Avinor Rotem - Ben-Gurion University
Section: Examinations

Number of students: 
Fill your answers in the booklet. Answers outside the booklet will not be checked.

Note:
0% on empty answers will be given with success!

Date of examination: 13.2.2017
Name of examiner: 
Name of the course: 

Number of the course: 202-1-2031
Taught in the Department of Computer Science, intended for students of Software Engineering: 197
Year: A
Semester: A
Duration of examination: Prohibited

Question 1 (30 points)

In this question, we consider a board game played by two players. The board is an N x N square. Each player has a strategy that determines the direction in which they will move at each turn. When one of the players reaches the cup, the game ends.

The following is a partial implementation of the system:

```java
public enum Direction { UP, DOWN, RIGHT, LEFT }
public enum Result { SUCCESS, FAIL, WIN, LOST }
public interface Strategy { Direction nextMove(); }
public final class Location {
    public final int i, j;
    public Location(int i, int j) { this.i = i; this.j = j; }
    @Override
    public boolean equals(Object obj) { … }
    public Location move(Direction direction) {
        switch (direction) {
            case RIGHT:
                return new Location(i, j + 1);
            case LEFT:
                return new Location(i, j - 1);
            case DOWN:
                return new Location(i + 1, j);
            case UP:
                return new Location(i - 1, j);
        }
    }
}
```

For the implementation of the board, enter a code block with 20% deducted!

Good luck!
return new Location(i + 1, j);
case UP:
    return new Location(i - 1, j);
default:
    throw new IllegalArgumentException();
}

public final class Player implements Runnable {
    private final Strategy strategy;
    private final Board board;
    private Location location;

    public Player(Board board, Strategy strategy, Location location) {
        this.board = board;
        this.strategy = strategy;
        this.location = location;
    }

    public void run() {
        while (true) {
            Location to = location.move(strategy.nextMove());
            Result result = board.move(location, to);
            if (result == Result.SUCCESS) location = to;
            else if (result == Result.WIN || result == result.LOST) break;
        }
    }
}

public interface Board {
    /** Start the game. */
    void start();

    /** Gracefully terminate the program. */
    void stop();

    /** Move a Player from Location a to Location b. */
    * Result.FAIL is returned if b is out of bounds or if there is a Player at b.
    * If the game has ended, Result.LOST is returned.
    * Otherwise the move is legit and the Player is moved to b.
    * Then, if the goblet is at b, stop() is called and Result.WIN is returned.
    * Otherwise, Result.SUCCESS is returned.
    * @param a The Location to move the Player from.
    * @param b The Location to move the Player to.
    * @return The result of the call.
@throws IllegalArgumentException if there is no Player at $a$ or if $a$ is out of bounds.

Result move(Location a, Location b);

```java
public static void main(String[] args) {
    Direction[] directions = Direction.values(); // Get all enum values in array.
    int[] counter = {0};
    Strategy[] strategies = {
        /* @TODO (A) */ ,
        /* @TODO (A) */
    };
    for (int i = 0; i < 9; i++) System.out.print(strategies[1].nextMove() + " ");
    int size = Integer.parseInt(args[0]);
    Board board = /* @TODO (B) */; // Prints: UP DOWN RIGHT LEFT UP DOWN RIGHT LEFT UP
    int size = Integer.parseInt(args[0]);
    Board board = /* @TODO (B) */;
    board.start();
}
```

/** Generates a set of $n$ random Locations within the bounds of $size \times size$. */
private static Set<Location> getRandomLocations(int size, int n)

(נקודות (6 נקודות)

