This document contains a programming problem related to producer-consumer patterns and task management. The solution involves implementing a class `ProducingTask` and `PurifyingTask` that interact with each other through a `purifiedPool` variable.

```java
class ProducingTask implements Runnable {
    ...
    public void run() {
        while (!Thread.interrupted()) {
            _purifiedPool.add(_purifiedPool.getCapacity());
        }
    }
}

class PurifyingTask implements Runnable {
    ...
    public void run() {
        ...
    }
}
```
while (!Thread.interrupted())
    _purifiedPool.add(_purifier.purify(_usedPool.remove()));
}
}
class ConsumingTask implements Runnable {
...
public void run() {
    while (!Thread.interrupted()) {
        float w = _purifiedPool.remove();
        try { Thread.sleep((long)w); } catch (InterruptedException e) { Thread.currentThread().interrupt(); }
        _usedPool.add(w);
    }
}
}
class WaterPool implements Pool {
...
public synchronized float remove() {
    while (_content == 0)
        try { wait(); } catch (InterruptedException e) { Thread.currentThread().interrupt(); return 0; }
    float ret = _content;
    _content = 0;
    notifyAll();
    return ret;
}
public synchronized void add(float addition) {
    while (_content + addition > _capacity)
        try { wait(); } catch (InterruptedException e) { Thread.currentThread().interrupt(); return; }
    _content += addition;
    notifyAll();
}
}
class WaterPool implements Pool {
    private final long _capacity;
    private float _content;
    private Object _monitorInc, _monitorDec;

    WaterPool(float content, long capacity) {
        _content = content; _capacity = capacity;
        _monitorInc = new Object(); _monitorDec = new Object();
    }

    public long getCapacity() { return _capacity; }
    public synchronized float getContent () { return _content; }
    public synchronized float remove() {
        while (_content == 0)
            synchronized (_monitorInc) {
                try (_monitorInc.wait();) catch (InterruptedException e) { return 0; }
            }
        float ret = _content;
        _content = 0;
        synchronized (_monitorDec) {
            try {_monitorDec.notifyAll();} catch (InterruptedException e) { return; }
        }
        return ret;
    }

    public synchronized void add(float addition) {
        while (_content + addition > _capacity)
            synchronized (_monitorDec) {
                try (_monitorDec.wait();) catch (InterruptedException e) { return; }
            }
        _content += addition;
        synchronized (_monitorInc) {
            try (_monitorInc.notifyAll();) catch (InterruptedException e) { return; }
        }
    }
}

הערה: סכנת deadlock קיימת בסכנת deadlocks. במועד זה של שנה זו, נתקশו המחלקות מאבדים יקרים חלולים.(deadlock)
deadlock, Semaphore, Pool, WaterPool, Deadlock
deadlock.
שאלה 2

סעיף א (7 נקודות)

 tabela

<table>
<thead>
<tr>
<th>הערך</th>
<th>מספר נקודות</th>
</tr>
</thead>
<tbody>
<tr>
<td>5012</td>
<td>1</td>
</tr>
<tr>
<td>5008</td>
<td>*vtable</td>
</tr>
<tr>
<td>5004</td>
<td>1200</td>
</tr>
<tr>
<td>5000</td>
<td>1200</td>
</tr>
</tbody>
</table>


סעיף ב (8 נקודות)

 tabela

<table>
<thead>
<tr>
<th>הערך</th>
<th>מספר נקודות</th>
</tr>
</thead>
<tbody>
<tr>
<td>5020</td>
<td>20</td>
</tr>
<tr>
<td>5016</td>
<td>2</td>
</tr>
<tr>
<td>5012</td>
<td>*vtable</td>
</tr>
<tr>
<td>5008</td>
<td>1000</td>
</tr>
<tr>
<td>5004</td>
<td>1200</td>
</tr>
<tr>
<td>5000</td>
<td>1400</td>
</tr>
</tbody>
</table>


IFDEFG (7 נקודות)

EventData::EventData(const EventsStack &other):_top(0),_last(0) {
    EventNode *cursor = other._last;
    while (cursor != 0) {
        push(cursor->data);
        cursor = cursor->prev;
    }
}


