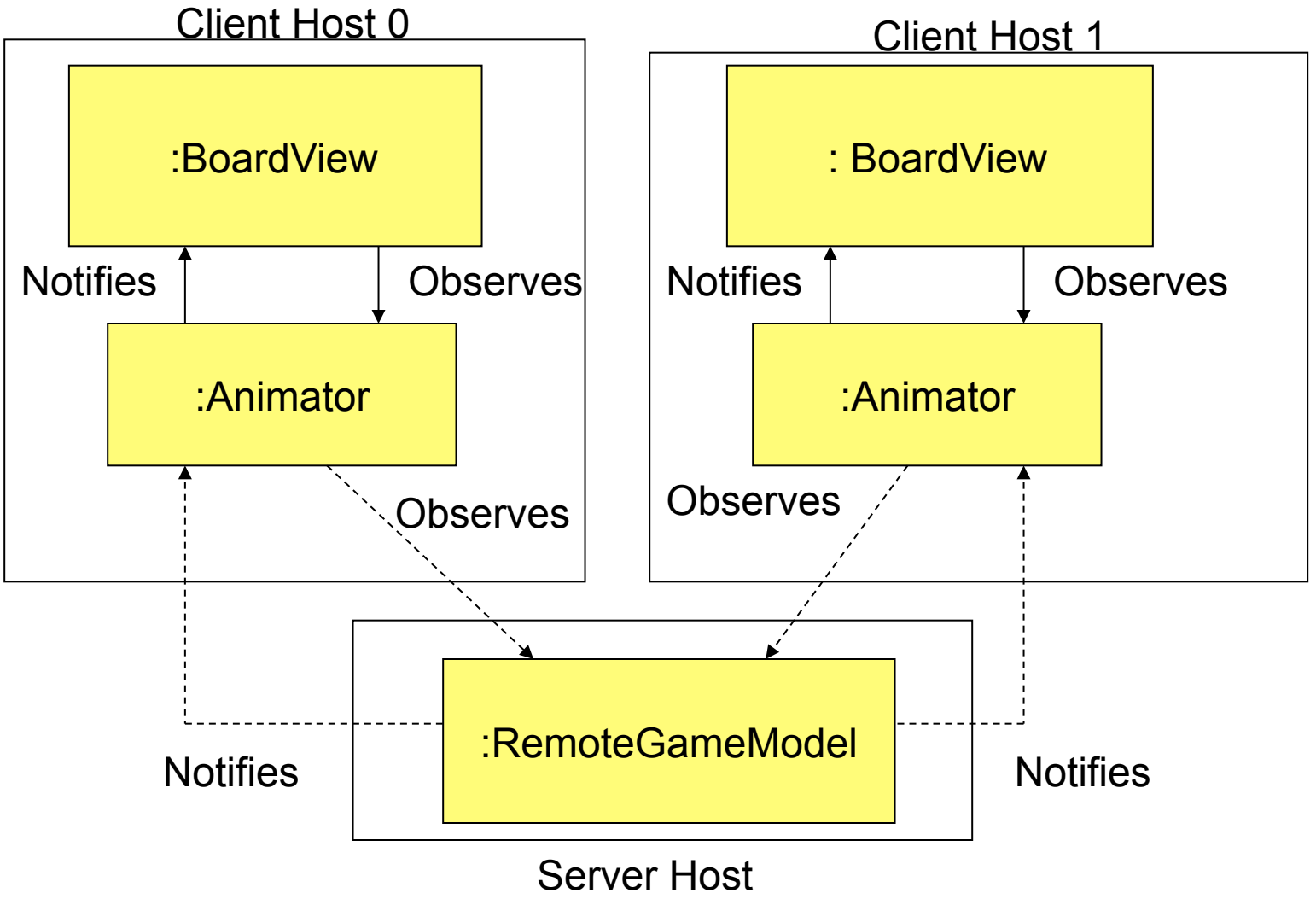



Subject alerts Observers of changes of state.

