גיליון תשובות

מספר נבחן: _______________

שאלה 1 (30 נקודות)

סעיף א (4 נקודות)

המחלקה ChessPiece בטוחה תחת כל חישוב מקבילי.
שדותיה הפומביים הם final ותקינות ערכיהם נבדקת בבונה.
שדותיה הסטטים הם private וערכם לא משתנה בתוך המחלקה.

סעיף ב (6 נקודות)

@Pre: legalMovement(i1,j1,i2,j2) == true
@Post: pieces_[i2][j2] == @Pre(pieces_[i1][j1]) && pieces_[i1][j1] ==null

עבור חישוב סידרתי קיום תנאי ההתחלה אכן יגרור את תנאי הסיום, אולם עבור חישוב מקבילי אין הדבר כן.
לדוגמא: כלי אחד מוצב ב (1,1) וכלי שני ב (2,2).
שחקן אחד מבצע movement(1,1,2,2) ושחקן שני מבצע במקביל movement(2,2,1,1).
במתודה movement מבצע השחקן הראשון את ההשמה של הכלי ב (1,1) ל (2,2), ואז עובר זמן ה cpu לשחקן השני המבצע אף הוא השמה של הכלי מ (2,2) ל (1,1) והשמת null ל (1,1) המכיל את הכלי שהועבר כבר על ידי השחקן הראשון. עם קבלת זמן cpu ישים השחקן הראשון ערך null ב (1,1).
בסיום הפעולה הן (1,1) והן (2,2) מכילים ערך null, כך שתנאי הסיום לא התקיימו לאף אחת משתי פעולות ה movement.

סעיף ג (6 נקודות)

התוכנית אינה נכונה.
בתרחיש שתואר בסעיף הקודם, התוצאות עבור הרצה סדרתית בכל סדר שהוא (קודם שחקן ראשון ואחר כך השני, או להפך, קודם השחקן השני ואח"כ הראשון) אחד מהתאים (1,1) או (2,2) יכילו כלי כלשהוא.

סעיף ד (4 נקודות)

המתכנת אינו צודק. הסנכרון יבטיח אמנם את קיומם של תנאי הסיום, אולם יתכנו מספר מהלכים רצופים של אותו שחקן, בהתאם ל scheduler.

סעיף ה (10 נקודות)

זוהי דוגמא קלאסית ל confinement. אנו משווקים אובייקט שאינו עומד מבטיח נכונות תחת חישוב מקבילי, עם 'הוראות הפעלה'.
הוראות ההפעלה במקרה שלנו, ידאגו לכך שרק ת'רד אחד בכל פעם יוכל לבצע מהלך.
ניתן לדאוג לכך גם ללא סנכרון (ובפרט אם זוהי דרישת הלקוח שאינו מוכן בשום פנים לנעול כל אובייקט שהוא).

נגדיר במחלקה ChessBoard שדהה מטיפוס פשוט שהקריאה ממנו אטומית (כל אחד מה simple types למעט long ו double), ונסמנו כ volatile כדי להבטיח שכל שינוי בערכו יתעדכן אצל כל הת'רדים בתהליך. שדה זה ישמש לציון השחקן המשחק כרגע, והוא יאותחל לציון אחד השחקנים. בניגוד לקוד הקיים בו שחקן מתעורר ומיד משחק, נוסיף תנאי הבודק האם כרגע תורו.

```
class ChessBoard {
  private ChessPiece[][] pieces_;
  private volatile char player_;

  ChessBoard() throws Exception{
    init();
    player_ = ‘w’;
  }

  …

  public char getPlayer() { return player_; }

  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;
    }
    player_ = (‘w’ ? ‘b’ : ‘w’);
  }
}

class Player implements Runnable {
  …
  public void run() {
    try {
      while (!strategy_.isFinish(board_)) {
        Thread.currentThread().sleep((int)(Math.random()*1000));
        if (board_.getPlayer() == color_)
          play(board_, color_);
      }
    } catch (Exception e) {
      return;
    }
  }
}
```

שאלה 2 (30 נקודות)

a) The problem is with the fact that the destructor of List deletes the head of the list, but not any other node (since the destructor of Link is empty).

The solution is to remove all element from the list when it is deleted:

```
~List() { while (head_) removeData();}
```

The solution which changed the destructor of Link like this:

```
~Link() { delete next_; }
```

is almost correct. If this is the change made, then one needs to change the "removeData" method. Otherwise, once the head is removed, all the list will be lost. Moreover, it makes life difficult when implementing (d).

(*) Those who argued that we cannot pass a regular char * string to insertData got only partial credit. This is not correct, since the type of the argument in insertData is const& and a char * can be converted to a const std::srtring & .

