Software Engineering: Design & Construction

Department of Computer Science
Software Technology Group

TECHNISCHE UNIVERSITÄT DARMSTADT

Final Exam **Sample Solution** **February 14, 2016**

First Name	
Last Name	
Matriculation Number	
Course of Study	
Department	
Signature	

Permitted Aids

- Everything except electronic devices
- Use a pen with document-proof ink (No green or red color)

Announcements

- Make sure that your copy of the exam is complete (19 pages).
- Fill out all fields on the front page.
- Do not use abbreviations for your course of study or your department.
- Put your name and your matriculation number on all pages of this exam.
- The time for solving this exam is 90 minutes.
- All questions of the exam must be answered in English.
- You are not allowed to remove the stapling.
- You can use the backsides of the pages if you need more space for your solution.
- Make sure that you have read and understood a task before you start answering it.
- It is not allowed to use your own paper.
- Sign your exam to confirm your details and acknowledge the above announcements.

Topic	1	2	3	4	Total
Points					
Max.	26	29	14	21	90

Name: _____ Matriculation Number: ⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋

Topic 1: Introduction — 26P

a) True or False — 6P

For each of the following statements, indicate whether it is true by writing ***yes*** or ***no*** below each statement. Shortly say why it is true or not. (1.5P per answer)

1. Scala traits and Java 8 interfaces with default methods are equally expressive.

2. In Scala, pattern matching can be an alternative to the visitor pattern.

3. The Fragile Base Class Problem can always be avoided by properly documenting your classes.

4. The Liskov Substitution Principle is different from the other S.O.L.I.D. principles in that it is about the correctness of a program, whereas the other principles are not.

Solution

1.5P for each. For false statements, 0.5P for the ***yes / no*** and 1P for the reasoning.

1. No. For example, interfaces don't have state.
2. Yes. Instead of adding a visit method for each concrete element to the visitor class, a case statement is added to the match expression.
3. No, since no one can anticipate all possible expectations a subclass could make.
4. Yes. LSP is

Name: _____ Matriculation Number: ⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋

b) Basic Knowledge — 11P

1. Take a look at the following Java code:

```
class Foo {
    public void exec(Consumer<String> f, String name) {
        f.accept(name);
    }
}

public class Closure {
    public static void main(String[] args) {
        String hello = "Hello";

        Consumer<String> sayHello =
                        (String name) -> System.out.println(hello + ", " + name);

        Foo foo = new Foo();
        foo.exec(sayHello, "Al");

        hello = "Hola";
        foo.exec(sayHello, "Lorenzo");
    }
}
```

This code will not compile. Shortly state why this code won't compile. Write Scala code that models the same functionality but does compile. Also shortly sketch what you would have to change in the Java code to make it work. (6P)

Solution

Scala:

```
class Foo {
  def exec(f: String => Unit, name: String): Unit = {
    f(name)
  }
}

var hello = "Hello"
def sayHello(name: String): Unit = {
  println(s"$hello, $name")
}

val foo = new Foo
foo.exec(sayHello, "Al")

hello = "Hola"
foo.exec(sayHello, "Lorenzo")
```

Java:

```
class Foo {
    public void exec(Consumer<String> f, String name) {
        f.accept(name);
    }
```

```
}

class Greeter {
    public String hello;
}


public class Closure {

    public static void main(String[] args) {

        final Greeter greeter = new Greeter();
        greeter.hello = "Hello";

        Consumer<String> sayHello =
            (String name) -> System.out.println(greeter.hello + ", " + name);

        Foo foo = new Foo();
        foo.exec(sayHello, "Al");

        greeter.hello = "Hola";
        foo.exec(sayHello, "Lorenzo");
    }
}
```

2. In which way do traits or mixin composition help us to close the representational gap compared to other languages? (2P)

Solution

Various aspects in the real world are often composed out of smaller pieces. Traits and mixin composition allows us to model them more realistic at some points.