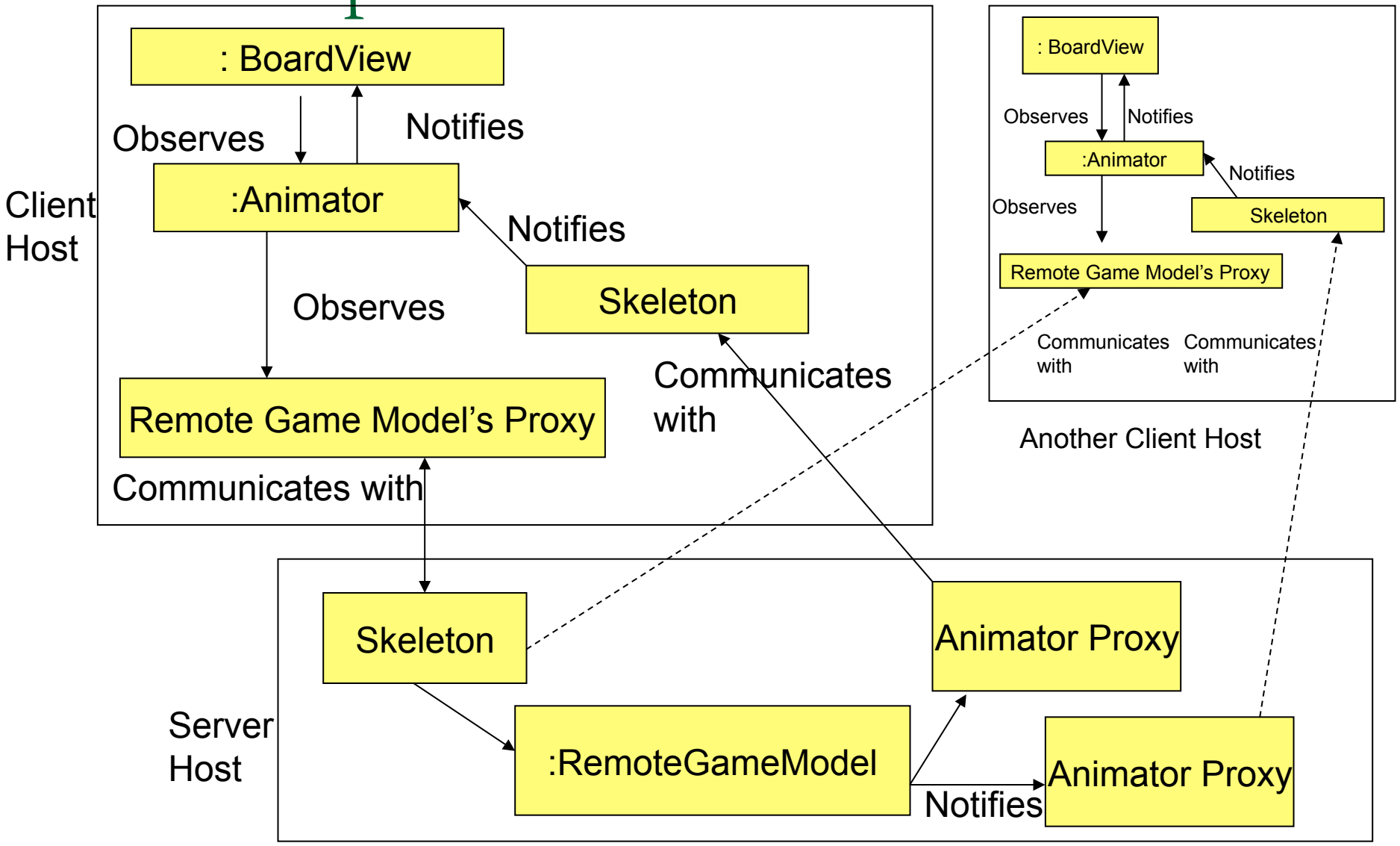
# Observer Pattern



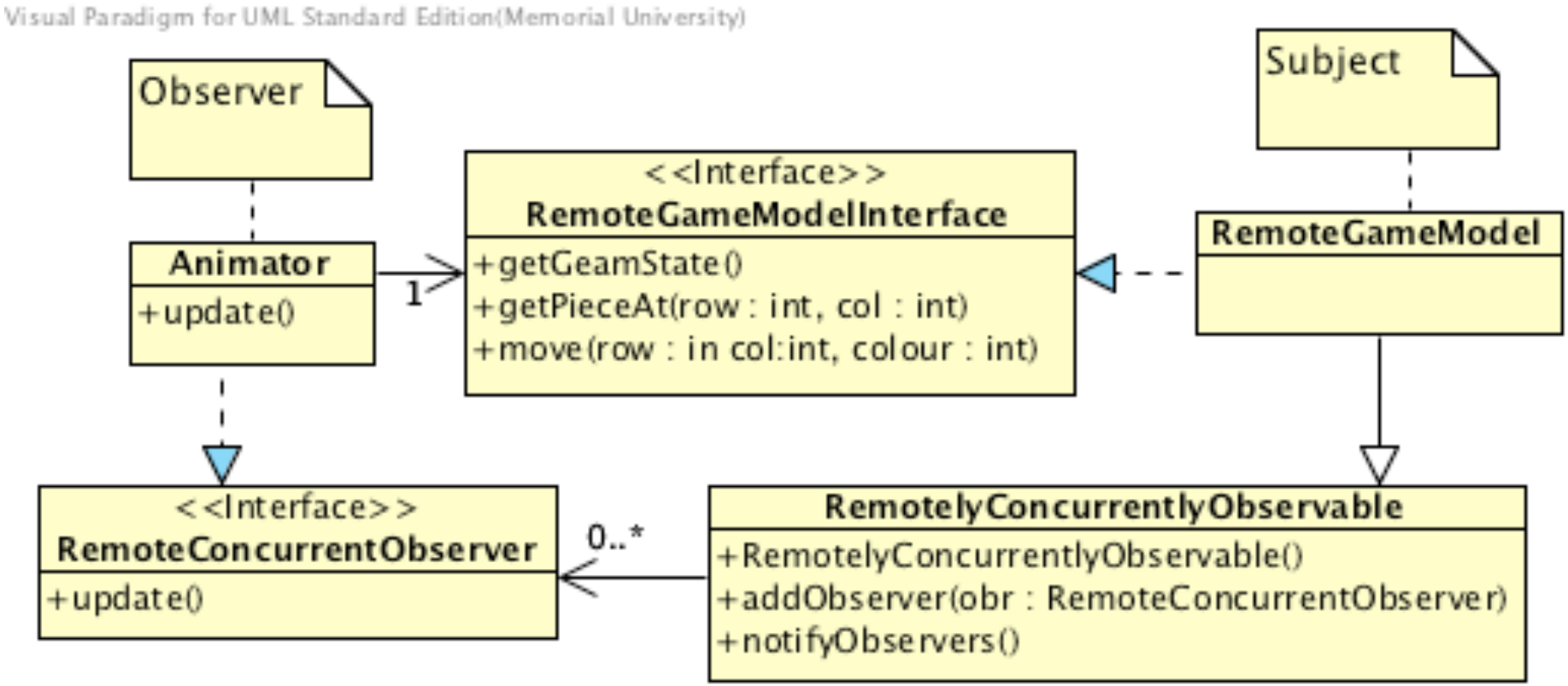
# Object diagram for Othello 2



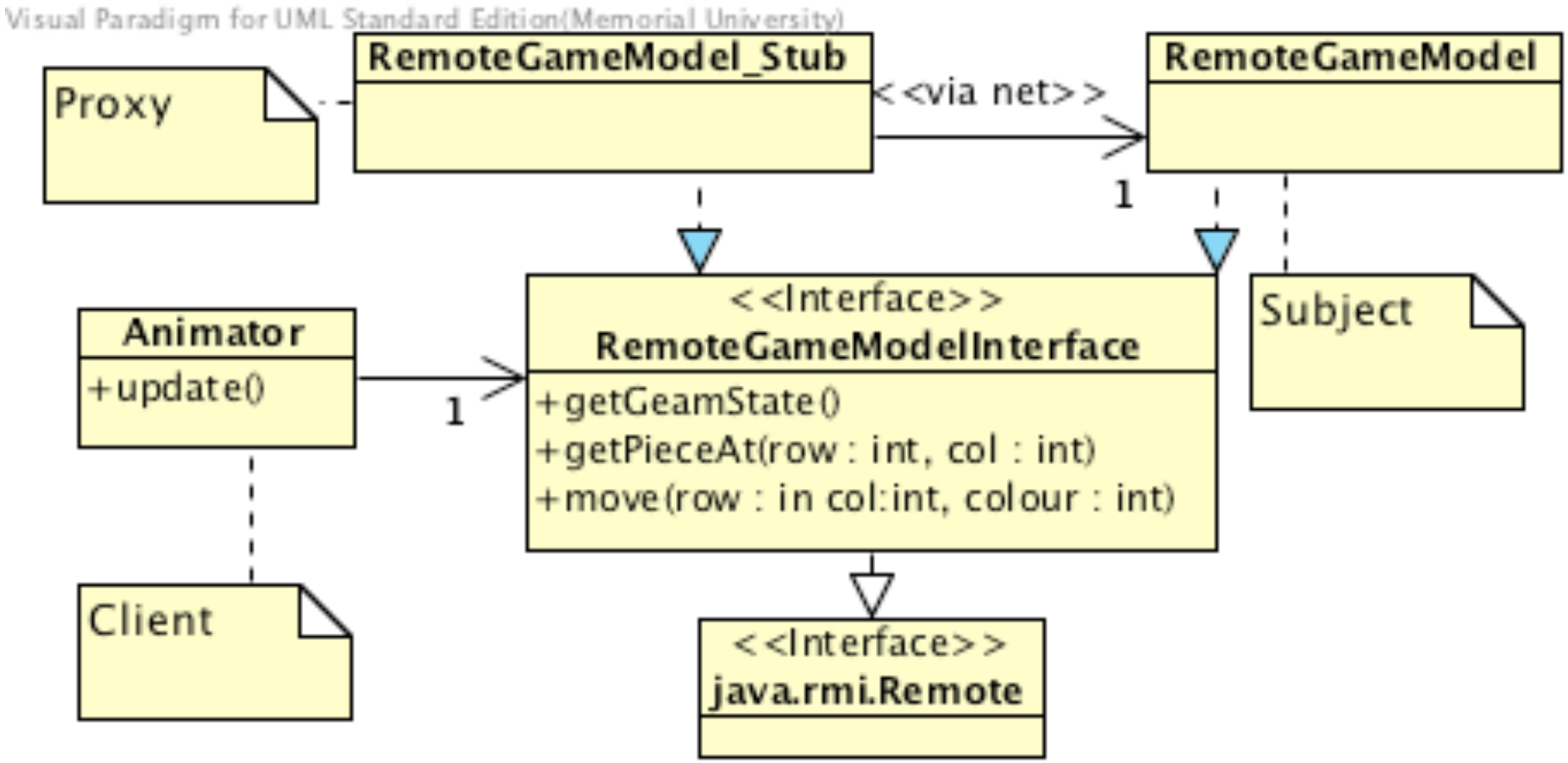
