Thread-Based Animation and Concurrent Observation

Here we look at two problems

- First: How can we use threads to create animation effects?
- Second: How can multiple threads concurrently observe changes in the same object?
- Solutions to both are found in my Othello-0, Othello-1, Othello-2, and Othello-3 projects.
  - Othello-0 explores the these problem in one computer.
  - Othello-1 explores the second question

First Problem

- How can we create animation effects using threads?

Using Threads for Animation

- Assumptions: View objects are JComponents that know a Model object.
- The view's paintComponent method consults the Model.
- Animation accomplished by starting a thread that repeatedly:
  - Changes the model
  - Calls repaint() on the view object.
  - sleeps for a short time (< 40ms is good)

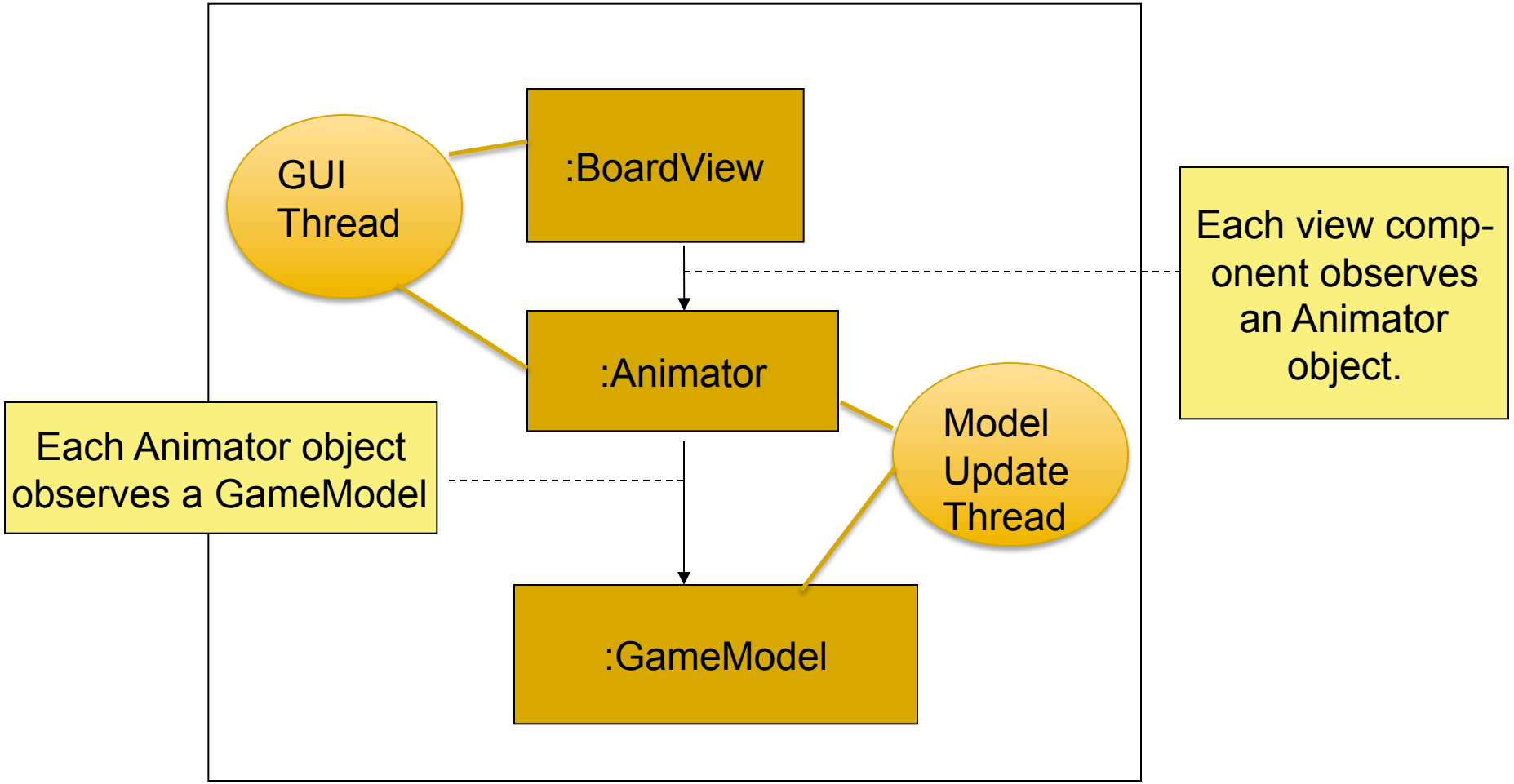
Animation In Othello-0

- The “model” (for animation purposes) is a class called Animator.
- The “view” is a class called BoardView.