3. Which part of the class `Record` has to be documented when the class is intended to be used by subclassing? Describe the relation to the Fragile Base Class Problem and give a concrete example. (3P)

```
class Record {
   private[this] var rows = List.empty[Row]
   def addRow(row: Row): Unit = rows = row +: rows
   def deleteRow(row: Row): Unit = rows = rows filterNot { _ == row }

   def moveRow(row: Row, targetRecord: Record): Unit = {
      if(rows.contains(row)){
         this.deleteRow(row)
         targetRecord.addRow(row)
      }
   }
}
```

Solution

The self-call structure (1P) should be documented (moveRow calls addRow and deleteRow) (1P) The following implementation breaks if `moveRow` is changed to an implementation not using `addRow` or `deleteRow`. (1P)

```
class LogRecord extends Record {
    def log(msg: String) = println(msg)

    override def addRow(row: Row): Unit = {
      log("Add " + row)
      super.addRow(row)
    }

    override def deleteRow(row: Row): Unit = {
      log("Remove " + row)
      super.deleteRow(row)
    }
  }
```

Name: _____ Matriculation Number: ⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋

c) Traits and Mixin Composition 6P

Given the following class and trait definitions, give the class linearization for all traits and classes. You have to provide all intermediate steps of the linearization and you won't get any points for just writing down the final solution. The linearization of class `D` defined in the following class hierarchy

```
class A
trait B extends A
class C extends A
class D extends C with B
```

should look like the following:

```
Lin(A) = {A}
Lin(B) = {B, Lin(A)}
       = {B, A}
Lin(C) = {C, Lin(A)}
       = {C, A}
Lin(D) = {D, Lin(B) ≫ Lin(C)}
       = {D, {B, A} ≫ {C, A}}
       = {D, B, C, A}
```

Reuse linearization results if possible (e.g. the linearization of `D` can reuse the result of the linearization for `B` and `C`). You can use the following abbreviations for the traits and classes.

- `Gift` (G)
- `GiftBox` (GB)
- `WrappingPaper` (WP)
- `Ribbon` (R)
- `BoxedGift` (BG)
- `BoxedAndWrappedGift` (BWG)
- `DoubleBoxedAndWrappedGiftWithRibbon` (2B2WGR)

```
abstract class Gift {
  def packaging: List[String] = List()
}

trait GiftBox extends Gift {
  override def packaging: List[String] = "Gift Box" :: super.packaging
}

trait WrappingPaper extends Gift {
  override def packaging: List[String] = "Wrapping paper" :: super.packaging
}

trait Ribbon extends Gift {
  override def packaging: List[String] = "Ribbon" :: super.packaging
}

class BoxedGift extends Gift with GiftBox
class BoxedAndWrappedGift extends BoxedGift with WrappingPaper
class DoubleBoxedAndWrappedGiftWithRibbon extends BoxedAndWrappedGift with Ribbon
  with WrappingPaper with GiftBox
```

Name: _____ Matriculation Number: ⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋

(Code excerpt to reduce page turning)

```
abstract class Gift { ... }
trait GiftBox extends Gift { ... }
trait WrappingPaper extends Gift { ... }
trait Ribbon extends Gift { ... }
class BoxedGift extends Gift with GiftBox
class BoxedAndWrappedGift extends BoxedGift with WrappingPaper
class DoubleBoxedAndWrappedGiftWithRibbon extends BoxedAndWrappedGift with Ribbon
  with WrappingPaper with GiftBox
```

Solution

```
Lin(G)      = {G} (0.5P)
Lin(GB)     = {GB, G} (0.5P)
Lin(WP)     = {WP, G} (0.5P)
Lin(R)      = {R, G} (0.5P)
Lin(BG)     = {BG, Lin(GB) ≫ Lin(G)}
            = {BG, {GB, G} ≫ {G}}
            = {BG, GB, G} (1P)
Lin(BWG)    = {BWG, Lin(WP) ≫ Lin(BG)}
            = {BWG, {WP, G} ≫ {BG, GB, G}}
            = {BWG, WP, BG, GB, G} (1P)
Lin(2B2WGR) = {2B2WGR, Lin(GB) ≫ Lin(WP) ≫ Lin(R) ≫ Lin(BWG)}
            = {2B2WGR, {GB, G} ≫ {WP, G} ≫ {R, G} ≫, {BWG, WP, BG, GB, G}}
            = {2B2WGR, R, BWG, WP, BG, GB, G} (2P)
```

Name: _____ Matriculation Number: ⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋

d) Forwarding vs Delegation 3P

Sketch a template for how to implement delegation semantics in Scala (2-3 classes).