# … with proxies and skeletons



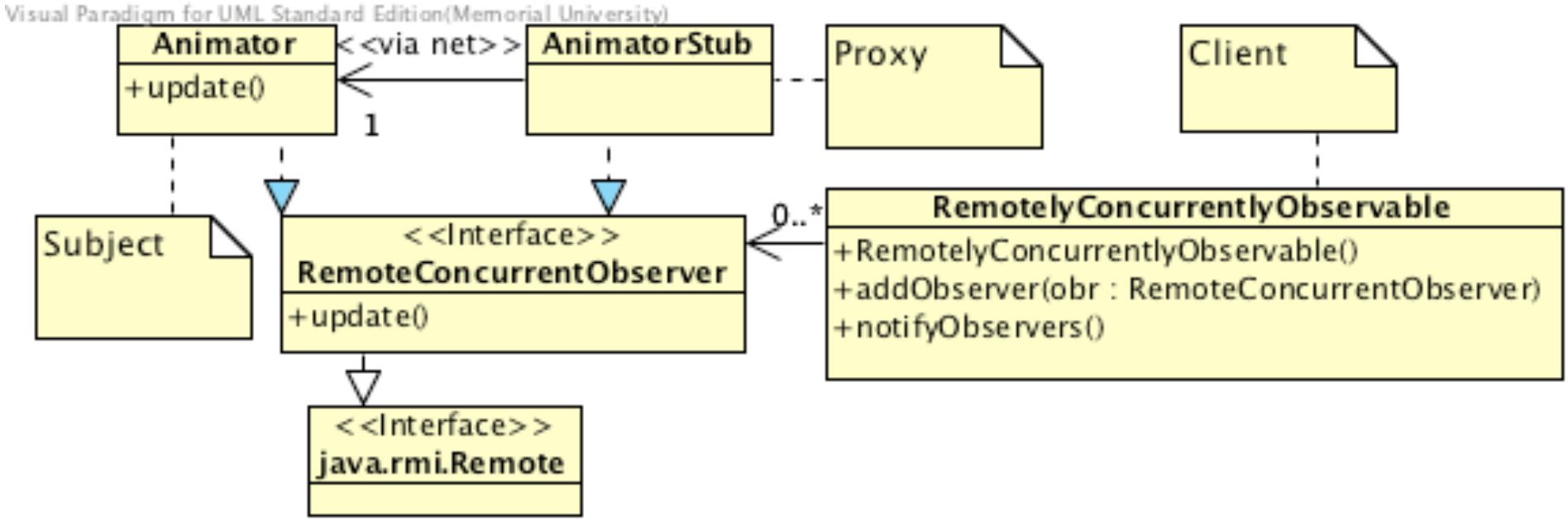
# The Observer Pattern for Othello



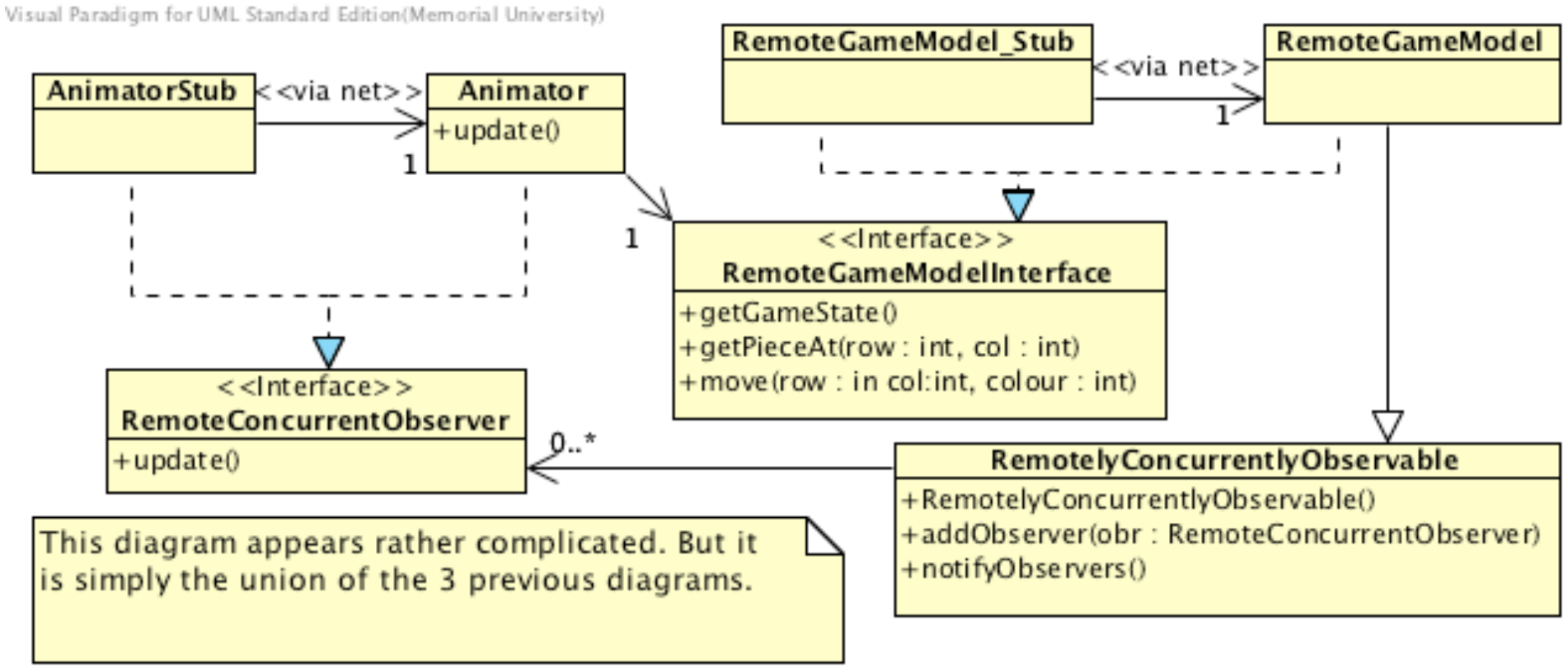
