גיליון תשובות

מספר נבחן:_____

שאלה 1 (30 נקודות)

סעיף א (6 נקודות)

```
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);
  }
}
```

סעיף ב (8 נקודות)

המחלקה Stock בטוחה: השדה _base הוא final, והשדות _ref ו _price משתנים תחת סינכרון לערכים חוקיים.

המחלקה BankAccount לעומתה אינה בטוחה: המתודה buyStock משנה את השדה _savings לאחר בדיקת מחיר המניה, אולם מחיר המניה עלול להשתנות בין בדיקת מחיר המניה לקנייתה בפועל, על ידי Dealer אחר הקונה את המניה וממילא מעלה את ערכה. לדוגמא: חשבון 1 עם 100 ₪ בחשבון וסכום מקסימאלי למשיכת יתר של 50 ₪, מעונין בקנית מניה בשווי 150 ₪, התנאי לקניה מתקיים, אך אז עובר זמן ה CPU ל Dealer אחר המבצע במלואה קניה של אותה מניה לחשבון בנק אחר תוך העלאת ערכה ל 151 ₪. כאשר יעבור זמן ה CPU לDealer הראשון, הוא יקנה את המניה ויגיע לאוברדראפט לא חוקי של 51 ₪.

סעיף ג (8 נקודות)

ההרצה אינה נכונה: בחישוב סידרתי, ההוראות לביצוע יכולות אמנם להתבצע בכל סדר אפשרי, אך ביצוע הוראה, ופרט הוראת קניה של מניה, מתבצע ברצף מתחילתו ועד סופו, כך שלתרחישים דוגמת התרחיש בסעיף הקודם, אין מקבילה בחישוב סידרתי.
תשובות שהתבססו על הסדר החופשי של ביצוע הפקודות בחישוב המקבילי לעומת הסדר הנוקשה של הביצוע בחישוב הסדרתי אינן נכונות:
- על פי הגדרה, נכונות נבדקת ביחס להרצה סדרתית בסדר כל שהוא.
- בתרגיל זה בפרט, על מנת להקל עליכם ולמנוע בלבול, הוגדר מראש מבנה הנתונים של ההוראות לביצוע, כך שאינו מבטיח סדר כלשהוא של הוצאת הוראות, כך שגם בחישוב הסדרתי אין סדר מסוים של ביצוע פקודות.

סעיף ד (8 נקודות)

יש לסנכרן את Stock בין בדיקת ערך המניה ועד לרכישתה בפועל, דבר המבטיח הן את הבטיחות והן את הנכונות:

```
public synchronized void buyStock(Stock stock) {
  synchronized(strock) {
      if ((savings_ + maxOverDraft_)  >= stock.getPrice()) {
        Integer num = stocks_.get(stock);
        if (num == null)
           stocks_.put(stock,new Integer(1));
        else
           stocks_.put(stock,new Integer(num.intValue() +1));
           decSaving(stock.getPrice());
           stock.incRef();
      }
  }
}
```

שאלה 2 (30 נקודות)

סעיף א (12 נקודות)

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.

```
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"); }
}
```

סעיף ב (18 נקודות)

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]}]
                Incoming events: [GO 2] [2 DOWN]        Traj[2,3] Pending{[2 DOWN]}
```

```
                Do:     IDLE → UP / floor++
[Clock 4]       [CF = 2 / UP / Traj[2,3] / Pending{[2 DOWN]}]
                Incoming events: none
                Do:     UP → IDLE / remove 2 from Traj
[Clock 5]       [CF = 2 / IDLE / Traj[3] / Pending{[2 DOWN]}]
                Incoming events: none
                Do:     IDLE → UP / floor++
[Clock 6]       [CF = 3 / UP / Traj[3] / Pending{[2 DOWN]}]
                Incoming events: none
                Do:     UP → IDLE / remove 3 from Traj / add 2 to Traj / Pending pop
[Clock 7]       [CF = 3 / IDLE  / Traj[2] / Pending empty]
                Do:     IDLE → DOWN / floor—
[Clock 8]       [CF = 2 / DOWN / Traj[2] / Pending empty]
                Do:     DOWN → IDLE / Remove 2 from Traj /
[Clock 9]       [CF = 2 / IDLE / Traj empty / Pending empty]
                Nothing to do.

