אוניברסיטת בן-גוריון

מדור בחינות

מספר נבחן:_____

רשמו תשובותיכם בגיליון התשובות בלבד
תשובות מחוץ לגיליון לא יבדקו.

בהצלחה!

תאריך הבחינה: 6.2.2007
שם המורה: ד"ר מיכאל אלחדד
ניר צחר
מני אדלר
שם הקורס: תכנות מערכות
מספר הקורס: 202-1-2031
מיועד לתלמידי: מדעי המחשב, הנדסת תוכנה
שנה: תשס"ז
סמסטר: א'
מועד: א'
משך הבחינה: שלש שעות
חומר עזר: אסור

שאלה 1 (30 נקודות)

משחק השחמט, שמקורו כנראה בצפון מערב הודו במאות הראשונות לספירה, הגיע עם הכיבוש המוסלמי במאה השמינית לספרד, משם עבר לאיטליה ולשאר ארצות אירופה. בשאלה זו נעסוק במימוש של המשחק כתוכנית Java לא נדרש כל ידע מוקדם על המשחק וחוקיו.

- כדי לשחק שח, יש להצטייד בלוח ריבועי בן 64 משבצות בצבעי שחור ולבן, המכיל קבוצת כלים שחורים וקבוצת כלים לבנים (סוגי הכלים אינם רלבנטיים לשאלה זו).
- במשחק משתתפים שני שחקנים, כאשר כל שחקן מזיז בתורו את אחד מכליו למשבצת חדשה (על פי חוקים שאינם רלבנטיים לשאלה זו). במידה והמשבצת החדשה מכילה כלי של היריב, כלי זה יוסר מהלוח והכלי המכה יתפוש את מקומו.

הקוד להלן מגדיר שני טיפוסים של אובייקטים פסיביים: כלי (**ChessPiece**) ולוח משחק (**ChessBoard**).

המצב הפנימי של כלי (**ChessPiece**) מוגדר על ידי צבע (**color_**) וסוג (**type_**), כאשר הצבע מציין האם הוא שחור או לבן, והסוג מציין באיזה כלי מדובר (מלך, מלכה, צריח וכו' – לא רלבנטי לשאלה).

המצב הפנימי של לוח המשחק (**ChessBoard**) מורכב ממערך דו ממדי של משבצות, אשר על חלק מהן מונחים הכלים השונים (**pieces_**). הבונה של **ChessBoard** מאתחל, על ידי קריאה למתודה (**init**), את לוח זה כך שהוא מכיל קבוצת כלים שחורים וקבוצת כלים לבנים במיקום שאינו רלבנטי לשאלה זו.

```
class ChessPiece {
  private static final char[] types_ = {'K', 'Q', 'b', 'k', 'p', 'r' };
  private static final char[] colors_ = {'b', 'w'};
  public final char type_;
  public final  char color_;

  ChessPiece(char type,char color)
      throws WrongTypeException, WrongColorException {
    if (Arrays.binarySearch(types_, type) < 0)
      throw new WrongTypeException(type);
    if (Arrays.binarySearch(colors_, color) < 0)
      throw new WrongColorException(color);
```

```
    type_ = type;
    color_ = color;
  }
}

class ChessBoard {
  private ChessPiece[][] pieces_;
  ChessBoard() throws Exception{
    init();
  }

  private void init() throws Exception {
    pieces_ = new ChessPiece[8][8];
    initPieces('b',7,6);
    initPieces('w',0,1);
  }
  private void initPieces(char color, int line1, int line2) throws Exception {
    for (int i=0; i<8; i++)
      pieces_[line2][i]=new ChessPiece('p', color);  // pawn
    pieces_[line1][0]=new ChessPiece('r', color); //rook
    pieces_[line1][7]=new ChessPiece('r', color);
    pieces_[line1][1]=new ChessPiece('k', color); //knight
    pieces_[line1][6]=new ChessPiece('k', color);
    pieces_[line1][2]=new ChessPiece('b', color); //bishop
    pieces_[line1][5]=new ChessPiece('b', color);
    pieces_[line1][3]=new ChessPiece('Q', color); //queen
    pieces_[line1][4]=new ChessPiece('K', color); // king
  }

  public void movement(int i1, int j1, int i2, int j2)  throws IllegalMovementException {
    if (! legalMovement(i1, j1, i2, j2))
      throw new IllegalMovementException(i1, j1, i2, j2);
    else {
        pieces_[i2][j2] = pieces_[i1][j1];
        pieces_[i1][j1] = null;
    }
  }
  private boolean legalMovement(int i1, int i2, int j1, int j2) { … }
}
```

א. האם המחלקה **ChessPiece** בטוחה תחת כל חישוב מקבילי – נמקו (4 נקודות).