(*) those who argued the we should change "str_" to "data_" in Link::getData got no credit at all, since this is not a memory management problem.

b)

```
void List::splice (List &otherList)
{
        Link *tmp = head_;
        if (! head_){
          head_ = otherList.head_;
          return;
        }

        while (tmp->getNext())
          tmp=tmp->getNext();

        tmp->setNext(otherList.head_);
        return;
  }
```

(*) solutions which popped all links from one list and added them to the other list got partial credit, as the method List::insertData allocates new memory by creating new links.

c) There are two problems. When ls1.removeData is called, a link in ls2 now pints to deleted memory. In itself, this can sometimes work, since deleted memory can still be accessed. However, when we exit the current activation frame and the destructor of ls1 and ls2 will be called, we will try to free an already freed memory.
Also, what will happen if ls2 is freed before ls1, regardless of whether removeData was called at all??

d) The simplest solution is to use reference counting --> for each link we count how many reference to it exits. We delete it only after all references are removed.

(*) several solutions offered to use a back pointer, to remember the previous link. However, this will not work in the following cases:
1. what if ls2 is freed before ls2?
2. what happens if a list is spliced several times?
3. many other problems...

The correct solution is presented:

```
#include <string>
#include <iostream>


class Link{
private:
        Link *next_;
        std::string data_;
        int references;
public:
        Link(const std::string &data, Link *link):
                data_(data){
                setNext(link);
                references = 0;
        }
        void setNext(Link *next){
                next_ = next;
        }
```

```
    Link *getNext() const { return next_; }
    std::string getData() const { return data_; }
    void incRef(){ references++; }
    void decRef(){ references--; }
    int getRef(){ return references; }
    void incRefRecursively(){
        incRef();
        if (next_)
            next_->incRef();
    }
    ~Link(){ }
};

class List{
private:
    Link *head_;

    // this can be made into the "delete" operator of Link, with slight
    // modifications. might be more elegant...
    void deleteLink(Link *link){
        link->decRef();
        //std::cout << "ref count for "<< link->getData() << " is: "<< link->getRef() << std::endl;
        if (link->getRef() <= 0)
            delete link;
    }
public:
    List(): head_(0){}
    void insertData(const std::string &data){
        head_ = new Link(data, head_);
        head_->incRef();
    }
    std::string removeData(){
        if (head_ == 0)
            return "";
        std::string ans = head_->getData();
        Link *tmp = head_;
        head_ = head_->getNext();
        deleteLink(tmp);
        return ans;
    }

    void splice(List &otherList){
        Link *tmp = head_;
        if (! head_){
            head_ = otherList.head_;
            head_->incRefRecursively();
            return;
        }
        while (tmp->getNext())
            tmp=tmp->getNext();

        tmp->setNext(otherList.head_);
        //we now need to incerent the reference count of every
        //member in otherList
        if (otherList.head_)
            otherList.head_->incRefRecursively();
        return;
    }

    ~List() { while (head_) removeData(); }
};


int main(){

    List ls1;
    List ls2;
    List ls3;

    ls1.insertData("hello");
    ls2.insertData("sami");
    ls2.insertData("susu");
    ls3.insertData("list3 tail");

    ls2.splice(ls1);
    ls3.splice(ls1);

    std::cout << ls1.removeData() << std::endl;
```

```
        std::cout << ls2.removeData() << std::endl;
        std::cout << ls3.removeData() << std::endl;
        std::cout << ls3.removeData() << std::endl;



        return 0;
}
```

שאלה 3 (30 נקודות)

סעיף א (4 נקודות)

Implement getCheapestSeller in class PriceServer - given a product name, return the Seller which sells this product at the cheapest price. If such a Seller is not found, return null.

```
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 {
    Seller bestSeller = null;
    int bestPrice = java.lang.Integer.MAX_VALUE;
    for (seller : sellers_) {
      // Assume getPrice return -1 if the seller does not sell product.
      int price = seller.getPrice(product);
      if (price >= 0 && price < bestPrice) {
        bestPrice = price;
        bestSeller = seller;
      }
    }
    return bestSeller;
  }
  public static void main(String[] args) throws Exception {
    PriceServer server = new PriceServer(…);
    sellers_ = new Vector<Seller>();
    Naming.rebind ("//lead:1984/dealer", server);
  }
}
```

Notes:
1.1 There is no lookup in this code. The sellers are known to the PriceServer because they have registered to it through the register method.
1.2 The sellers_ vector contains remote object references. Each invocation to seller.getPrice() is a round-trip over the network. It is an expensive operation. Therefore, efforts must be made to avoid calling this method whenever possible.