# Proxy for the Remote Game Model



# Proxies for the Animators



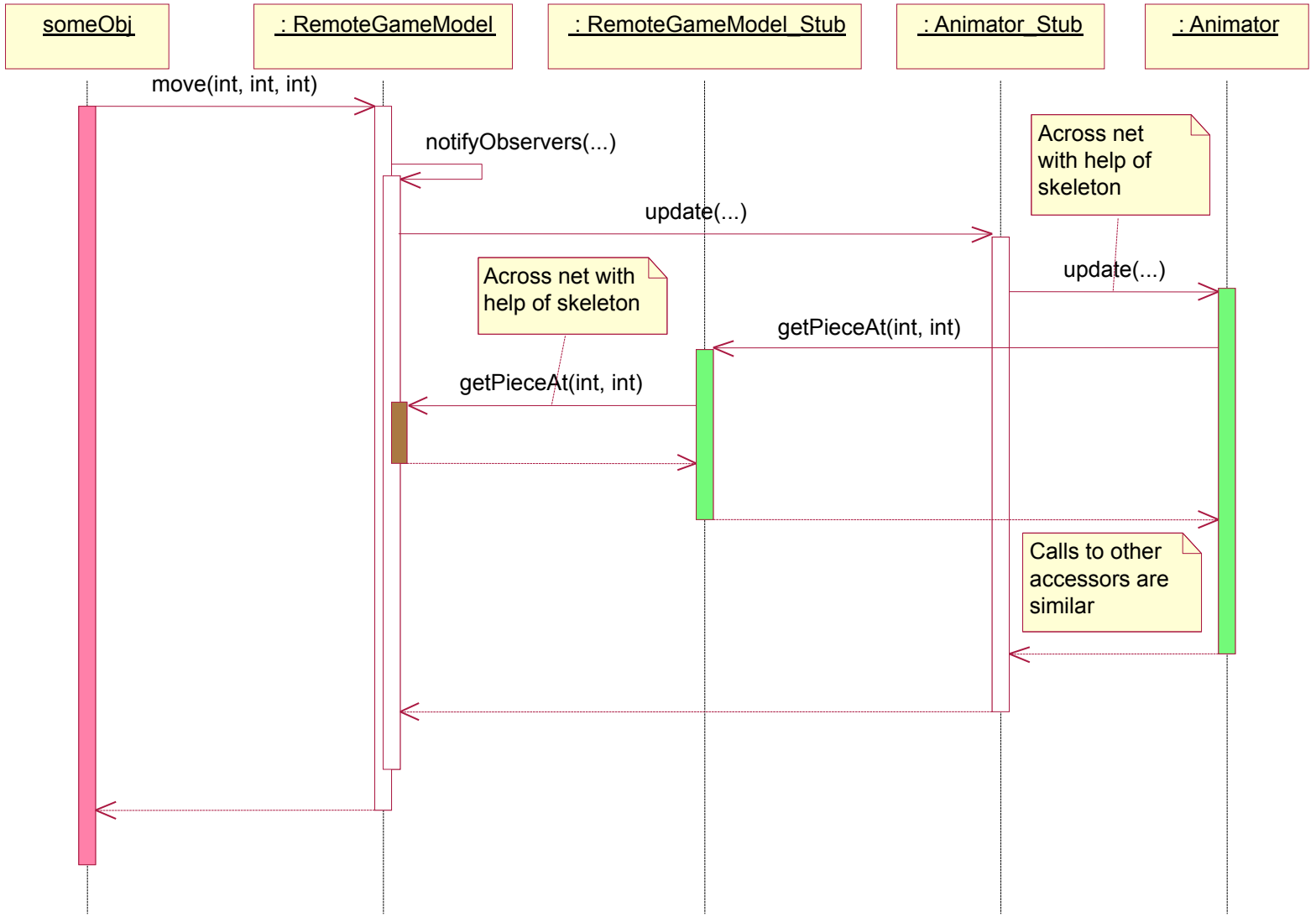
# Animator / RemoteGameModel Relationship