(נקודות (14 נקודות)
שאלה 2

عقد ת活動י בקורים "ס-קלקל" מבקרון מחוץ לטור הצביבה לטקסים בלטש. על פ נוריו, הת chociaż
לדעת למלל אוף טמסים כלא באוף ילי. המשודרגים חמד את הקוד הבא:

```cpp
class TestRes{
    private:
        char* description_;
};
class Test{
    public:
        virtual TestRes run() = 0;
};
class TestImplA : public Test{
    public:
        virtual TestRes run() { TestRes tr("OK!"); return tr; } //assume ctor exists
};
class TestSuite{
    public:
        unsigned int getSize() const { return size_; }
        void setSize(unsigned int i) const { size_ = i; }
    Test*
        getTest(unsigned int i) const { return test_arr_[i]; }
    void
        setResult(unsigned int i, TestRes res) { result_arr_[i] = res; }
        void
        initImplA(unsigned int n);
    private:
        Test** test_arr_;
        TestRes* result_arr_;
        unsigned int size_; //size of test_arr_ and result_arr_
};
void run_tests(TestSuite &ts){
    for (unsigned int i = 0; i < ts.getSize(); i++) {
        Test* test = ts.getTest(i);
        ts.setResult(i, test->run());
    }
}
int main(){
    TestSuite ts1;
    ts1.initImplA(10);
    run_tests(ts1);
    TestSuite ts2 = ts1;
    run_tests(ts1);
    return 0;
}
```
The function needs to create a TestSuite and a TestImplA instance.

```cpp
// (code)

TestSuite ts2 = ts1; // Add the line that adds the required for the class
```

After checking the students' work closely, it was found that the line:

```cpp
ts.setResult(i, test->run());
```

of the 5 rule:

A student in the course suggested changing the class so that:

```cpp
// ActorThreadPool
    this.threads = new ActorThreadPool(threads);
```

Alternatively, they could have only used one Executor instead of two, as follows:

```cpp
    this.threads = Executors.newFixedThreadPool(threads);
```

(0 points)

A question was asked whether in the implementation of the student's suggestion you could give up the Executor and use only one.

To answer this question, you can add a method in the class:

```cpp
    float ioThroughput () {
        //@TODO
        return (byteRead+byteWritten)/time;
    }
```

```cpp
    float compThroughput () {
        //@TODO
        return processedMessages/(time*threadPoolSize);
    }
```
ioThroughput

\[
\text{ioThroughput} = \frac{\text{byteRead} + \text{byteWritten}}{\text{time}}
\]

compThroughput

\[
\text{compThroughput} = \frac{\text{processedMessages}}{\text{time} \times \text{threadPoolSize}}
\]

\(\text{processedMessages}\) - The number of messages processed by the protocol for each client.

\(\text{threadPoolSize}\) - The size of the thread pool.

\(\text{time}\) - The time elapsed since the server started running in milliseconds.

\(\text{byteRead}\) - The number of bytes read by the server.

\(\text{byteWrite}\) - The number of bytes written to the client by the server.

\(\text{System.currentTimeMillis}\) - Returns the current time in milliseconds.

The methods ioThroughput and compThroughput are designed to be used in implementations of the methods in the original code in the appendix. If you cannot answer a question, it is because the methods in the answer refer to the source code, which is not included in this document.

The customer layer supports internet-style glasses. In this question, you are required to create the following store:

- Adding glasses to stock: each pair has a model number and price.
- Registering a new customer: the customer will fill out the following details: name and phone number.
- Purchase: the customer enters their phone number and the model number of the glasses they wish to purchase.
- The store will issue a receipt containing details of the purchase and a unique number that identifies it (receipt number).
- Price confirmation: the store owner can request a printout of the price of glasses of a specific model.
- Receipts: the store owner can request a printout of all receipts that have been issued.

To fulfill these requirements, you must create functions in Python that use the SQLite library to interact with the database. These functions should be implemented as DAO (Data Access Object) classes. To create these classes, you must:

1. Create a connection to the database using the sqlite3 module.
2. Select a DAO class for each table defined in the persistence layer.
3. For each DAO class, create functions for insert, find_all, and update operations.
4. Make sure you test your functions to ensure they work correctly.
בכל מהלך, לא צור ולזייר פעולות אלא החלק, לפי הצרכים הנתונים מעלה לא דורשת •
(לדוגמה,CEED של מהלך של פעולות של delete •
לפי צרכי החלק, בנפש יש מהלך של פעולות של DAO
בכל מחלקת)
לפי צרכי החנות, בנוסף יש מחלקות שלא דורשות
בפונקציות השונות או צור לזרז ולהזמין כל או לעימד בשאלה.