סעיף ב (4 נקודות)

Implement the method getMinPriceSeller of the Client class:

```
class Client {
  public void buyProduct(String product) {
    Seller seller = getMinPriceSeller(product);
    if (seller != null)
      seller.buy(product);
  }
  private Seller getMinPriceSeller(String product) {
    Dealer dealer = (Dealer)Naming.lookup("//lead:1984/dealer");
    return dealer.getCheapestSeller(product);
  }
}
```

Notes:
2.1 There must be a lookup call to locate the dealer server.

סעיף ג (4 נקודות)

How many round-trip exchanges are performed when the buyProduct() method is invoked?

```
Client.buyProduct(p)
--> Client.getMinPriceSeller(p)
   --> Naming.lookup(d): This is one round-trip operation to the RMI name server.
   --> dealer.getCheapestSeller(p): This is one round-trip operation to dealer.
      --> For each seller in dealer.sellers_:
         seller.getPrice(p): This gives N round-trip operations to the sellers.
--> seller.buy(p): This is one round-trip operation to the cheapest seller.
```

Altogether - N+3 round-trip operations - where N is the number of stores registered in the PriceServer.
If there are no stores selling the product p, then only N+2 round-trips.

Notes:
3.1 If your code in (2) is not optimized, there may be many more round-trips.
3.2 Full Credit was given for any answer mentioning N.

סעיף ד (8 נקודות)

Define a message protocol instead of the interfaces Dealer and Seller.

We must define 2 protocols - one for each interface (Dealer and Seller).
A protocol determines how 2 sides communicating with each other interact with each other. A protocol to be used over a TCP type of transport layer, must include:

- Framing: how messages are split into distinct units.
- Serialization: how messages are formed, including which verbs and arguments can be composed into each message type,
  which encodings are used for each data type.
- For each message type, what possible answers are expected.
- Which side of the protocol starts an exchange.
- Naming and Addressing: how partners in the exchange are located (what addresses are used).

We decide for both protocols:
- Framing: all messages are sent as "lines" - that is as strings terminated by a newline character.
- Serialization: use UTF16 unicode strings for all exchange.
  Data types encoding:
  - Products are encoded as a string (denoting the name of the product). Product names contain no space and no newline.
  - Prices are encoded as a string denoting the price of the product as an integer encoded in decimal.
  - A price value -1 has the meaning that a product is not sold.
  - Sellers are encoded as the address of the store in the form <host>:<port> - for example silver:10120

Protocol 1: Dealer

Message "register":
  Syntax: "register <seller>"
  Example: register silver:10120

  Sent by: seller to price server.
  Possible replies:
  - OK [Registration performed correctly]
  - NOK <error> [Registration failed - reason provided - for example: ill-formed seller, seller not accessible]

Message "getCheapestSeller":
  Syntax: "getCheapestSeller <product>"
  Example: getCheapestSeller bread

  Sent by: client to price server.
  Possible replies:

- <seller>   [Name of seller selling product for cheapest price - for example: silver:10120]
- null   [No seller found selling product]
- NOK <error> [Query failed - reason provided - for example: bad product name]

Protocol 2: Seller

Message "getPrice":
  Syntax: "getPrice <product>"
  Example: getPrice bread

  Sent by: price server to seller.
  Possible replies:
  - <integer>   [Price of product - -1 if seller does not sell product]
  - NOK <error> [Query failed - reason provided - for example: bad product name]

Message "buy":
  Syntax: "buy <product>"
  Example: buy bread

  Sent by: client to seller.
  Possible relies:
  - OK   [Transaction completed]
  - NOK <error> [Transaction not completed - reason provided - for example: product not found, no product available]

Connection Client/PriceServer
- Client starts and sends getCheapestSeller message.

Connection Client/Seller
- Client starts and sends buy message.

Connection Seller/PriceServer
- Seller starts and sends register message.

Connection PriceServer/Seller
- PriceServer starts and sends getPrice message.

Notes:
4.1 There was no requirement to write any code in this question.
4.2 There was no mention in the question that moving from RMI to TCP meant changing the way the
parties interact - for example, that the priceServer becomes the one that performs the "buy" operation on behalf of the client.

סעיף ה (10 נקודות)