- I used the observer pattern so each BoardView observes one Animator.
- Animation is done by making a small change to the Animator, notifying the observers, and waiting a little while.

Animation In the Othello game

- Aims
  - Minimize dependence.
  - Keep model simple.
  - Prepare for distributed (internet) version.
- Method
  - Split the model into two parts
    - One part (GameModel) knows nothing about animation.
    - The other part (Animator) deals only with animation.
  - Observer pattern is used to reduce dependence.

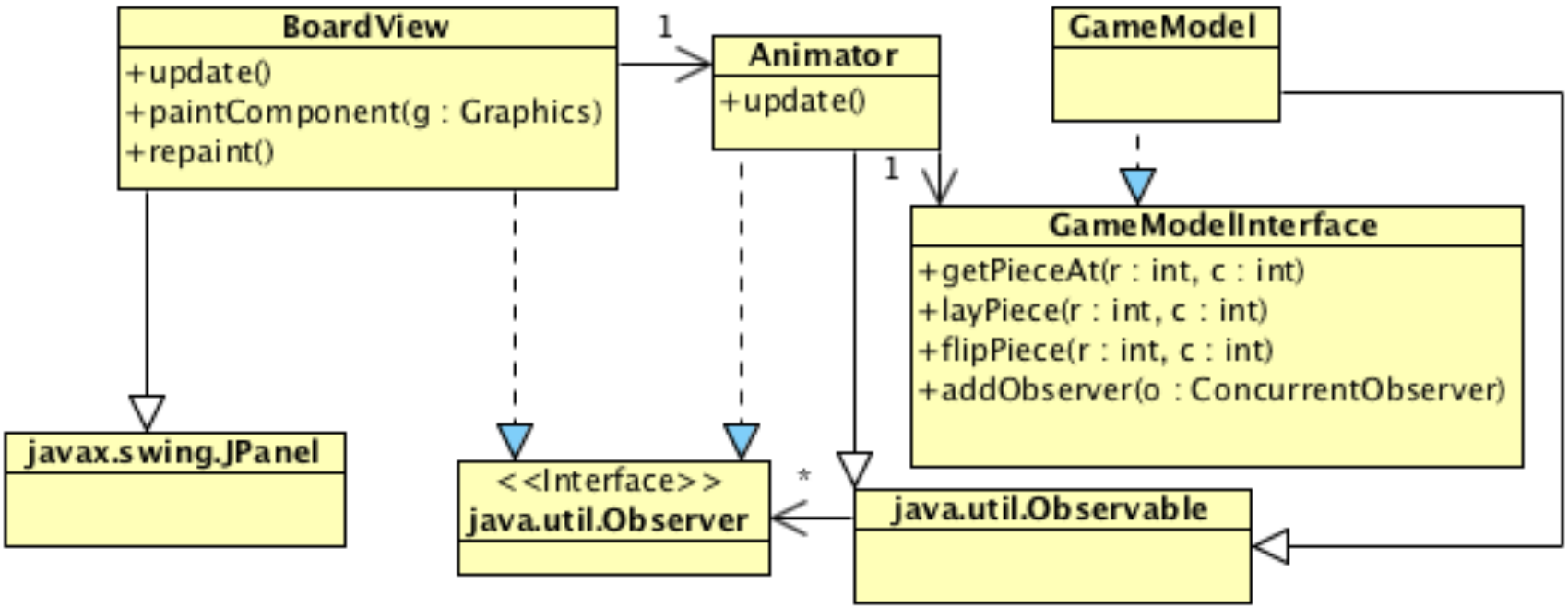
The Othello 0 system



Major classes in the design

- BoardView extends JPanel
  - Can draw a static image.
  - Handles world-to-view transformation.
  - *observes* an Animator object
- Animator
  - Provides a model in which pieces may be flying or flipping.
  - *observes* a GameModelInterface object
  - Animates changes in the GameModel
- GameModel
  - represents state of the board
  - implements GameModelInterface