Solution

There is an additional m method that takes the caller (self) as an argument and can therefore delegate the call back to the caller.

```
trait Base {
  def m();
  def f();
}

class Callee extends Base {
  def m(self: Base) {
    ...
    self.f
  }
  def m() {
    m(this)
  }
  ...
}

class OtherClass extends Base {
  val callee = new Callee()
  def m() {
    callee.m(this)
  }
  ...
}
```

Name: _____ Matriculation Number: ⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋

Topic 2: Design Patterns 29P

The following tasks are related to the shown code. Please, read the tasks first.

```
trait Task {
  def name: String
  def duration: Duration
  def accept(v: Visitor): Unit
}

case class SimpleTask(name: String, duration: Duration) extends Task {
  def accept(v: Visitor): Unit = v visit this
}

trait DependentTask extends Task {
  def dependsOn: Task
}

trait TaskGroup extends Task {
  var tasks = List[Task]()

  def create(tasks: Task*): TaskGroup
  def add(t: Task): Unit = tasks = t :: tasks
  def duration: Duration = tasks.foldLeft(Duration.ZERO)(_ plus _.duration)
}

trait Visitor {
  def visit(a: SimpleTask)
  def visit(d: DependentTask)
}

trait OutputFormat {
  def format(tasks: List[Task]): String
}

case object CSV extends OutputFormat {
  private val output = new StringBuilder

  def format(tasks: List[Task]): String = {
    output append "Name, Duration\n"
    tasks foreach { t =>
      output append s"${t.name}, ${t.duration}"
    }
    output toString
  }
}

trait Export[T] {
  def result: T
}

class TaskList(t: List[Task], val f: OutputFormat) extends Export[String] {
  def result: String = f format t
}
```

Name: _____ Matriculation Number: ⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋

a) "Pattern Recognition" 12P

Identify ***all*** design patterns used in the previous Scala code. For each pattern instance, shortly state which classes are responsible for what role in the pattern. If the pattern has methods that are characteristic, name the corresponding methods in the code. You only need to name patterns which can be clearly identified.

Solution

- 2P Factory Method: All case classes have an apply method which is a factory method for creating tasks; `create` (1P) in `TaskGroup` (1P) is a factory method.
- 3P Visitor: Task = Element (0.5P), subclasses of Task = ConcreteElement (0.5P), Visitor = Visitor (1P), accept(visitor: Visitor) = accept method (0.5P), visit(...) = visit methods (0.5P).
- 3P Composite: `Task` = Component (1P), `TaskGroup` = Composite (1P), `SimpleTask` = Leafs (1P)
- 4P Bridge: `Export` = Abstraction (1P), `GanttDiagram` = Refined Abstraction (1P), `OutputFormat` = Implementor (1P), `CSV` = Concrete Implementor (1P)

Name: _____ Matriculation Number: ⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋

b) Assess the Design — 10P

1. Give one example where the design is open for extension but closed for modification and one example where the design is open for extension but not closed for modification. Provide a concrete example for both extensions. (4P)

2. With respect to the Open Closed Principle name the patterns that are **implemented in a conflicting manner** and describe the conflict. (2P)

3. Are there any violations of the Liskov Substitution Principle? (1P)

4. Part of which design pattern(s) could the class `DependentTask` be? Name the information that is required in order to make a final decision. (3P)

Solution

1. The design is open for extensions and closed for modification when extending `Task` (1P) with new functionality using the visitor pattern. This does not apply for adding new tasks, since new tasks would require to modify the `Visitor` class (1P). One concrete example for each case (2P).
2. The composite (0.5P) and visitor pattern (0.5P) are conflicting. The visitor prevents the extension of new Task types. (1P)
3. No. (1P)
4. Decorator pattern or proxy pattern. In order to make a clear distinction, you either need an implementation of a concrete decorator adding new behavior to `dependsOn`, or ...?