לנוחיותך מצורפת בנחת חוד עבה (שמות מהלך, הפונקציות בתבנית הם לעיזורlandırונא בלבד)

```python
def create_tables(conn):
    conn.execute("CREATE TABLE ...
")

#Data Transfer Objects
class Dto1(object):
    def __init__(self, ...):
...

#Data Access Objects
class Dao1(object):
    def __init__(self, conn):
        self._conn = conn

def insert(self, dto1):
    self._conn.execute("INSERT INTO ...
")

def find_all(self):
    c = self._conn.cursor()
    c.execute("SELECT ... FROM ...
")
    return [Dto1(*row) for row in c.fetchall()]
...
```
public interface MessageEncoderDecoder<T> {
    T decodeNextByte(byte nextByte);
    byte[] encode(T message);
}

public interface MessagingProtocol<T> {
    T process(T msg);
    boolean shouldTerminate();
}

public class Reactor<T> implements Server<T> {
    private final int port;
    private final Supplier<MessagingProtocol<T>> protocolFactory;
    private final Supplier<MessageEncoderDecoder<T>> readerFactory;
    private final ActorThreadPool pool;
    private Selector selector;
    private Thread selectorThread;
    private final ConcurrentLinkedQueue<Runnable> selectorTasks = new ConcurrentLinkedQueue<>();

    public Reactor(
        int numThreads,
        int port,
        Supplier<MessagingProtocol<T>> protocolFactory,
        Supplier<MessageEncoderDecoder<T>> readerFactory) {
        this.pool = new ActorThreadPool(numThreads);
        this.port = port;
        this.protocolFactory = protocolFactory;
        this.readerFactory = readerFactory;
    }

    @Override
    public void serve() {
        selectorThread = Thread.currentThread();
        try (Selector selector = Selector.open();
            ServerSocketChannel serverSock = ServerSocketChannel.open()) {
            this.selector = selector; //just to be able to close
            serverSock.bind(new InetSocketAddress(port));
            serverSock.configureBlocking(false);
            serverSock.register(selector, SelectionKey.OP_ACCEPT);
            while (!Thread.currentThread().isInterrupted()) {
                selector.select();
                runSelectionThreadTasks();
                for (SelectionKey key : selector.selectedKeys()) {
                    if (!key.isValid()) {
                        continue;
                    } else if (key.isAcceptable()) {
                        handleAccept(serverSock, selector);
                    } else {
                        handleReadWrite(key);
                    }
                }
                selector.selectedKeys().clear(); //clear the selected keys set so that we can know about new events
            }
        } catch (ClosedSelectorException ex) {
            //do nothing - server was requested to be closed
        } catch (IOException ex) {
            //this is an error
            ex.printStackTrace();
        }
    }
}
System.out.println("server closed!!!");
pool.shutdown();

void updateInterestedOps(SocketChannel chan, int ops) {
    final SelectionKey key = chan.keyFor(selector);
    if (Thread.currentThread() == selectorThread) {
        key.interestOps(ops);
    } else {
        selectorTasks.add(() -> {
            key.interestOps(ops);
        });
        selector.wakeup();
    }
}

private void handleAccept(ServerSocketChannel serverChan, Selector selector) throws IOException {
    SocketChannel clientChan = serverChan.accept();
    clientChan.configureBlocking(false);
    final NonBlockingConnectionHandler handler = new NonBlockingConnectionHandler(
        readerFactory.get(),
        protocolFactory.get(),
        clientChan,
        this);
    clientChan.register(selector, SelectionKey.OP_READ, handler);
}

private void handleReadWrite(SelectionKey key) {
    NonBlockingConnectionHandler handler = (NonBlockingConnectionHandler) key.attachment();
    if (key.isReadable()) {
        Runnable task = handler.continueRead();
        if (task != null) {
            pool.submit(handler, task);
        }
    }
    if (key.isValid() && key.isWritable()) {
        handler.continueWrite();
    }
}

private void runSelectionThreadTasks() {
    while (!selectorTasks.isEmpty()) {
        selectorTasks.remove().run();
    }
}