# Typical sequence —slightly simplified

- A remote client calls a mutator on the RemoteGameModel via its stub & skeleton
  - The RemoteGameModel updates its state and notifies each Animator via its stubs & skeletons.
    - The Observers (Animator objects) call the RemoteGameModel accessor “getPieceAt” via its stubs and skeleton.
    - getPieceAt returns
  - The update routines return.
- The original mutator call returns and the RemoteGameModel becomes unlocked.

# Typical sequence —slightly simplified

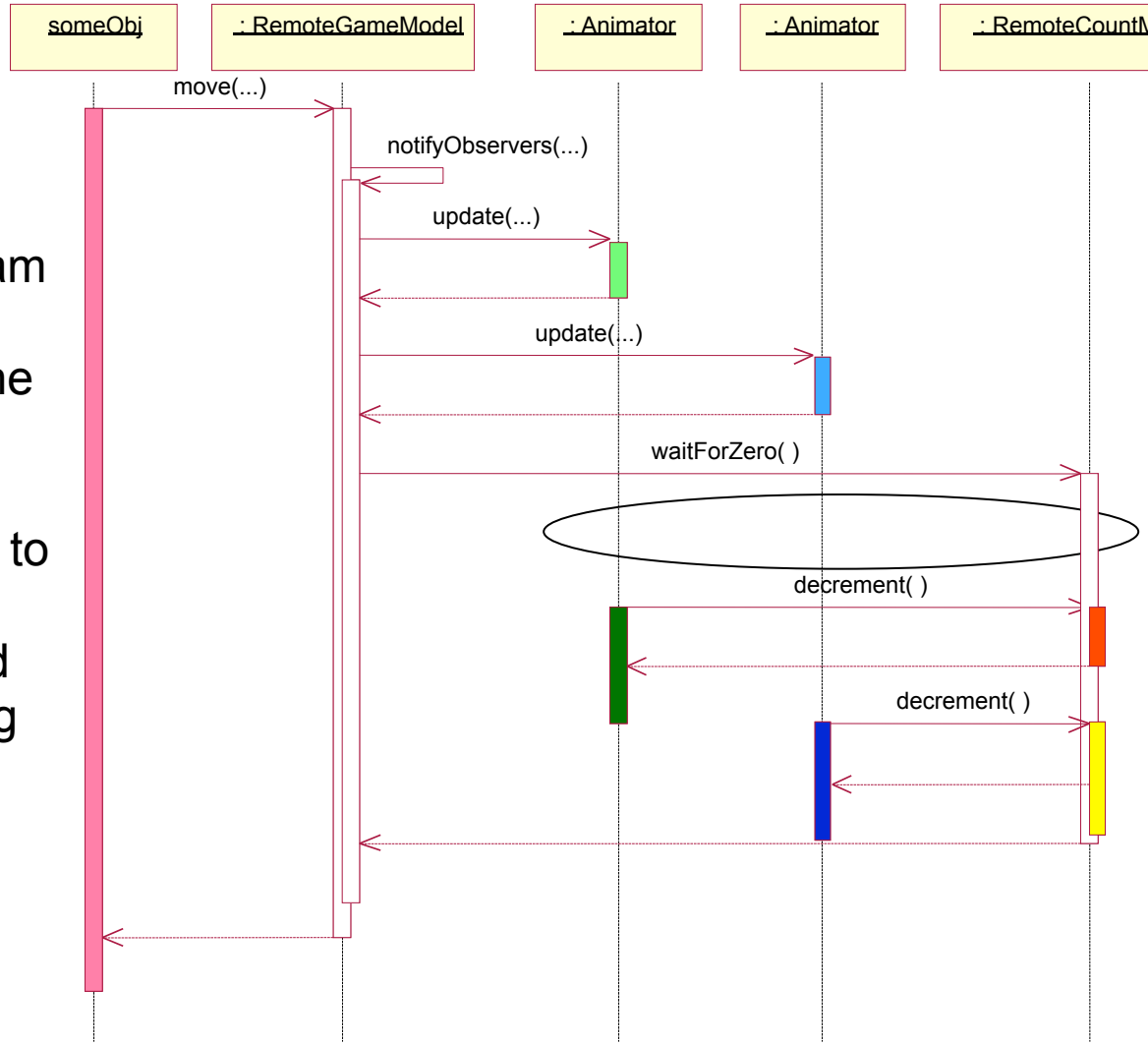


# Concurrent Notification

- The previous sequence is slightly simplified.
  - In fact the Animators start fresh animation threads and return from update almost immediately (so that all can be informed essentially at the same time).
    - The Animation threads will inform the RemoteGameModel of when they have completed their animation.
  - The RemoteGameModel waits until it has been informed that all animations are complete.

# Concurrent Notification

I simplifed this diagram a bit by omitting the callbacks from the animators to the game model and by omitting the stubs and skeletons.