Name: _____ Matriculation Number: ⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋

c) Strategies in Functional Programming Languages — 1P

Which feature makes it easy to implement the Strategy design pattern in Scala (and functional programming languages)?

Solution

Closures, higher-order functions or lambda expressions (1P)

d) Sortable Sets — 6P

Sketch two designs for a generic `SortableSet` trait in Scala that supports sorting. Assume that you have a base class `Set[+A]`, which is covariant in `A`. The outcome of the sorting method should be a fresh set and the order of elements depends on how elements are compared. One of your designs should use the template pattern and the other the strategy pattern to parameterize the sorting process with a comparison operation.

It is sufficient to write two Scala class definitions for the above design. You do not have to implement the sorting method, but indicate in a comment where you use the respective comparison operation.

Discuss the advantages and disadvantages of each design in not more than 4 sentences.

Solution

```
// template
trait SortableSet[+A] extends Set[A] {
  def compare[B >: A](a1: B, a2: B): Int

  def sort(): SortableSet[A] = ... // using compare
}

// strategy
trait SortableSet[+A] extends Set[A] {
  def sort[B >: A](cmp: (B, B) => Int): SortableSet[A] = ... // using cmp
}
```

2P for each class:

1. 0.5P for extending `Set`
2. 0.5P for **either** using invariant `A` **or** covariant `A` and bounded `B`.
3. 1P for correct pattern design.

In the template design, comparison is a parameter of the set itself. To sort elements of an existing set with a different comparison operation, we have to create a new subclass of the set and copy the original set (1P). In the strategy design, comparison is a parameter of the sorting operation and the above problem does not arise (1P).

Name: _____ Matriculation Number: ⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋

Topic 3: Reactive Programming — 14P

a) Signals vs Events — 3P

Explain the difference between signals (i.e., time-varying values) and events in not more than two sentences.

For each of the following abstractions, say whether it is conceptually a signal (i.e., time-varying value) or rather an event.

1. The oil pressure inside of a turbine.
2. The contents of a text field.
3. The latest shot on a goal in a soccer game.
4. A tap on a touch screen.

Solution

A signal is defined/holds a value at all times. (0.5P)

An event on the other hand is defined/holds a value only at concrete points in time (propagation turns). (0.5)

1. Signal
2. Signal
3. Signal or Event
4. Event

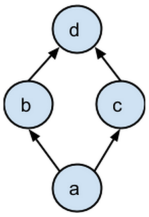
Name: _____ Matriculation Number: ⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋

b) Glitches 5P

1. Explain what glitches are.
2. Describe **two** reasons why glitches are not desirable in a reactive programming language.
3. Show an example of a graph configuration where a glitch can occur. Show **two** propagation orders, one with a glitch and one without.

Solution

1. Glitches are temporary spurious values due to the propagation order in the graph. They occur when a node is updated using some new and some old values in the predecessors. (1P)
2. two of them are enough: (1P each, max is 2P)
   - nodes are evaluated redundantly
   - spurious values have no meaning
   - events can erroneously fire more than once
3. glitch: a → b → d → c → d or a → c → d → b (0.5P)
   no glich: a → b → c → d or a → c → b → d (0.5P)



(1P)

Name: _____ Matriculation Number: ⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋

c) Video animation 6P

Video animations consist of still images, called ***frames***, rendered at a high rate – ideally around 60 frames per second. For debugging and logging purposes, video games can often display how many frames per second they currently render. Since the render time can vary from frame to frame by a certain amount, the ***frames per second*** number is often averaged over the last *n* frames.

Given a signal

```
val frameTime: Signal[Int]
```

which holds the number of milliseconds of the last frame that was rendered completely. You can assume that two consecutive frames never take the same time to render. Implement a signal `fps`, which holds the current value of ***frames per second***, averaged over the last 50 frames. At the beginning of the animation, when less than 50 frames have been rendered in total, average over all previously rendered frames.