@Override
public void close() throws IOException {
    selector.close();
}
public class NonBlockingConnectionHandler<T> implements ConnectionHandler<T> {
    private static final int BUFFER_ALLOCATION_SIZE = 1 << 13; //8k
    private static final ConcurrentLinkedQueue<ByteBuffer> BUFFER_POOL = new
            ConcurrentLinkedQueue<>();

    private final MessagingProtocol<T> protocol;
    private final MessageEncoderDecoder<T> encdec;
    private final Queue<ByteBuffer> writeQueue = new ConcurrentLinkedQueue<>();
    private final SocketChannel chan;
    private final Reactor reactor;

    public NonBlockingConnectionHandler(
            MessageEncoderDecoder<T> reader,
            MessagingProtocol<T> protocol,
            SocketChannel chan,
            Reactor reactor) {
        this.chan = chan;
        this.encdec = reader;
        this.protocol = protocol;
        this.reactor = reactor;
    }

    public Runnable continueRead() {
        ByteBuffer buf = leaseBuffer();
        boolean success = false;
        try {
            success = chan.read(buf) != -1;
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        if (success) {
            buf.flip();
            return () -> {
                try {
                    while (buf.hasRemaining()) {
                        T nextMessage = encdec.decodeNextByte(buf.get());
                        if (nextMessage != null) {
                            T response = protocol.process(nextMessage);
                            if (response != null) {
                                writeQueue.add(ByteBuffer.wrap(encdec.encode(response)));
                                reactor.updateInterestedOps(chan, SelectionKey.OP_READ | SelectionKey.OP_WRITE);
                            }
                        }
                    }
                } finally {
                    releaseBuffer(buf);
                }
            };
        } else {
            releaseBuffer(buf);
            close();
            return null;
        }
    }

    public void close() {
        try {
            chan.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public boolean isClosed() {
        return true;
    }
}
return !chan.isOpen();
}

public void continueWrite() {
    while (!writeQueue.isEmpty()) {
        try {
            ByteBuffer top = writeQueue.peek();
            chan.write(top);
            if (top.hasRemaining()) {
                return;
            } else {
                writeQueue.remove();
            }
        } catch (IOException ex) {
            ex.printStackTrace();
            close();
        }
    }
}

if (writeQueue.isEmpty()) {
    if (protocol.shouldTerminate()) close();
    else reactor.updateInterestedOps(chan, SelectionKey.OP_READ);
}

private static ByteBuffer leaseBuffer() {
    ByteBuffer buff = BUFFER_POOL.poll();
    if (buff == null) {
        return ByteBuffer.allocateDirect(BUFFER_ALLOCATION_SIZE);
    }
    buff.clear();
    return buff;
}

private static void releaseBuffer(ByteBuffer buff) {
    BUFFER_POOL.add(buff);
}

public class ActorThreadPool {
    private final Map<Object, Queue<Runnable>> acts;
    private final ReadWriteLock actsRWLock;
    private final Set<Object> playingNow;
    private final ExecutorService threads;

    public ActorThreadPool(int threads) {
        this.threads = Executors.newFixedThreadPool(threads);
        acts = new WeakHashMap<>();
        playingNow = ConcurrentHashMap.newKeySet();
        actsRWLock = new ReentrantReadWriteLock();
    }

    public void submit(Object act, Runnable r) {
        synchronized (act) {
            if (!playingNow.contains(act)) {
                playingNow.add(act);
                execute(r, act);
            } else {
                pendingRunnablesOf(act).add(r);
            }
        }
    }

    public void shutdown() {
        threads.shutdownNow();
    }
}
private Queue<Runnable> pendingRunnablesOf(Object act) {
    actsRWLock.readLock().lock();
    Queue<Runnable> pendingRunnables = acts.get(act);
    actsRWLock.readLock().unlock();
    if (pendingRunnables == null) {
        actsRWLock.writeLock().lock();
        acts.put(act, pendingRunnables = new LinkedList<>());
        actsRWLock.writeLock().unlock();
    }
    return pendingRunnables;
}

private void execute(Runnable r, Object act) {
    threads.execute(() -> {
        try {
            r.run();
        } finally {
            complete(act);
        }
    });
}

private void complete(Object act) {
    synchronized (act) {
        Queue<Runnable> pending = pendingRunnablesOf(act);
        if (pending.isEmpty()) {
            playingNow.remove(act);
        } else {
            execute(pending.poll(), act);
        }
    }
}