During this time the animators animate (using new threads created in update). The last thing these threads do is call decrement (via RMI). Once all the animation threads have called decrement, the original server thread returns from waitForZero

# A concurrency problem

- Since the RemoteGameModel is accessed by multiple threads, I originally made all public methods **synchronized**.
  - This was a mistake. Look at the sequence diagram a few slides back. Note that the white thread executes “move”, but the brown thread executes the callback to “getPiece”. This caused a deadlock!
- The solution was to use a reader/writer lock
  - This ensured that only one mutator at a time could be executed. But it allows accessors to be executed during the observation phase.
  - See my notes on *Animation with Threads* and the code for RemoteGameModel in othello-2 for details.

# Naming the Server

- The clients must be able to find the server (RemoteGameModel object)
- A Registry is an object that listens on a known port and that associates proxies (stubs) with names.
- The server registers the RemoteGameModel under an known name.
- The client uses a URI of the form

  rmi://*host:port*/*name*

- To ask the registry at *host:port* for a copy of the stub.

# Binding the server to a name.

- The main routine for the server
  - The static method re*bind* in *Naming* gives a URI to *gameModel*

```
public static void main( String[] args )  {
        String name = "gameModel" ;
        int port = 1491 ;
        try {
           // Make the model
           RemoteGameModel gm = new RemoteGameModel();
           // Create a registry
           Registry registry = LocateRegistry.createRegistry(port);
           // Create a proxy and bind it to the name.
           registry.rebind(name, gm ); }
        catch( Throwable  e) { … } }
```

# Binding the observable

1 new RemoteGameModel()

2 LocateRegistry.makeRegistry(1491)

3 registry.rebind( “gameModel”, gm )

Registry object

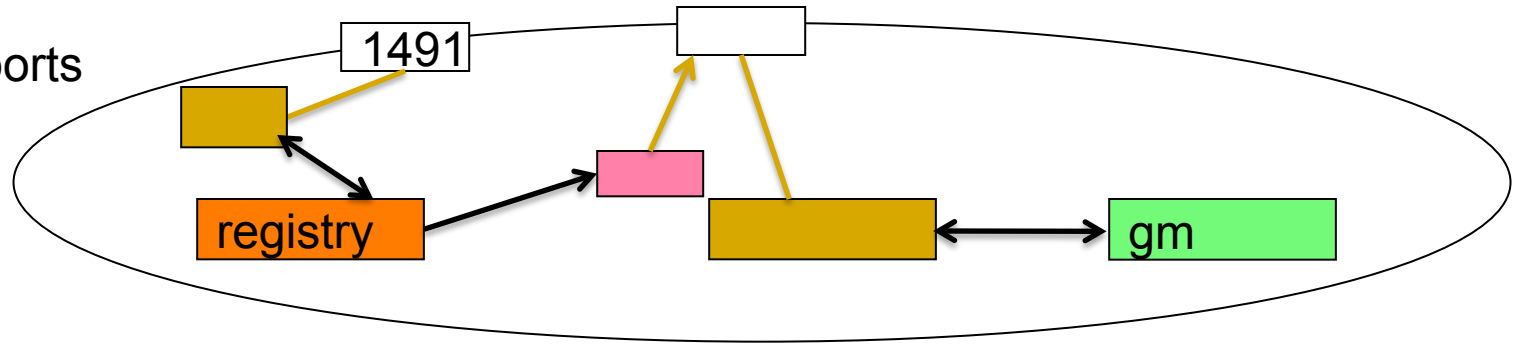
RemoteGameModel object

Proxy objects

Skeleton object

JVMs

ports



# Looking up the server object

- The clients obtain a proxy for the game model using *Naming.lookup( URI )*
- From ClientMain.java

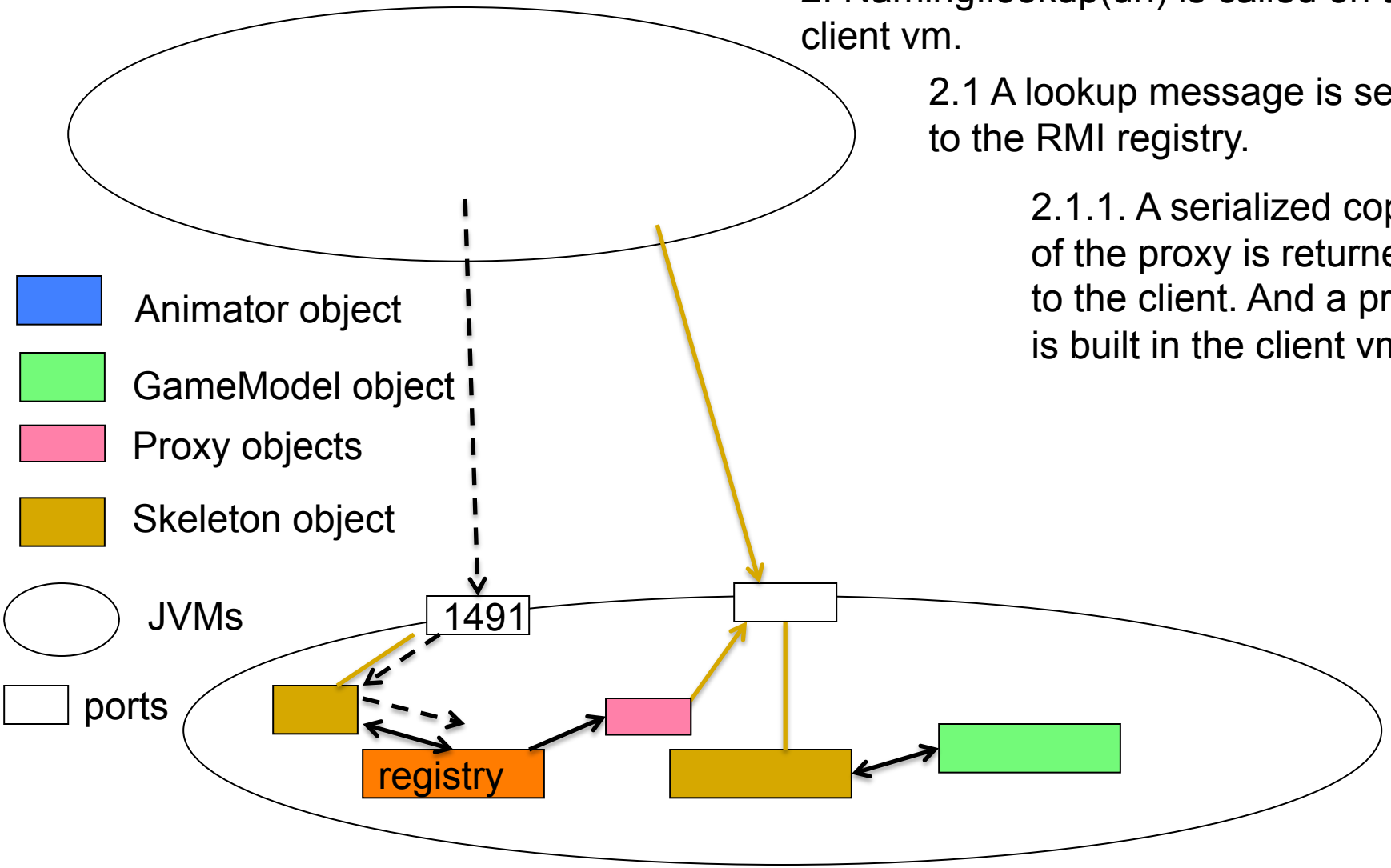
```
public static void main( String[] args) {
        RemoteGameModelInterface proxy = null ;
        String host = … ; // host for server
        try {
            String name = "rmi://" +host+ ":1491/gameModel" ;
            proxy = (RemoteGameModelInterface)
                        Naming.lookup( name ) ; }
        catch( Throwable e) { … }
… continued on the slide after next …
```