Complete the code of class Client that implements the protocol over TCP.

```
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) {
   Seller seller = getMinPriceSeller(product);
   if (seller != null)
     seller.buy(product);
  }
  private Seller getMinPriceSeller(String product) {
   try {
    Socket dealer = new Socket(new InetAddress("black", 2035));
    String reply = callServer(dealer, "getCheapestSeller " + product);
    if (reply.equals("null")) {
     return null;
    } else if (reply.startsWith("NOK")) {
     return null;
    } else {
     return new SellerTCP(reply);
    }
   } catch (Exception e) {
    return null;
   }
  }

  // This code was not expected in the reply
  // This is a "proxy" object built-by hand over TCP.
  class SellerTCP implements Seller {
   public SellerTCP(String address) throws RemoteException {
    String[] tokens = address.split(":");
    host_ = tokens[0];
    port_ = Integer.parseInt(tokens[1]);
    try {
     s_ = new Socket(host_, port_);
    } catch (Exception e) {
     throw new RemoteException("Seller " + address + " not found");
    }
   }
   public int getPrice(String product) throws RemoteException {
    if (s_ == null) throw new RemoteException("Not connected");
    String reply = callServer(s_, "getPrice " + product);
    if (reply.startsWith("NOK")) {
     thow new RemoteException(reply);
    } else {
     return Integer.parseInt(reply);
    }
   }
   public  boolean buy(String product) throws RemoteException {
    if (s_ == null) throw new RemoteException("Not connected");
    String reply = callServer(s_, "buy " + product);
    if (reply.startsWith("NOK")) {
     throw new RemoteException(reply);
    } else {
     return true;
    }
   }
   private String host_;
   private int port_;
   private Socket s_;
  }
}
```

Notes:

5.1 The whole idea of RMI is that it works with INTERFACES.
The same interface that is implemented through RMI could also be implemented over TCP as shown above.

5.2 The following alternative was naturally accepted:

```
  public void buyProduct(String product) {
   TCPSeller2 seller = (TCPSeller2)getMinPriceSeller(product);
   if (seller != null) {
    try {
     Socket s = new Socket(seller.getAddress());
     String reply = callServer(s, "buy " + product);
    }
```

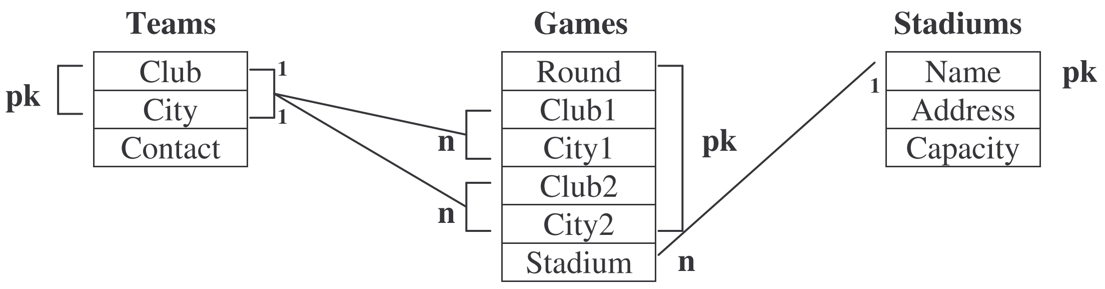
```
      }
    }
    private Seller getMinPriceSeller(String product) {
      try {
        Socket dealer = new Socket(new InetAddress("black", 2035));
        String reply = callServer(dealer, "getCheapestSeller " + product);
        if (reply.equals("null")) {
          return null;
        } else if (reply.startsWith("NOK")) {
          return null;
        } else {
          return new SellerTCP2(reply);
        }
      } catch (Exception e) {
        return null;
      }
    }

public class TCPSeller2 implements Seller {
    public SellerTCP(String address) {
        String[] tokens = address.split(":");
        addr_ = new InetAddr(tokens[0], Integer.parseInt(tokens[1]));
    }
    public boolean buy(String product) { return true; }
    public int getPrice(String product) { return -1; }
    public String getAddress() { return addr_; }
    private InetAddr addr_;
}
```

5.3 There was no reason to send the "buy" command to the priceServer.

5.4 There was no reason to send the "buy" command without first checking which store is the cheapest.

שאלה 4 (10 נקודות)



A complete solution:

```
SELECT DISTINCT Games.Round FROM
        Stadiums JOIN ( (Games JOIN Teams ON
                          (Games.Club1 = Teams.Club AND Games.City1 = Teams.City)
                          OR
                         (Games.Club2 = Teams.Club AND Games.City2 = Teams.City))
                  ON Stadiums.Name = Games.Stadium
WHERE Teams.City = 'Jersualem' AND Stadiums.Capacity > 10000
```

An answer with the other syntax was also accepted:

```
SELECT DISTINCT Games.Round
FROM Games, Teams, Stadiums
WHERE
  (((Games.Club1 = Teams.Club AND  Games.City1 = Teams.City) OR
    (Games.Club2 = Teams.Club AND  Games.City2 = Teams.City) AND
   Stadiums.StadiumName == Games.Stadium)
 AND
  Teams.City = 'Jersualem' AND Stadiums.Capacity > 10000
```

Notes:
- ‘DISTINCT' was not required for full credit.
- Answers with different/bad syntax were also accepted in cases the intent was clear.
- Other SQL queries that result in the same output were also accepted.