class Stock {
    ...
    public boolean synchronized testInv() {
        return (base_ >= 0 && ref_ >= 0 && price_ == (base_ + 0.1 * ref_));
    }
}

class BankAccount {
    ...
    public boolean synchronized testInv() {
        return (savings_ + maxOverDraft_ >= 0);
    }
}
Principles of the Test Units:
- You must only use the public interface of the Object Under Test in the test
- You must test only a single method in a single case (assume the other methods work)
- You must test return value AND post-conditions of the tested method
- You must test both positive (successful) and negative cases (those that don’t succeed)
- You must make sure a “trivial” implementation would fail some of the cases
- You must make sure “extreme” cases are tested.

1. Tests for AddStop:
- Enumeration of the cases:
  - With a state of UP, DOWN and NONE
  - UP and DOWN are symmetric (both must be tested – but only one is sufficient in answer)
  - NONE must always fail.
  - It is useful to decide whether successive addStop with the same stop should be counted as one stop or several stops. In the implementation given, the same value can be added several times – which is not the desired behavior. This can be caught when writing the test of AddStop or RemoveStop.

```java
void testAddUpPos() {
    Traj t1(UP):
```
if (! t1.addStop(1, 3)) { fail("cannot add (UP, 1, 3)"); return; }
    // Test @post
    if (t1.empty()) { fail("empty after add"); return; }
    if (t1.nextStop(1) != 3) { fail("bad next after add"); return; }
}

void testAddUpNeg() {
    Traj t1(UP);
    if (t1.addStop(3, 1)) { fail("add bad stop (UP, 3, 1)"); return; }
}

void testAddNone() {
    Traj t1(NONE);
    if (t1.addStop(1, 3)) { fail("add bad stop (NONE, 1, 3)"); return; }
}

2. Tests for RemoveStop
   - Enumeration of cases:
     - Remove on empty traj
     - Remove of element – test that it is removed
     - Remove of non-element
     - [Optional] A trivial implementation could “clear” for remove – test this is not done

void testRemoveEmpty() {
    Traj t1(UP);
    t1.removeStop(1); // Could suspect an exception try/catch possible
}

void testRemovePos() {
    Traj t1(UP);
    t1.addStop(1, 3);
    t1.removeStop(3);
    if (! t1.empty()) { fail("remove failed"); }
}

void testRemoveNeg() {
    Traj t1(UP);
    t1.addStop(1, 3);
    t1.removeStop(2);
    if (t1.empty()) { fail("remove had unexpected effect"); }
}

3. Tests for nextStop()
   - Enumeration of cases (symmetric for up and down) – none is not relevant
- There was an INTENTIONAL bug in the code to be discovered by your test units – when 
the nextStop is invoked after the max element or on empty trajectories, the code should return 
a known value.

void testNextEmpty() {
    Traj t1(UP);
    if (t1.nextStop(1) != -1) { fail("next on empty"); }
}

void testNextUp() {
    Traj t1(UP);
    t1.addStop(1,3);
    if (t1.nextStop(1,3) != 3) { fail("next UP 1,3 != 3"); }
    if (t1.nextStop(3,3) != 3) { fail("next UP 3,3 != 3"); }
    if (t1.nextStop(5,3) != -1) { fail("next UP 5,3 != -1"); }
}

This case is an example of a state machine. One must recognize that the update of the state 
machine depends on its state (in our case, the current trajectory and the current operation) and 
that external requests can trigger transitions from one state to another. The code must reflect 
this structure.

Important points to identify in the code:
- If your code is “long” (say more than 10 lines) – remember to split it into functions!
- The requests had to be deleted - each one, and the vector too
- A new trajectory has to be constructed when the current one is completed.
- The current floor must be updated by one floor maximum at each step
- The current state and the current trajectory must be updated at each step

To understand the method, it is useful to “simulate” manually the elevator using the test case 
that was given in the exam:

[Clock 0] [CF = 0 / IDLE / Traj empty / Pending empty]
[Clock 1] Incoming events: [1 UP] [3 DOWN] Traj[1,3] 
Do: IDLE → UP / floor++
[Clock 2] [CF = 1 / UP / Traj[1,3] / Pending empty] 
Incoming events: [GO 2] [2 DOWN] Traj[1,2,3] Pending[2 DOWN]
Do: UP → IDLE / remove 1 from Traj
[Clock 3] [CF = 1 / IDLE / Traj[2,3] / Pending[2 DOWN]]
Do: \( \text{IDLE} \rightarrow \text{UP} / \text{floor}^{++} \)