// Code:

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* nr) {
        addRequests(nr);
        delete nr;
        nextState();
        if (currentTraj.isEmpty()) {
                currentTraj = buildNewTraj();
        }
        // Do not invoke addRequest on pendingRequests_ itself – because it updates it
        Requests tmpReqs;
        swap(pendingRequests_, tmpReqs);
        addRequests(tmpReqs);
}
```

שאלה 3 (30 נקודות)

סעיף א (3 נקודות)

Client1: TCP
Client2: UDP

סעיף ב (12 נקודות)

UDP: אם הודעה מתקבלת ברמת האפליקציה היא בעלת תוכן נכון, אולם ייתכן כי הודעות ילכו לאיבוד, או יגיעו שלא בסדר בו הן נשלחו.
TCP: הודעות שנשלחו תגענה עם תוכן נכון ובסדר בו הן נשלחו (בהנחה ששני התהליכים רצים והרשת לא נפלה)

רמת האמינות של UDP מושגת בעזרת ה checksum המצורף לכל הודעה נשלחת, והמאפשר אימותה ביעד.
רמת האמינות של TCP מושגת על ידי שימוש ב checksum, ומנגנון לשליחה חוזרת של הודעות ממוספרות, לדוגמא, האלגוריתם Go Back N (הערה: ההוראות מכוונות לזכר ולנקבה):

שולח:
- שלח עד N הודעות לא מאושרות
- בהתקבל אישור על הודעה I, כל ההודעות עד אליה (ועד בכלל) נחשבות כמאושרות
- בכל timeout שלח מחדש את כל ההודעות שאינן מאושרות

מקבל:
- צפה לקבלת הודעה i, תוך התעלמות מכל הודעה אחרת.
- עם התקבלות הודעה i, העבר אותה לרמת האפליקציה, וקדם את i ב 1.
- עבור כל הודעה המתקבלת שלח אישור על הודעה i-1.

סעיף ג (5 נקודות)

ב Client1 עלולה שורת הקוד:
Socket socket = new Socket("tapuz",1300);
להיקלע ל I/O Blocking, שכן היא כרוכה ביצירת קשר עם ה Socket בתהליך ב tapuz.
[ב UDP רק נפתח socket בתהליך של Client2 ללא "התקשרות" לתהליך היעד, כמו התקנת תיבת דואר בפתח הבית ביחס להתקשרות בטלפון למספר אחר]

סעיף ד (10 נקודות)

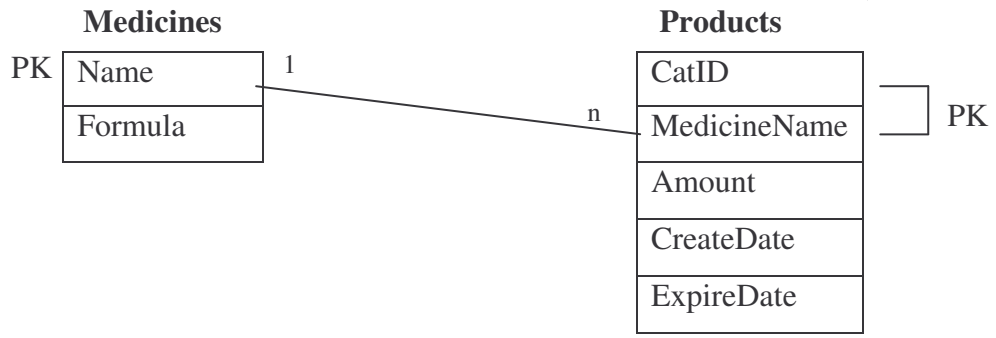
כדי לפתור את בעיית ה I/O blocking, נשתמש ב SocketChannel במקום ב Socket. כזכור Channel תומך ב Non blocking מחד, וארועים בו ניתנים לזיהוי על ידי Selector מאידך.
את ה SocketChannel נגדיר מבלי להתחבר מיד ל לתהליך היעד. נגדיר Selector ונבצע register של ה SocketChannel ל Selector על אירוע OP_CONNECT, המציין כי ניתן לסיים את התחברות של SocketChannel.
בכל פעם שיזהה ה Selector אירוע OP_CONNECT ב SocketChannel, נבצע finishConnect() לשם סיום ההתחברות. אם תהליך החיבור הסתיים נסיר את ה SocketChannel מה Selector (על ידי ביצוע cancel()), להלן הקוד:

```
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();
      }
    }
  }
}
```

שאלה 4 (10 נקודות)

סעיף א (5 נקודות)



סעיף ב (5 נקודות)

```
SELECT Products.MedicineName,Products.CatID,Products.Amount
FROM   Products
WHERE  Products.ExpireDate = '17/2/2006'
ORDER BY Products.MedicineName, Products.Amount DESC
```