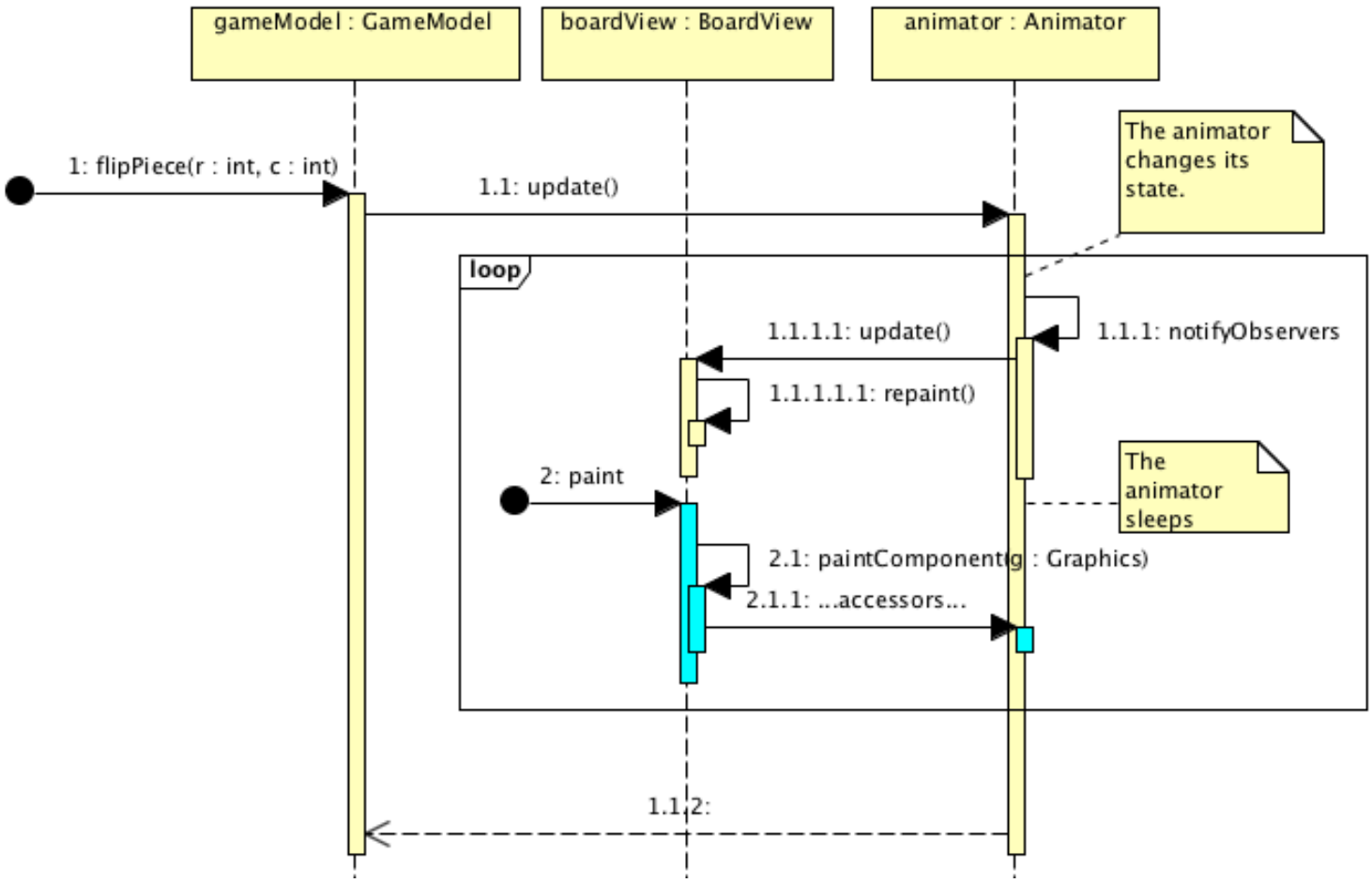
Classes


BoardView
+update()
+paintComponent(g : Graphics)
+repaint()
1
Animator
+update()
GameModel
1
GameModelInterface
+getPieceAt(r : int, c : int)
+layPiece(r : int, c : int)
+flipPiece(r : int, c : int)
+addObserver(o : ConcurrentObserver)
javax.swing.JPanel
<<Interface>>
java.util.Observer
*
java.util.Observable

Animation In Othello game

- Animator: a closer look
  - A copy of the model that includes information about the state of animated objects.
  - When the Animator object is alerted of a change in the GameModel it
    - slowly changes the Animator object's state to match the model state, while
    - notifying listeners (views)
  - The Animator is thus shared by the updating thread and the GUI thread.
    - So synchronization is used to ensure safety.

Updates


Update thread
GUI thread
gameModel : GameModel
boardView : BoardView
animator : Animator
1: flipPiece(r : int, c : int)
1.1: update()
The animator changes its state.
loop
1.1.1.1: update()
1.1.1: notifyObservers
1.1.1.1.1: repaint()
2: paint
The animator sleeps
2.1: paintComponent(g : Graphics)
2.1.1: ...accessors...
1.1.2:

From the Animator Class

- When notified of a piece being flipped:

```
private void animateFlip( int row, int col, int newColour ) {
  // Enter the flipping state.
   synchronized( this ) {
      state = FLIPPING_STATE ;
      this.newColour = newColour ;
      square[ row ][ col ] = EMPTY ;
      flippingRow = row ;
      flippingCol = col ;
      amountOfFlip = 0.0 ; }
   …
```

From the Animator Class (cont.)

```
…
// Flip the piece bit by bit, allowing observation between.
int iterations = 50 ;
double delta =  1 / (double)iterations ;
for( int i=0 ; i<iterations ; ++i ) {
    synchronized( this ) {
        amountOfFlip += delta ;
        setChanged() ; }
    notifyObservers();
    try{ Thread.sleep(20) ; }
    catch( InterruptedException e ) {}
} …
```

From the Animator Class (cont.)

```
…
// Change to the quiet state
synchronized( this ) {
    state = QUIET_STATE ;
    square[ row ][ col ] = newColour ;
    setChanged() ; }
notifyObservers(); }
```

From the BoardView class.

