Beispiel Klausuraufgaben

HINWEIS: In diesem Dokument befinden sich mehrere Aufgaben. Es ist keine Beispielklausur.

In der Klausur werden nur ca. 2 Aufgaben zu meinen Themengebieten (Nebenläufigkeit, Visuelle Programmierung, Entwurfsmuster) gestellt.

Aufgabe 1: Welche der folgenden Methoden sind threadsafe?

```
public class DemoParallel {


	final private int scaleFactor = 2;
	private int sum=0;
	private int counter = 0;



	public int square (int x) {

		return x * x;

	}

	public int addNumber (int n) {

		sum += n;

		return sum;

	}


	public void addSquareNumber (int n) {


		sum += n*n;


		counter++;



	}

	public double getScaleFactor () {

		return scaleFactor / 100.0 ;

	}



}
```

Begründen Sie Ihre Antworten.
Ergänzen Sie den Programmcode so, dass die Klasse threadsafe wird und möglichst wenig blockiert.

Lösung Aufgabe 1
(ergänzter Code in rot)

square ist threadsafe, da nur lokale Variable auf dem Stack verändert werden.

addNumber ist nicht threadsafe, da mehrere Threads gleichzeitig auf sum zugreifen können und Wettlaufbedingungen auftreten. Z.B. kann zwischen dem Addieren von n und dem Rückgabewerte von n ein zweiter Thread den Wert von sum schon wieder verändert haben.

Alternative Begründung: Die Anweisung sum+=n wird nicht atomar auf dem Prozessor ausgeführt. So kann es passieren, dass ein Thread den gerade geänderten Wert eines anderen Thread überschreibt. Die Folge sind inkonsistente Werte.

Mit dem synchronized – Schlüsselwort können Sie die Methode threadsicher gestalten:

```
public synchronized int addNumber (int n) {

        sum += n;

        return sum;

}
```

addSquareNumber ist nicht threadsicher, da mehrere Threads gleichzeitig auf sum oder counter zugreifen können und es Wettlaufbedingungen kommen kann.

Threadsicherheit lässt sich durch Objektsperren herstellen. Damit das Programm möglichst wenig blockiert wird für die unabhängigen Daten sum und counter jeweils eine andere Sperre verwendet. In Java kann jedes Objekt als Sperre verwendet werden, wir erzeugen also einfach zwei Objekte.

```
Object sumGuard = new Object ();
Object counterGuard = new Object ();
public void addSquareNumber (int n) {
   synchronized (sumGuard) {
      sum += n*n;
    }
    synchronized (counterGuard) {
      counter++;
     }

}
```

getScaleFactor ist threadsicher, da nur auf unveränderliche Werte zugegriffen wird.

Aufgabe 2: Nebenläufigkeit mit Actors

Im Folgenden sehen Sie einen in Java geschriebenen Programmcode, der den Inhalt von Dateien lädt, den Inhalt verschlüsselt und dann wieder als Datei speichert. Das Speichern soll nebenläufig in einem eigenen Thread geschehen.

Die main-Methode übergibt die verschlüsselten Daten mit einem neuen Dateinamen an eine Warteschlange (BlockingQueue). Der FileSaver läuft in einem eigenen Thread und bedient sich fortlaufend aus dieser Warteschlange.

```java
// Hilfsklasse als Datencontainer
public class EncryptedData {
	public String fileName;
	public String data;
}



public class Setup {

	public static void main(String[] args) {

	 // Warteschlange für die Daten, die gespeichert werden sollen
	BlockingQueue <EncryptedData> saveQueue =
			new LinkedBlockingQueue<EncryptedData>();

		// Dieser Thread erledigt das Speichern:
		Thread saving = new Thread ( new FileSaver (saveQueue));
		saving.start();

		// Alle als Argument übergebenen Dateien sollen verschlüsselt werden:
		for (int i=0; i<args.length; i++) {
			try {
				// Datei laden, Inhalt verschlüsseln und als
				// EncryptedData-Objekt in die
				// Warteschlange zum Speichern legen
				String fileName = args [i];
				String content = UtilsClass.loadFile (fileName);
				EncryptedData ec = new EncryptedData();
				ec.fileName = fileName + ".encrypted";
				ec.data = UtilsClass.encrypt(content);
				saveQueue.put(ec);
				System.out.println("Unable to encrypt all files.");
			}
		}
	}


public class FileSaver implements Runnable {

	BlockingQueue<EncryptedData> saveQueue;

	public FileSaver (BlockingQueue<EncryptedData> saveQueue ) {
		this.saveQueue = saveQueue;
	}

	public void run() {
		 while (true) {
			try {
				// Nimm das nächste Element aus der Warteschlange
				// und speichere es ab.
				EncryptedData ec =  saveQueue.take();
				UtilsClass.saveFile(ec.fileName, ec.data);


			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		 }
	}
```

Schreiben Sie ein Programm mit gleicher Funktionalität in Scala unter Verwendung von Actors. Nutzen Sie eine Case Klasse für die Nachrichtenübermittlung.