# Looking up the server object

2. Naming.lookup(uri) is called on the client vm.

2.1 A lookup message is sent to the RMI registry.

2.1.1. A serialized copy of the proxy is returned to the client. And a proxy is built in the client vm.



# Closing the loop

- The client can then use the proxy. E.g.
  - Continuing the ClientMain main routine

```
…
Animator animator = null ;
try {
        animator = new Animator( proxy ) ; }
catch( RemoteException e ) { …  }
```

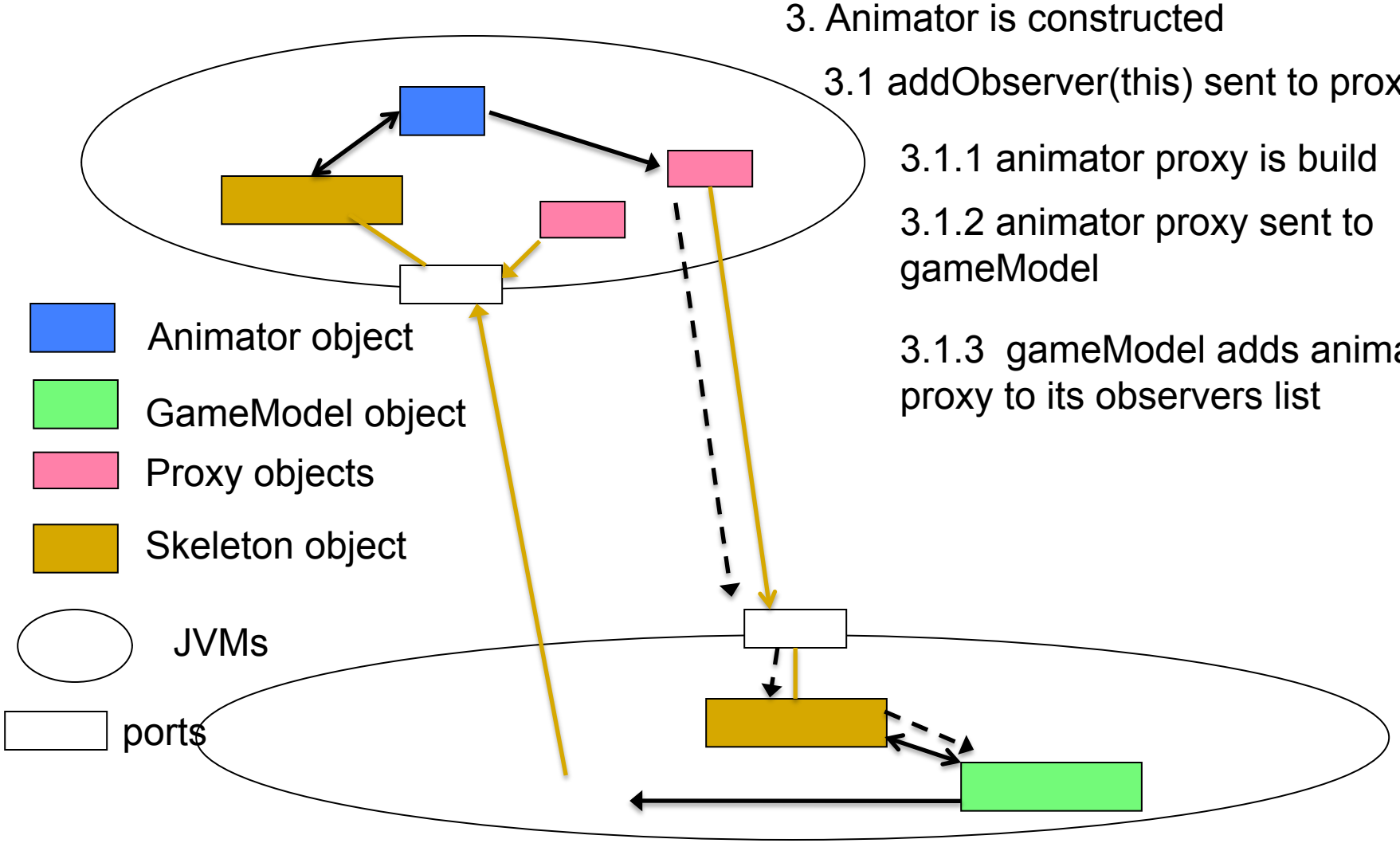
  - The constructor for Animator

```
class Animator extends UnicastRemoteServer {
  public Animator( RemoteGameModelInterface gameModel )
      throws RemoteException {
   this.gameModel = gameModel ;
   gameModel.addObserver( this ); }
```