ב. הגדירו תנאי התחלה וסיום (Pre/Post Conditions) עבור המתודה **movement** במחלקה **ChessBoard** ונמקו האם קיום תנאי ההתחלה גורר, בכל ביצוע אפשרי, את קיומם של תנאי הסיום (אין צורך לפרט או להיכנס להגדרת המתודה **legalMovement**) (6 נקודות).

בחפירות ארכיאולוגיות נתגלתה גירסה קדומה של המשחק. בגירסה זו יכול כל שחקן לבצע מהלכי משחק בכל זמן שירצה, בניגוד לשיטת המשחק כיום, בה שחקן מבצע מהלך בתורו בלבד.

כדי לממש גירסה זו, מגדירה התוכנית הראשית במחלקה **Game** שני שחקנים ומפעילה אותם.

המחלקה **Player** מממשת שחקן במשחק זה, כאובייקט בעל מתודת **run()** אשר מצבו הפנימי מכיל לוח (**board_**), צבע (**color_**), ואובייקט הממש ממשק של אסטרטגיה (**strategy_**).

אובייקט האסטרטגיה (**strategy_**) של **Player** ממש את הממשק **Strategy** המכיל מתודה **getNextMovement** ומתודת **isFinish**. מתודת **getNextMovement** מקבלת לוח וצבע ומחזירה מהלך אחד עבור כלי אחד בצבע הנתון. מתודה **isFinish** מחזירה ערך true אם המשחק הגיע לסיומו על פי מצב הלוח.

פעילותו של השחקן מוגדרת במתודת ה **run()** של **Player**: כל פרק זמן מסוים מבוצעת קריאה למתודת **play** עד לסיום המשחק.

```
interface Strategy {
  public boolean isFinish(ChessBoard board);
  public int[] getNextMovement(ChessBoard board, char color);
}
class Strategy1 implements Strategy { … }
class Strategy2 implements Strategy { ... }

class Player implements Runnable {
  private Strategy strategy_;
  private ChessBoard board_;
  private char color_;

  public void play(ChessBoard board, char color)  {
     try {
        int[] movement = strategy_.getNextMovement(board, color);
        board.movement(movement[0], movement[1], movement[2], movement[3]);
     } catch(IllegalMovementException e) {
        // do nothing
     }
  }

  Player(Strategy strategy, ChessBoard board, char color) {
    strategy_ = strategy;
    board_ = board;
    color_ = color;
  }

  public void run() {
    try {
      while (!strategy_.isFinish(board_)) {
        Thread.currentThread().sleep((int)(Math.random()*1000));
        play(board_, color_);
```

```
      }
    } catch (Exception e) {
      return;
    }
  }
}

class Game  {
  public static void main(String[] args) {
    try {
      ChessBoard board = new ChessBoard();
      new Thread(new Player(new Strategy1(), board, 'w')).start();
      new Thread(new Player(new Strategy2(), board, 'b')).start();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

ג. האם הרצת התכנית נכונה, מבחינת התאמת תוצאת המשחק לתוצאה של הרצה סדרתית, בסדר מהלכים כלשהוא? נמקו (6 נקודות)

ד. לקראת שיווק המשחק בשוק הארופאי, הוחלט בחברה לשנות את אופן המשחק, כך שמהלך יתבצע בכל פעם על ידי שחקן אחד כל אחד בתורו, כפי שנהוג לשחק כיום. אחד המתכנתים בחברה טען כי מספיק לשם כך לסנכרן את המתודה **movement** במחלקה **ChessBoard** ע"י הוספת **synchronized** בהגדרתה. האם צודק המתכנת בטענתו? נמקו. (4 נקודות)

ה. עדכנו את התוכנית כך שיתקבל משחק שחמט בו כל שחקן מבצע מהלך אחד בתורו, ללא שימוש בסנכרון, תוך שמירה על נכונות ההרצה (לא ניתן לשנות את המחלקה **Game**). (10 נקודות).

שאלה 2 (30 נקודות)

להלן הגדרתה של המחלקה **Link** המממשת רשימה משורשרת של מחרוזות. בנוסף ניתנת הגדרה של המחלקה **List** המכילה שדה **head** המצביע לתחילתה של רשימה שכזו.