- The `update` method (from the Observer interface) calls repaint

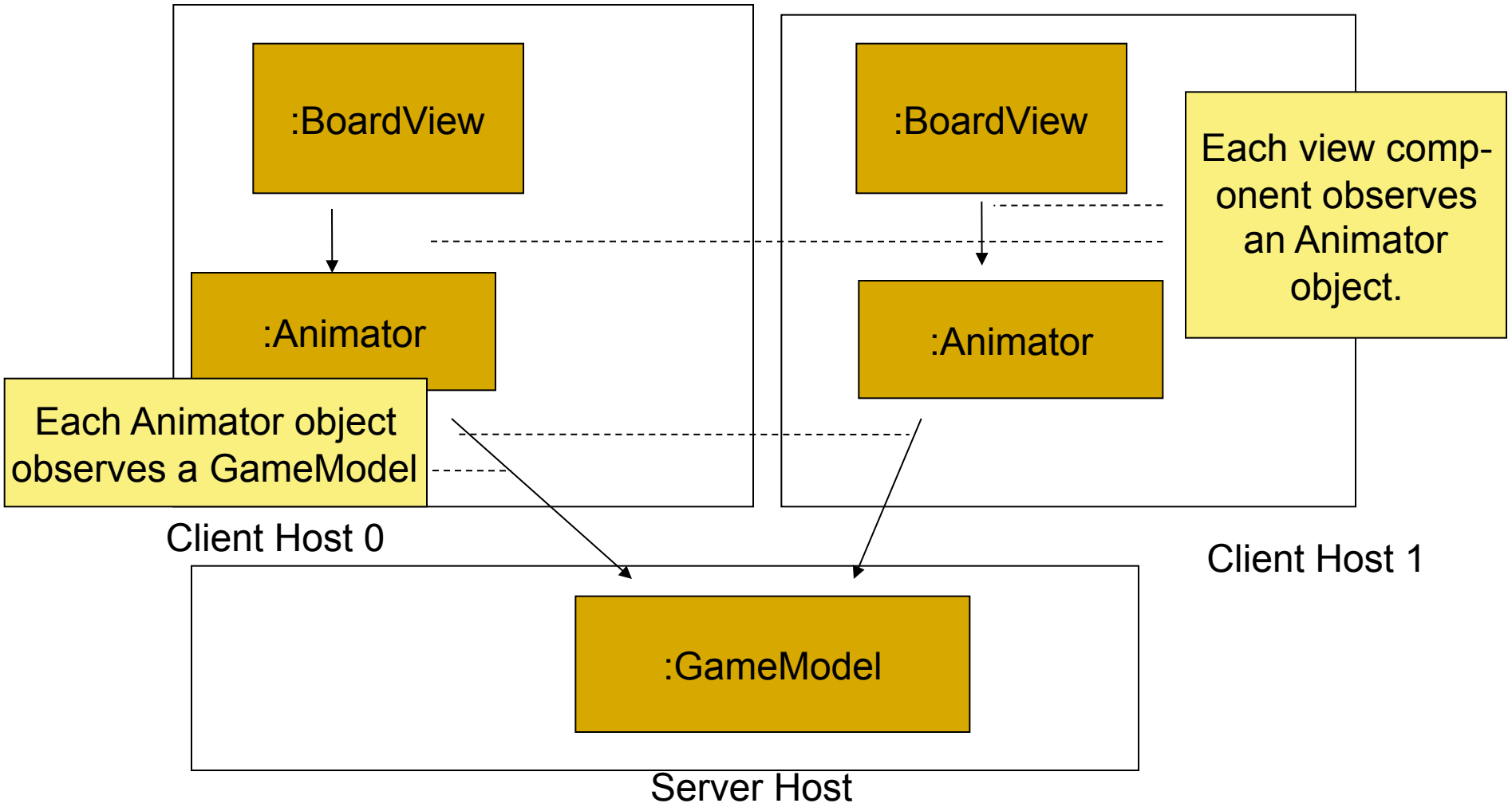
```
/** This implements the Observer interface*/
public void update(Observable o, Object arg) {
        repaint() ; }
```

- Recall that calling `repaint` causes a call to `paint` in the near future.
- While the call to `repaint` is on the animation thread, the call to `paint` will be on the GUI thread.
- Be careful: Most AWT and Swing defined methods should only ever be called from the GUI thread. `repaint` is an exception.