Hinweis: um die Laden, Speichern, und Verschlüsseln Funktionen müssen Sie sich nicht kümmern, diese sind über einfach über ein UtilsObject vorhanden.

```
import scala.actors.Actor._

object Setup {

  case class EncryptedData ( fileName : String , data : String)

  def main(args: Array[String]): Unit = {

// TODO: Definieren und starten sie den Actor zum Speichern hier:




// Hier werden bereits alle Dateien geladen und verschlüsselt
// Sie müssen aber noch die Nachricht zum Speichern an den
// Actor senden

    for (fileName <- args) {
      val content = UtilsObject.loadFile(fileName)
      val encrypted = UtilsObject.encrypt(content)


      // TODO: Senden Sie die Nachricht zum Speichern hier:



    }
  }
}
```

Lösung Aufgabe 2
(ergänzter Code in rot)

```
import scala.actors.Actor._

object Setup {

  case class EncryptedData ( fileName : String , data : String)

  def main(args: Array[String]): Unit = {

// TODO: Definieren und starten sie den Actor zum Speichern hier:

    val saveActor = actor {
            while(true) {
                  receive {
                    case ec : EncryptedData =>
                       UtilsObject.saveFile(ec.fileName , ec.data )
                  }
            }
    }


// Hier werden bereits alle Dateien geladen und verschlüsselt
// Sie müssen aber noch die Nachricht zum Speichern an den
// Actor senden

    for (fileName <- args) {
         val content = UtilsObject.loadFile(fileName)
              val encrypted = UtilsObject.encrypt(content)

      // TODO: Senden Sie die Nachricht zum Speichern hier:
              saveActor ! EncryptedData(content,encrypted)

     }



  }

}
```

Aufgabe 3: Deadlocks

Was ist ein Deadlock?

Beispiellösung: Eine Verklemmung (deadlock) ist eine durch eine zyklische Abhängigkeit bewirkte Verhinderung der Programmausführung. Die Abhängigkeit zwischen verschiedenen Threads entstehen, wenn diese um gemeinsame Ressourcen (z.B. Objektsperren) konkurrieren.

Nennen Sie ein Alltagsbeispiel

Beispiellösung: In eine enge Straße fahren zwei Autos von gegenüberliegenden Seiten hinein. Die Autos kommen nicht aneinander vorbei. Jeder Fahrer beharrt auf sein Recht als erster zu fahren und weigert sich zurückzusetzen – das kostet ja Zeit. Also stehen beide Autos dort für ewig…

Nennen Sie eine Maßnahme, mit der sich Deadlocks vermeiden lassen.

Beispiellösungen (eine Antwort würde reichen):

- Vermeiden von Methodenaufrufen innerhalb eines gesperrten Bereichs, weil diese Methoden eventuell weitere Sperren nutzen. Deadlocks entstehen immer nur wenn mind. zwei Sperren involviert sind.
- Möglichst kurze Codeabschnitte sperren, innerhalb eines kritischen Abschnitts keine blockierenden Methoden aufrufen, z.B. put oder take einer BlockingQueue nicht aus einem kritischen Abschnitt heraus aufrufen
- Für voneinander unabhängige Daten sollten verschiedene Sperren genutzt werden

Warum können Maßnahmen zur Threadsicherheit die Wahrscheinlichkeit eines Deadlocks erhöhen?

Beispiellösung:

Um Wettlaufbedingungen zu vermeiden, werden häufig Objektsperren eingesetzt, mit denen man kritische Bereiche schützt. Wenn zwei kritische Abschnitte gleichzeitig gesperrt sind, kann es vorkommen, dass zwei Threads gegenseitig aufeinander warten.

Warum lässt sich schwer testen, ob ein Programm Deadlock-Situationen enthält?