```
class Link {
private:
  Link* next_;
  std::string data_;
public:
  Link(const std::string& data, Link* link) : data_(data) { setNext(link); }
  ~Link() { }
  void setNext(Link* link) { next_ = link; }
  Link* getNext() const { return next_; }
```

```
  std::string getData() const { return data_; }
};

class List {
private:
  Link* head_;
public:
  List( ) : head_(0) { }
  void insertData(const std::string& data) {
    head_ = new Link(data, head_);
  }
  std::string removeData() {
    if (head_ == 0)
      return "";
    else {
      std::string data = head_->getData();
      Link* tmp = head_;
      head_ = head_->getNext();
      delete tmp;
      return data;
    }
  }
  ~List() {
    if (head_ != 0)
      delete head_;
  }
};
```

א. זהו בעיות בניהול זיכרון בהרצת קטע הקוד הבא, ותקנו את הגדרת המחלקות בהתאם (אין להוסיף שיטות/שדות) (6 נקודות)

```
{
  List ls;
  ls.insertData("Hello");
  ls.insertData("Sami");
  ls.insertData("Susu");
  std::cout << ls.removeData()  <<  std::endl;
}
```

ב. כעת נדרשת במחלקה **List** מתודה חדשה: **void splice(List& otherList)**

מתודה זו מקבלת רשימה **otherList** ומשרשרת אותה לסוף הרשימה הנוכחית, כך שבסיום הפעולה קיימות שתי רשימות:

- הרשימה הנוכחית, המכילה את אברי שתי הרשימות.
- הרשימה **otherList**, שהועברה כפרמטר למתודה, אשר תוכנה לא השתנה.

ממשו את המתודה **splice** <u>ללא הקצאת זיכרון חדש בheap</u>. אין להוסיף או לשנות שיטות/שדות קיימים (6 נקודות)

ג. מה הבעיה בקטע הקוד הבא העושה שימוש במתודה **splice**? (6 נקודות)

```
{
  List ls1;
  List ls2;
  ls1.insertData("Hello");
  ls2.insertData("Sami");
  ls2.insertData("Susu");
  ls2.splice(ls1);
  std::cout << ls1.removeData()  <<  std::endl;
}
```

ד. תקנו את הגדרת המחלקות כך שהבעיה בסעיף הקודם לא תתרחש. על **splice** עדיין להתבצע ללא הקצאת זיכרון. (12 נקודות)

שאלה 3 (30 נקודות)

שרת מחירים הינו תהליך העומד בקשר עם קבוצת תהליכים של חנויות ברשת. שרת המחירים יכול לברר עם כל אחת מחנויות אלו מה מחירו של מוצר מסוים ולתת ללקוח המעוניין בכך סקירה של המחירים של המוצר בחנויות השונות ברשת.
במימוש הבא, הקשר בין השרת לחנויות נעשה בעזרת RMI, באופן הבא:

חנות מממשת את הממשק **Seller** המכיל את המתודה **getPrice** המחזירה את מחירו של המוצר, אשר שמו נשלח כפרמטר, ואת המתודה **buy** המבצעת רכישה של המוצר:

```
interface Seller extends java.rmi.Remote {
   public int getPrice(String product) throws RemoteException;
  public  boolean buy(String product) throws RemoteException;
}
```

שרת המחירים מממש את הממשק **Dealer** המכיל את המתודה **register**. **register** מקבלת חנות (או ליתר דיוק אובייקט הממש את הממשק **Seller**), ומבצעת רישום של חנות זו על ידי הוספתה לקבוצת החנויות שנרשמו עד כה. הממשק **Dealer** כולל גם את המתודה **getCheapestSeller** המקבלת שם של מוצר ומחזירה את החנות המוכרת מוצר זה, במחיר הזול ביותר:

```
interface Dealer extends java.rmi.Remote {
   public void register(Seller seller) throws RemoteException;
   public Seller getCheapestSeller (String product) throws RemoteException;
}
```

כאשר מריצים את השרת, הוא מוסיף את עצמו כ **Dealer** ל **rmiregistry** על ידי ביצוע **rebind**. באופן זה יכולה חנות לפנות בפעולת **lookup** ל **rmiregistry**, לקבל את שרת המחירים כ **Dealer**, ולאחר מכן להירשם בשרת זה על ידי הפעלת המתודה **register** שלו.

בקוד להלן, אנו מניחים שתהליך **rmiregistry** רץ על **lead** בפורט 1984.