Solution

Using `last`:

```
val frameTime: Signal[Int]
val e = frameTime.changed // 1P
val window = e.last(50) // 2P
// 1P for signal constructor, 0.5P for sum apply,
// 0.5P for length apply, 1P for correct logic:
val fps = Signal { 1000.0 / (window().sum / window().length) }
   // or: Signal { 1000.0 * window().length / window().sum }
```

Using `fold` (encoding `last`) with a queue:

```
...
// 0.5P for using fold, 0.5P for initial value
// 0.5P for special case < or >= 50, 0.5P for queue construction
val window = e.fold(new Queue[Int]) { (acc, t) =>
  if(acc.length >= 50) acc.dequeue
  acc enqueue t
  acc
}
...
```

Using `fold` (encoding `last`) with a list:

```
...
// initial value can be Nil, even though that does not type check
// grading same as for the queue version
val window = e.fold(Nil) { (acc, t) =>
  val res = if(acc.length < 50) acc else acc.tail
  res ::: List(t)
}
...
```

Using `fold` to go the whole way:

```
val frameTime: Signal[Int]
val e = frameTime.change // 1P
val fps = e.fold(0) { (acc : (Double, Int, Int), t) =>
  val i = if (acc._2 < 50) acc._2+1 else 50
  val res = acc._1 -
```

```
} map (_._1)
...
```

Name: _____ Matriculation Number: ⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋⌊_⌋

Topic 4: Software Design — 21P

Your task is to design the architecture for a computer-aided design tool (CAD). Use design patterns and follow design principles to adhere to the description below.

Your CAD tool should have multiple components:

1. **Solids.** The basic building block inside of the CAD tool is a solid. A cuboid is one example for a solid. It is defined by its width, height and length and it consists of a specific material. It should also be possible to group multiple solids into a new one.
2. **Transformations.** Within your CAD tool it should be possible to apply various transformations to solids. For example, it should be possible to move or rotate a solid. However, transformations should only be applicable if desired. Not every solid should be movable or rotatable, some might be either movable or rotatable or neither of them. It should also be possible to undo the last n actions (e.g. transformations applied) performed in the CAD tool.
3. **Operations.** In addition, it should be possible to define various operations on solids, like calculating the volume or the center of a solid. For example, the volume of a cuboid is calculated by multiplying its width, height and length.

Sketch your design as Scala code. Use expressive class and method names and shortly document the used patterns and the responsibilities of each class you add in the pattern.

The sketch has to contain the code for:

- Representing solids
- Group multiple solids
- Performing the computation of the overall volume
- Moving a solid
- (Un)doing the last n transformations

Hint: You can omit method implementations that are not necessary for your pattern(s) by using ???.

Solution

```
trait Solid {
  def accept[T](v: Visitor[T]): T
}

class Cuboid(val width: Double, val height: Double, val length: Double) extends Solid
with Rotatable with Movable {
  def accept[T](v: Visitor[T]): T = v visit this
}

class CompositeSolid extends Solid {
  var solids = List[Solid]()
  def add(s: Solid): Unit = solids = s :: solids
  def accept[T](v: Visitor[T]): T = v visit this
}

trait Visitor[T] {
  def visit(solid: Cuboid): T
  def visit(cs: CompositeSolid): T
}

class VolumeCalculation extends Visitor[Double] {
  def visit(c: Cuboid): Double = {
    (c.width * c.height * c.length)
  }
  def visit(cs: CompositeSolid): Double = {
    cs.solids.map(s => s.accept(this)).sum
  }
}

trait Command {
  def execute()
  def undo()
}

class MoveCommand(m: Movable, offset: (Double, Double, Double)) extends Command {
  def execute(): Unit = {
    m.position = (m.position._1 + offset._1, m.position._2 + offset._2,
            m.position._3 + offset._3)
  }
  def undo(): Unit = {
    m.position = (m.position._1 - offset._1, m.position._2 - offset._2,
            m.position._3 - offset._3)
  }
}

trait Movable {
  var position: (Double, Double, Double)
}

trait Rotatable {
  var rotation: (Double, Double, Double)
}
```

```
class CADTool() {
  private var history = List[Command]()

  def run(c: Command): Unit = {
    c.execute()
    history = c :: history
  }

  def undo(n: Int): Unit = {
    history.take(n).foreach {
      c => c.undo()
    }
    history = history.drop(n)
  }
}
```