IFDEFD (8 נקודות)

void EventsStack::pop() {
    if (_top == 0) return;
    EventNode *oldTop = _top;
    _top = _top->next;
    if (_top != 0) {
        /* ...
    */
}
What must be change in StompOperatorImp to work with a message receiver:
1. Remove the getMessages message implementation.
2. Add a messageReceiver parameter to the constructor.
3. Pass the messageReceiver parameter to the Listener.
4. Change the Listener so that it passes the received messages to the receiver as soon as received.
5. Remove the _msgs member.

```java
public class StompOperatorImpl extends java.rmi.server.UnicastRemoteObject, implements StompOperator {
    protected PrintWriter _writer;
    protected BufferedReader _reader;
    private List<String> _msgs;

    // Define a listener task to receive the messages from the Stomp server.
    private class Listener implements Runnable {
        private MessageTokenizer _tok;
        private MessageReceiver _mr;

        public Listener(MessageReceiver mr) {
            _tok = new MessageTokenizer(_reader, '\0');
            _mr = mr;
        }
        public void run() throws IOException {
            while (_tok.isAlive())
                synchronize (_msgs) { _msgs.add(_tok.nextToken());
            _mr.message(_tok.nextToken());
        }
    }

    public StompOperatorImpl(InputStream in, OutputStream out, MessageReceiver mr)
        throws java.rmi.RemoteException, IOException {
        _reader = new BufferedReader(new InputStreamReader(in, "UTF-8");
```
The required changes in StompClient are:
- Create a MessageReceiver instance to handle incoming messages.
- Pass this instance as a parameter to the StompConnector.
- Remove the loop that calls getMessages – since messages are now handled by the receiver.
We want the client to send back to the StompServer a message each time the client receives a message. NOTE: the decision to send back the message MUST be made in the StompClient – it cannot be only in the StompClientRMI process – it must be triggered by code that lives inside the StompClient process. (This is indicated in the question by the sentence “the message is sent from the memory of the StompClient process”).

There were several possible answers. The best one is one where ALL the changes are only in the StompClient side – without any changes to the StompClientRMI.

The way for a StompClient to react to the event of a message arriving is to implement a new version of MessageReceiver. So this is where we start.

The way for a StompClient to send a message to the StompServer is to use the StompOperator send() method.

Let’s put these 2 things together in this solution:

```java
public class MessageReceiverImpl2 extends java.rmi.server.UnicastRemoteObject implements MessageReceiver {
    private StompOperator _sender = null;
    MessageReceiverImpl2() throws java.rmi.RemoteException {
    }
    void setSender(StompOperator s) {_sender = s;}
    public void message(String str) throws java.rmi.RemoteException {
        if (_sender != null) _sender.send("q1", str);
    }
}

public class StompClient {
    public static void main(String[] args) {
        try {
            MessageReceiverImpl2 mr = new MessageReceiverImpl2();
            StompConnector c = (StompConnector)Naming.lookup("rmi://132.23.5.8:2010/StompConnector");
            StompOperator s = c.connect("user", "password", mr);
            mr.setSender(s);
            s.subscribe("q1"); s.subscribe("q2"); s.send("q3", "Suzy Surprise");
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```
Another acceptable solution (less nice because it causes more changes to more components) was to change the StompClientRMI code and the MessageReceiver interface so that message(str) returns a string instead of being a void method.

```java
public interface MessageReceiver extends java.rmi.Remote {
    // If message returns a non-null value, then send this value back to the sender.
    String message(String str) throws java.rmi.RemoteException;
}
```

Then change the Listener class inside StompOperatorImpl:

```java
private class Listener implements Runnable {
    private MessageTokenizer _tok;
    private MessageReceiver _mr;
    public Listener(MessageReceiver mr) {
        _tok = new MessageTokenizer(_reader, '\0');
        _mr = mr;
    }
    public void run() throws IOException {
        while (_tok.isAlive()) {
            String answer = _mr.message(_tok.nextToken());
            if (answer != null) {
                _writer.send("q1", answer);
            }
        }
    }
}
```

And finally change the MessageReceiverImpl:

```java
public class MessageReceiverImpl extends java.rmi.server.UnicastRemoteObject implements MessageReceiver {
    MessageReceiverImpl() throws java.rmi.RemoteException { }
    public String message(String str) throws java.rmi.RemoteException {
        return "Got your message: " + str;
    }
}
```
CREATE TABLE Planes (  
    ID integer PRIMARY KEY,  
    NextMaintenancePlanned Date,  
    BusinessClassCapacity integer,  
    -- Could also be put in the Flights or in the PlaneModels table  
    Model varchar(20) FOREIGN KEY REFERENCES PlaneModels(Model)
)

CREATE TABLE Flights (  
    ID integer PRIMARY KEY,  
    Destination varchar(100),  
    Terminal varchar(20),  
    ExitGate varchar(20),  
    ExitDate date,  
    ExitTime time,  
    PilotName varchar(100),  
    PlanId integer FOREIGN KEY REFERENCES Planes(ID)
)

Remarks:
- Different pilots can fly the same plane. Therefore the pilot name is located in the Flights table and not in the Planes table.
- The planned maintenance is specific to each plane instance. It does not depend on each flight. Therefore it is in the Planes table.
- The business class capacity generally is adjusted for each flight within a maximum that depends on the plane model. So the most likely setting would be to have the field in the Flights table and a maximum value in the PlaneModels table.

Select Flights.ID, Planes.BusinessClassCapacity  
From Flights JOIN Planes on Flights.PlanId = Planes.Id  
Order by Flights.destination asc