```
class PriceServer implements Dealer extends java.rmi.server.UnicastRemoteObject {
  private Vector<Seller> sellers_;
  PriceServer (…)  throws RemoteException  { …}
  public void register(Seller seller) throws RemoteException { sellers_.add(seller);  }
 public Seller getCheapestSeller (String product) RemoteException {
    //@1: TODO
  }
  public static void main(String[] args) throws Exception {
    PriceServer server = new PriceServer(…);
    Naming.rebind ("//lead:1984/dealer", server);
  }
}
```

```
class Store implements Seller extends java.rmi.server.UnicastRemoteObject {
  Store(…)  throws RemoteException  { …}
  Public int getPrice(String product) throws RemoteException { … }
  public boolean buy(String product) throws RemoteException { … }
  public static void main(String[] args) throws Exception {
    Store store = new Store(…);
    Dealer dealer = (Dealer)Naming.lookup("//lead:1984/dealer");
    dealer.register(store);
  }
}
```

א. ממשו את המתודה **getCheapestSeller** במחלקה **PriceServer** המקבלת שם של מוצר, ומחזירה את ה **Seller** המוכר מוצר זה במחיר הזול ביותר (אם לא קיימת חנות כזו המתודה מחזירה **null**). (4 נקודות)

ב. להלן תוכנית הממשת לקוח המתחבר לשרת לשם קבלת החנות המוכרת מוצר זה במחיר הזול ביותר, ולאחר מכן מבצע את הקניה של המוצר מחנות זו. ממשו את המתודה **getMinPriceSeller** (4 נקודות)

```
class Client {
  public void buyProduct(String product) {
    Seller seller = getMinPriceSeller(product);
    if (seller != null)
      seller.buy(product);
  }
  private Seller getMinPriceSeller(String product) {
    //@2: TODO
  }
```

```
}
```

ג. כמה פעולות תקשורת נדרשות לשם ביצוע הבקשה של הלקוח בסעיף ב' (כאשר מעבר הלוך-חזור בין שני תהליכים מוגדר כפעולת תקשורת אחת) (4 נקודות)

הוחלט לשנות את מימוש התקשורת בין שרת המחירים והחנות ללא שימוש ב RMI אלא בקשר TCP מעל sockets.

ד. הגדירו פרוטוקול של הודעות במקום הממשקים **Seller** ו **Dealer**. יש לציין את ההודעות שיכולות לעבור מהשרת לחנות, ואת ההודעות שיכולות לעבור מהחנות לשרת. הגדרת הודעה תכלול את המבנה המדויק שלה, את הפעולה שאמורה להתבצע עם קבלתה, ואת ההודעה הנשלחת בחזרה, אם נדרש, בסיום ביצוע הפעולה. (8 נקודות)

ה. השלימו את הקוד החלקי הבא של **Client**, הממש את הפרוטוקול בעזרת קשר TCP לשרת **PriceServer** הרץ על **black**, עם **ServerSocket** על פורט 2035. ע"י שימוש בפונקצית **callServer** ובהתאם לפרוטוקול שהגדרתם בסעיף הקודם. (10 נקודות)

```
class Client {
   public String callServer(Socket server, String msg) {
      PrintWriter out = new PrintWriter(server.getOutputStream(), true);
      out.println(msg);
      BufferedReader in = new BufferedReader(
         new InputStreamReader(server.getInputStream()));
      return in.readline();
   }
   public void buyProduct(String product) {
      // @1: TODO
   }
   private Seller getMinPriceSeller(String product) {
      // @2: TODO
   }
}
```

שאלה 4 (10 נקודות)

לאחר דיון סוער, הוחלט בהתאחדות לכדורגל לבנות בסיס נתונים שישמש לאחסון נתוני המשחקים בליגת העל. משחק מוגדר על ידי שתי קבוצות, מגרש, ומספר המחזור. נתוני קבוצה כוללים את שם המועדון, שם היישוב בו הוא פועל, ואת שם איש הקשר של המועדון. נתוני מגרש כוללים את שמו, כתובתו, והקיבולת שלו (מספר צופים מקסימאלי).

א. הגדירו מודל נתונים עבור בעיה זו. הגדרת המודל תתבסס על טבלאות, שדות, ומפתחות ראשיים וזרים (5 נקודות)

ב. כתבו שאילתת SQL המציגה את רשימת המחזורים בהם משחקת קבוצה מירושלים באצטדיון עם קיבולת של למעלה מ 10,000 מקומות. (5 נקודות)