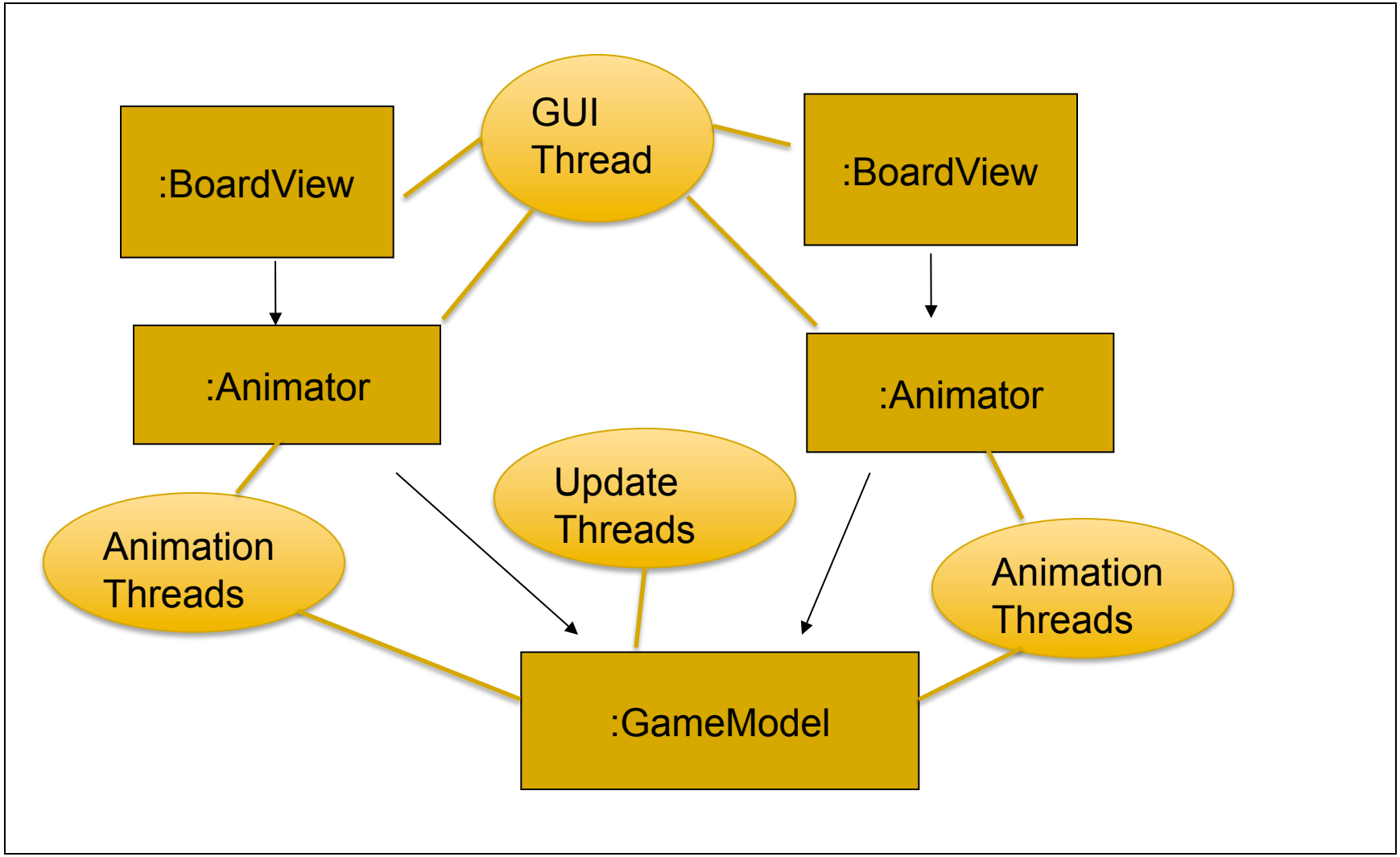
Second Problem

- How can multiple observers observe the same model concurrently?

Ultimately we want a distributed system



The Othello-1 program (nondistributed)



Sequential Observers

- The Game model is observed by 2 Animators
- Animating a change of state takes time.
- With the standard observer pattern, returning from Observer.update() indicates that an observer is finished observing.
- So, if the first observer takes a long time, the second is delayed.