[Clock 4] \[CF = 2 / \text{UP} / \text{Traj[2,3]} / \text{Pending[2 DOWN]}] \]

Incoming events: none

Do: \( \text{UP} \rightarrow \text{IDLE} / \text{remove 2 from Traj} \)

[Clock 5] \[CF = 2 / \text{IDLE} / \text{Traj[3]} / \text{Pending[2 DOWN]} \]

Incoming events: none

Do: \( \text{IDLE} \rightarrow \text{UP} / \text{floor}^{++} \)

[Clock 6] \[CF = 3 / \text{UP} / \text{Traj[3]} / \text{Pending[2 DOWN]} \]

Incoming events: none

Do: \( \text{UP} \rightarrow \text{IDLE} / \text{remove 3 from Traj} / \text{add 2 to Traj} / \text{Pending pop} \)

[Clock 7] \[CF = 3 / \text{IDLE} / \text{Traj[2]} / \text{Pending empty} \]

Do: \( \text{IDLE} \rightarrow \text{DOWN} / \text{floor}^{--} \)

[Clock 8] \[CF = 2 / \text{DOWN} / \text{Traj[2]} / \text{Pending empty} \]

Do: \( \text{DOWN} \rightarrow \text{IDLE} / \text{Remove 2 from Traj} / \)

[Clock 9] \[CF = 2 / \text{IDLE} / \text{Traj empty / Pending empty} \]

Nothing to do.

// Code:

```c
void addRequests(Requests* rs) {
    foreach (r in rs) {
        bool added = canAddRequestToTraj(r);
        addRequest(r);
        if (added) { delete r; } // If not, r is now in pendingRequests_ and should be kept
    }
}
```

// What to do for each type of transition in the state machine
void moveToUp() { state = GOINGUP; floor++; }
void moveToDown() { state = GOINGDOWN; floor--; }

// Assume removeStop() removes duplicate – else loop here
void moveToIdle() { state = IDLE; currentTraj.removeStop(currentFloor); }

// This is the typical structure of a state machine: switch on current state
// determine how to move to the next state based on received events.
void nextState() {
switch (state) {
    case IDLE:  
        if (currentTraj.empty()) {/* nothing */}
        else if (currentTraj.getDirection() == UP) moveToUp();
        else if (currentTraj.getDirection() == DOWN) moveToDown();
        else {/* nothing */}
        break;
    case UP:  
        if (currentTraj.empty()) moveToIdle();
        else if (currentTraj.nextStop(currentFloor) == currentFloor) moveToIdle();
        else if (currentTraj.getDirection() == UP) moveToUp();
        else {/* error */}
        break;
    case DOWN: /* symmetric */
    }
}

Trajectory buildNewTraj() {
    if (pendingRequests_.empty()) return Trajectory(NONE);
    int floor = pendingRequests_.at(0).getFloor();
    if (floor < currentFloor_) return Trajectory(DOWN);
    else if (floor > currentFloor_) return Trajectory(UP);
    else return Trajectory(NONE);
}

void handleRequests(Requests* rr) {
    addRequests(rr);
    delete rr;
    nextState();
    if (currentTraj.isEmpty()) {
        currentTraj = buildNewTraj();
    }
    // Do not invoke addRequest on pendingRequests_ itself – because it updates it
    Requests tmpReqs;
    swap(pendingRequests_, tmpReqs);
    addRequests(tmpReqs);
}
Client1: TCP
Client2: UDP

Client1: TCP
Client2: UDP

(3) Socket socket = new Socket("tapuz",1300);
    I/O blocking

Client1: TCP
Client2: UDP

SocketChannel

SocketChannel socket = SocketChannel.open();
    socket.configureBlocking(false);
    InetAddress addr = new InetAddress("tapuz",1300);
Boolean bConnected = socket.connect(addr);
if (bConnected) {
    Selector selector = Selector.open();
    Socket.register(selector, SelectionKey.OP_CONNECT, socket);
    while (!bConnected) {
        selector.select();
        Iterator it = selector.selectedKeys().iterator();
        while (it.hasNext()) {
            SelectionKey sk = (SelectionKey)it.next();
            it.remove();
            if (sk.isValid() && sk.isConnectable()) {
                socket = (SocketChannel)sk.attachment();
                bConnected = socket.finishConnect();
                if (bConnected)
                    socket.cancel();
            }
        }
    }
}