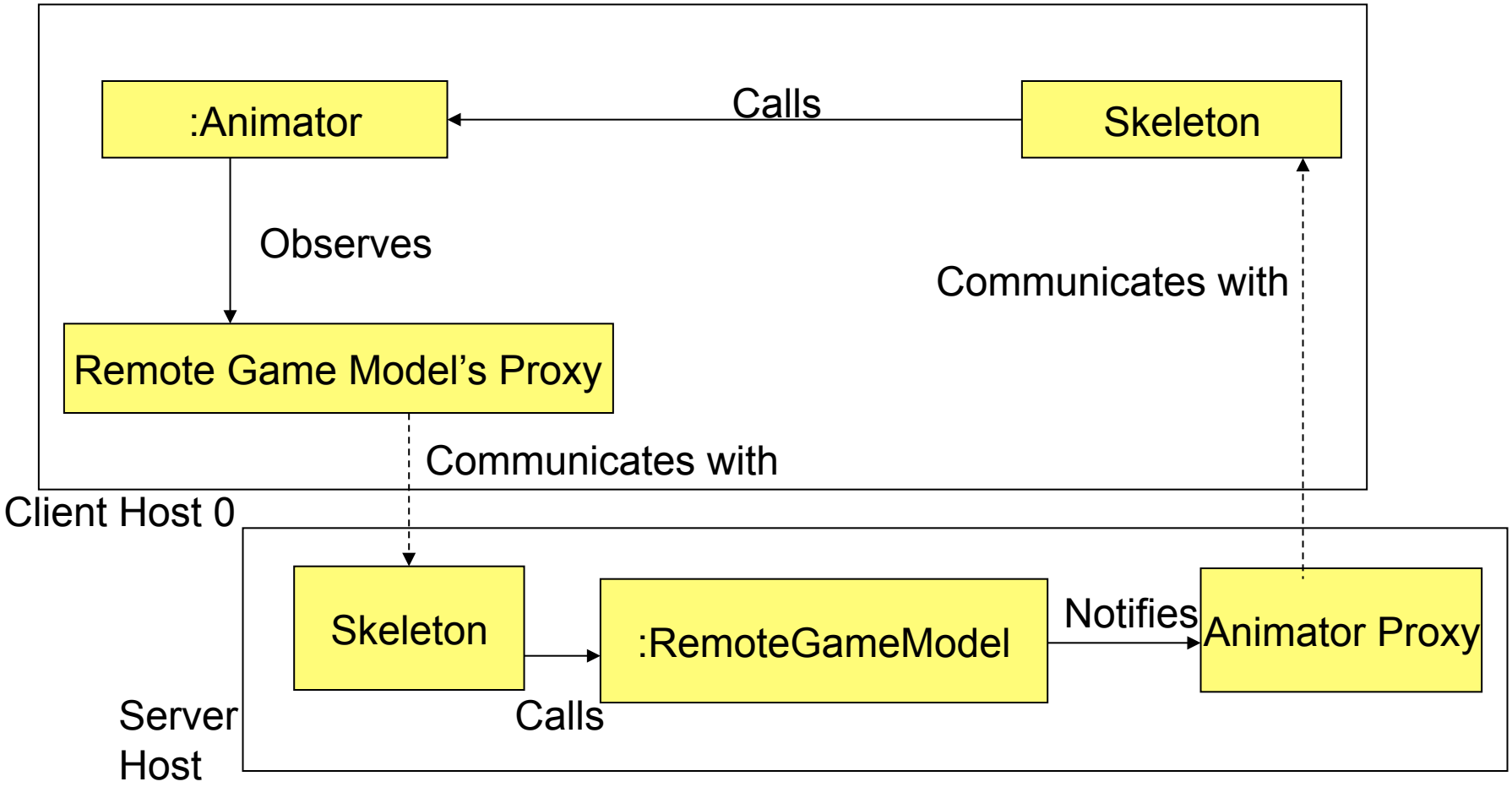
# Adding a remote Observer.

- The Animator calls addObserver(this) on the RemoteGameModel's proxy.
  - Since Animator extends UnicastRemoteServer, it is passed by proxy, meaning
  - a proxy for the Animator is constructed in the JVM of the RemoteGameModel.
  - (see next slide.)

# Creating an observer (animator)



# The final hookup (again)



# Deployment

- Both the server and the client JVMs need to have access to the same .class files.
- An easy way to do this is to put all class files in one jar file that is on both machines.
- The server is started with

  java -cp othello-2.jar othello.ServerMain
- The client is started with

  java -cp othello-2.jar othello.ClientMain
- Source for a complete working game is on the course website as othello-3.

# A Few Words of Warning

- RMI makes it seductively easy to treat remote objects as if they were local.
- Keep in mind
  - Partial Failure
    - Part of the system of objects may fail
    - Partial failures may be intermittent
    - Network delays
    - On a large network, delays are indistinguishable from failures
      - In the Othello example, failure was not considered. The system is **not** designed to be resilient to partial failures.

# A Few Words of Warning (cont.)

- Keep in mind (cont.)
  - Performance
    - Remote calls are several orders of magnitude more expensive than local calls (100,000 or more to 1)
      - E.g. in the Othello example, this motivated splitting the model into local (Animator) and remote (RemoteGameModel) copies.
  - Concurrency
    - Remote calls introduce threads that may not be in a nondistributed system.
      - E.g. in the remote observer pattern, callbacks from the observer to the observable will be on a thread different from the one that calls “update”.

# A Few Words of Warning (cont.)

- Keep in mind (cont.)
  - Semantics changes
    - In Java, local calls pass objects by pointer value.
    - Remote calls pass objects either by copy or by copying a proxy.
      - E.g. in the Othello game as I converted from the nondistributed to a distributed version, the semantics of some calls changed, even though I did not change the source code for those calls.