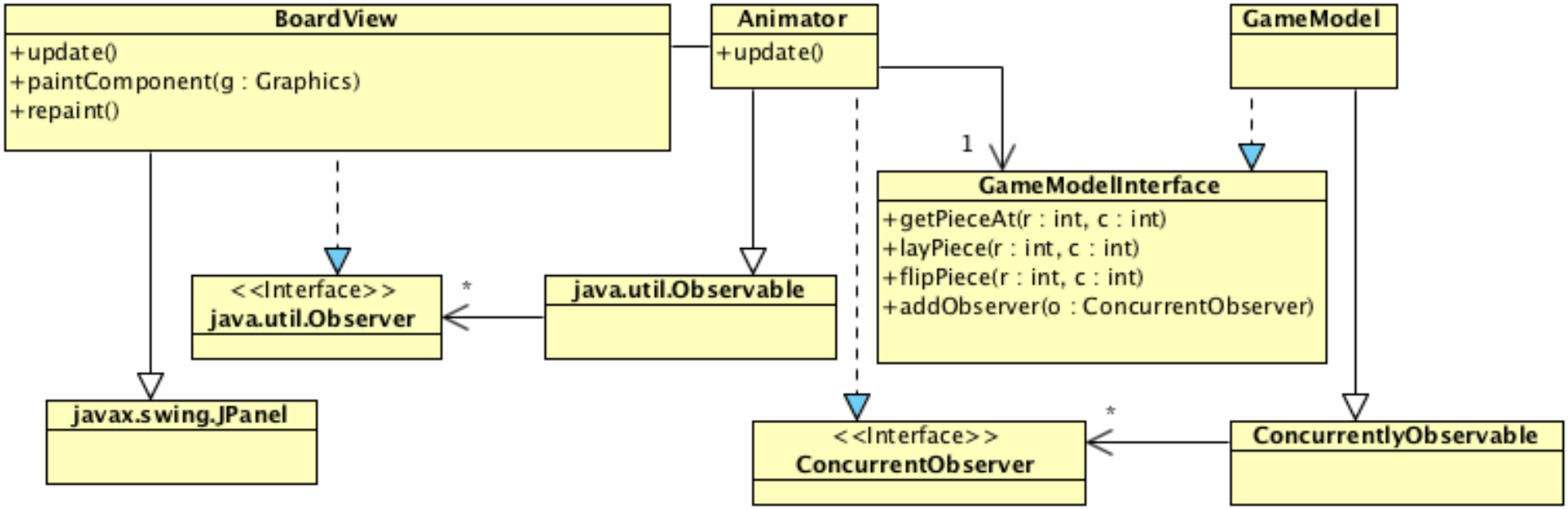
Concurrent Observers

- Instead, I did the following.
  - The GameModel is a ConcurrentlyObservable
  - Each observer (Animator) is a ConcurrentObserver
  - notifyObservers sets a counter to the number of observers
    - On update, each observer launches an animation thread and returns quickly from update.
    - Each animation thread decrements a counter when it is finished.
    - notifyObservers() waits until the counter is 0.