Beispiellösung:

Aufgrund des nicht-deterministischen Ablaufs nebenläufiger Programme treten Deadlocks meist nur unter bestimmten (oft zufälligen) Konstellationen auf. Es kann z.B. sein, dass ein Deadlock nur bei jedem 100000. Mal auftritt, während 99999 Testläufe vorher problemlos funktionierten. Es kann ssein

Aufgabe 4: Entwurfsmuster

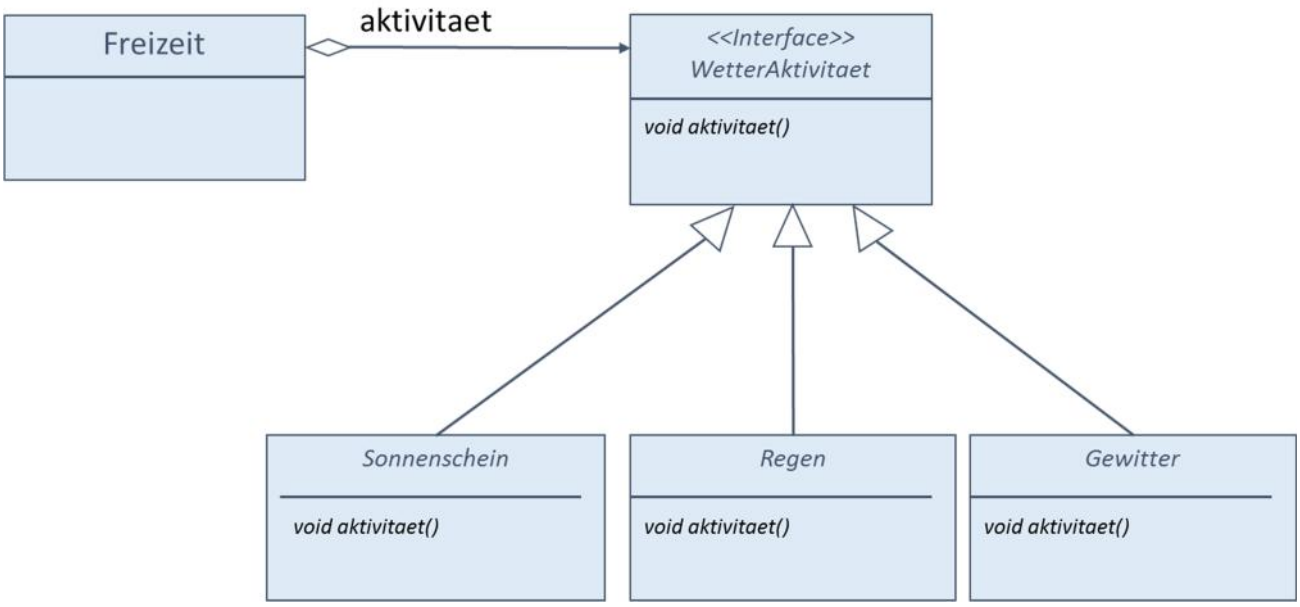
Gegeben sei folgender Code, der je nach Wetterlage verschiedene Aktivitäten startet.

```
public class Freizeit {


	final int SONNE = 1;
	final int REGEN = 2;
	final int SCHNEE = 3;
	final int STURM = 4;

	int wetter = SONNE;


	public void setWetter (int neuesWetter) {wetter = neuesWetter;}


	public void aktivitaet () {

		if (wetter == SONNE) {
			p.eisEssen ();
			p.setLaune ( "super") ;

		}

		if (wetter == REGEN) {
			p.mantelAnziehen();
			p.setLaune ("mies");
		}

		if (wetter == SCHNEE) {
			p.mantelAnziehen();
			p.setLaune("gut");

		}

	}

}
```

Mit welchem Entwurfsmuster lässt sich der Code verbessern?
Zeichnen Sie das Klassendiagramm für eine verbesserte Struktur.

Lösung Aufgabe 4

Das Strategie Muster (Strategy Patter) kann hier helfen:



Aufgabe 5: Visuelle Programmierung

Skizzieren Sie ein datenflussorientiertes Programm, mit dem Sie folgenden Ausdruck berechnen:

ergebnis = power (5 * 2 + 34 , 3)

Sie dürfen eigene Symbole und Zeichen erfinden, aber das datenflussorientierte Paradigma muss erkennbar sein.

Lösungsbeispiel Aufgabe 5