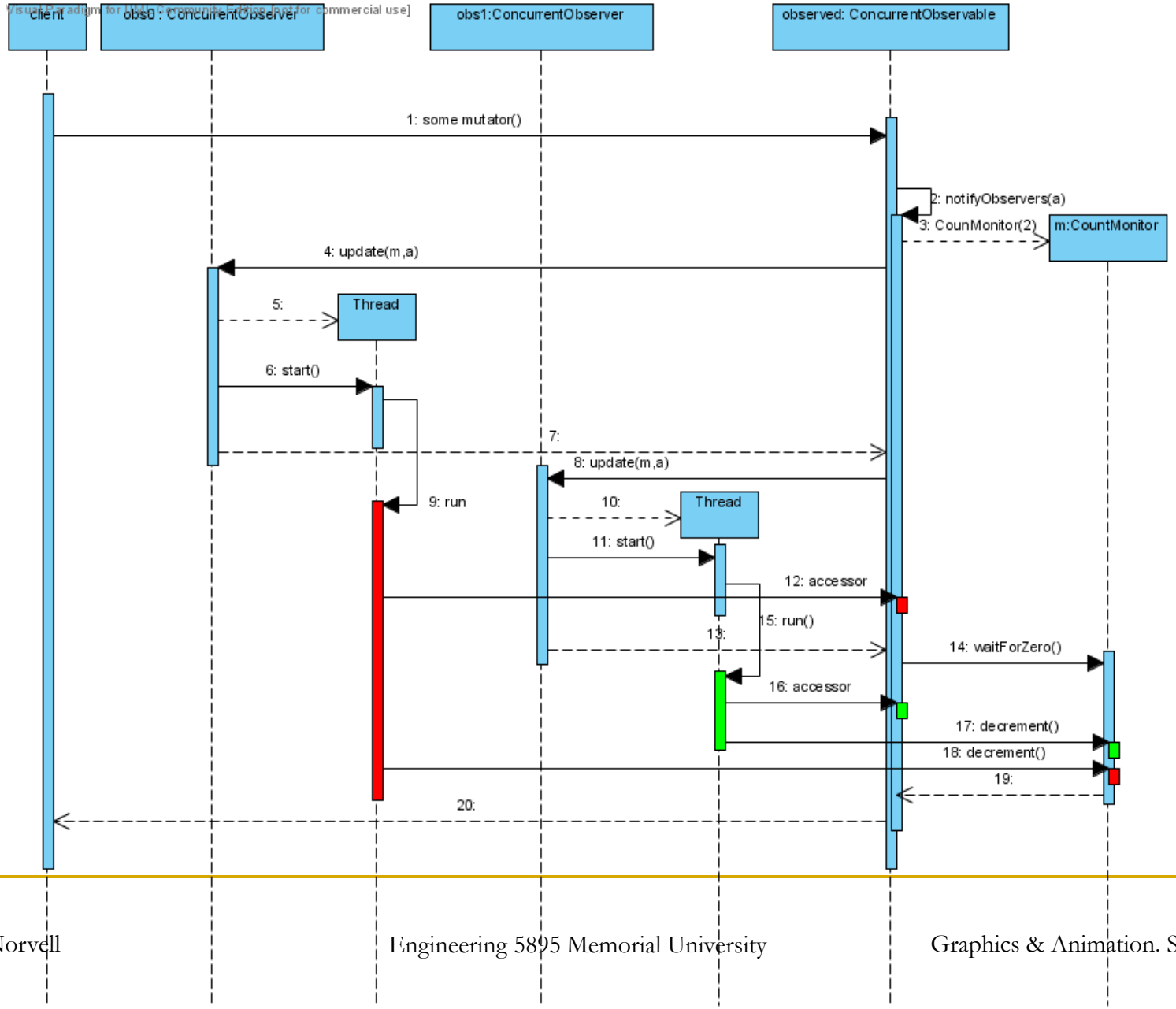
The classes

Each BoardView observes an Animator.

Each Animator observes a GameModel



Concurrent Observers



The CountMonitor

```
class CountMonitior implements CountMonitorInterface {
    private int count ;
    CountMonitor( int start ) {
        count = start ; }
    public synchronized void decrement() {
        count -= 1 ;
        if( count == 0 ) notifyAll() ; }
    synchronized void waitForZero() {
        if( count > 0 )
            try { wait() ; } catch( InterruptedException e ) { } }
}
```

Avoiding races

- The GameModel is shared by many threads.
  - In a future distributed varesion, there could be multiple update threads all trying to call the GameModel's mutators at the same time.
  - The animation threads that may need to read (observe) the GameModel while a mutator is still active.
  - In the distributed system, there will be more threads, as any callbacks from "update" will be on a new thread.

Reader/writer locks

- Threads accessing an object are classified as readers or writers depending on whether they call mutators or accessors
- Writers are either active or passive
  - Only active writers should change fields.
- At any time:
  - At most 1 writer can execute.
  - Any number of readers execute.
  - If the writer is active, no readers may execute.

Reader/writer locks

Mutators

```
public void mutator( … ) {
    // Wait until no writers
    // or readers are in.
    rw.writerEnters() ;
    try {
      …make changes… }
    finally {
        rw.writerExits() ; } }
```

Accessors

```
public type accessor( … ) {
    // Wait until no writers
    // are in.
    rw.readerEnters() ;
    try {
      … read the state …
       return result ; }
    finally {
      rw.readerExits() ; } }
```

Avoiding races

- I used a reader/writer lock to ensure that:
  - Only one thread may be executing mutators, at any given time.
  - Normally no thread may execute any accessor while another thread is executing a mutator.
  - But: During notifyObservers, I allow accessors to execute.

Avoiding races

- Mutators for observables

```
public void mutator() {
        try { rw.writerEnters()
                …Make changes…
                notifyObservers() ; }
        finally { rw.writerExits() ; } }
```

Avoiding races

```
proteced void notifyObservers() {
        rw.writerGoesPassive() ;
        create a counter initialized to the
            number of observers;
        for each observer
            call update, passing the counter;
       wait for the count to reach 0
       rw.writerGoesActive()  }
```

End of thread-based animation and concurrent observation