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

Topic 1: Introduction — 25P

a) True Or False? — 9P

For each of the following statements, indicate whether it is ***true*** or ***false*** below each statement. ***Shortly justify your answer***. (1.5P per answer)

1. In a project written in an object-oriented programming language the software design differs from one written in a functional programming language.

2. Functional programming languages and declarative programming languages support the same design patterns.

3. A trait in Scala cannot implement methods.

4. It is possible to have a single design pattern distributed over multiple class files.

5. One of the S.O.L.I.D principles is formally defined.

6. In Scala, pattern matching can be used as an alternative to the visitor pattern.

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

b) Basic Knowledge — 8P

1. Name two differences between closures in Scala and Java 8? ***Mark*** the occurrences of these differences in the following Scala code fragment. (3P)

```
1  def foo: Int = {
2     var x = 1
3     val f = (y: Int) =>
4         if (y > 10) {
5               x += 1
6               x
7         } else
8             return y
9     f(f(x))
10 }
```

2. How can the designer of an API prevent that certain methods are overridden by the user of the API? (1P)

3. Which design pattern is implemented in the following code snippet? (1P)

```
// Library
trait IntPair {
   def left: Int
   def right: Int
}

// Your application
class IntPoint(p: IntPair) extends Point {
   def x = p.left
   def y = p.right
}
```

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

4. The following code snippet shows three classes where the subclass EncryptedDirecotry is shown in two distinct versions. Describe for each respective subclass one possible scenario where the fragile base class problem can occur when an ***addAll*** method is added to the base class. (3P)

```
class Directory { /* version 1.5*/
  private var files = Set.empty[File]

  def add(file: File): Unit = files += file
  def delete(file: File) : Unit = files = files.filterNot( _ eq file)
}

/* Based on Directory version 1.0 */
class EncryptedDirectory(encryptionScheme: EncryptionScheme) extends Directory {

  override def add(file: File): Unit = super.add(encryptionScheme.encrypt(file))
  override def delete(file: File) : Unit =
    if (file.isEntrypted) super.delete(file)
    else super.delete(encryptionScheme.encrypt(file))
}

/* Based on Directory version 1.5 */
class EncryptedDirectory(encryptionScheme: EncryptionScheme) extends Directory {
   def addAll(files: List[File]): Unit = files foreach {file => add(file) }
   override def add(file: File): Unit = super.add(encryptionScheme.encrypt(file))
   override def delete(file: File) : Unit =
      if (file.isEntrypted) super.delete(file)
      else super.delete(encryptionScheme.encrypt(file))
}
```

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

c) Traits and Mixin Composition 6P

Given the following class and trait definitions, give the class linearization of class `F`. Show all steps how to derive the linearization by using the notation of the following example:

```
abstract AbsIterator extends AnyRef { ... }
trait RichIterator extends AbsIterator { ... }
class Iter extends AbsIterator with RichIterator { ... }

Lin(Iter) = {Iter, Lin(RichIterator ≻≻ Lin(AbsIterator)) }
          = {Iter, Lin(RichIterator) ≻≻ { AbsIterator, AnyRef}}
          = {Iter, {RichIterator, AbsIterator, AnyRef} ≻≻ {{ AbsIterator, AnyRef}}}
          = {Iter, RichIterator, AbsIterator, AnyRef}
```

Now derive the class linearization of class `F`:

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

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

d) Forwarding vs Delegation 2P

Implement forwarding and delegation semantics in the method `m()` in the respective classes.

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

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

class OtherClassForwarding extends Base {
  val callee = new Callee()
  def m() { // Add code here for forwarding:




  }
  ...
}

class OtherClassDelegation extends Base {
  val callee = new Callee()
  def m() { // Add code here for delegation:




  }
  ...
}
```

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

Topic 2: Design Patterns 30P

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

```
trait Statement {
   var errorListeners = Set.empty[(Statement) => Unit]
   def registerErrorListener(f: (Statement) => Unit) = {errorListeners += f}
   def fireInterpretationError(stat: Statement) = {errorListeners foreach( f => f(stat))}
   def interpret(env: Map[Symbol, Int]): Statement
   def accept(visitor: Visitor): Unit
}

case class InterpretedStatement(val result: Option[Int], val original: Statement) extends Statement{
   override def interpret(env: Map[Symbol, Int]): Statement = result match {
      case Some(result) => this
      case None => original.interpret(env)
   }
   override def accept(visitor: Visitor): Unit = visitor.visit(this)
}

case class Return(val value: Symbol) extends Statement {
   override def interpret(env: Map[Symbol, Int]): Statement = env.get(value) match {
      case Some(value) => InterpretedStatement(Some(value), this)
      case None => fireInterpretationError(this); InterpretedStatement(None, this)
   }
   override def accept(visitor: Visitor): Unit = visitor.visit(this)
}

case class Assignment(val name: Symbol, val value: Int, val next: Statement) extends Statement {
   override def interpret(env: Map[Symbol, Int]): Statement = next.interpret(env.+(name -> value))
   override def accept(visitor: Visitor): Unit = visitor.visit(this)
}

case class Condition(val lhs: Symbol, val rhs: Symbol, val check: (Int, Int) => Option[Boolean]){
   def interpretCondition(env: Map[Symbol, Int]): Option[Boolean] = (env.get(lhs), env.get(rhs)) match {
      case (Some(lhs), Some(rhs)) => check(lhs, rhs)
      case _ => None
   }
}

object Condition{
   def createGreaterThan(lhs: Symbol, rhs: Symbol): Condition = Condition(lhs, rhs, (l, r) => Some(l > r))
}

case class Branch(val cond: Condition, val thenBranch: Statement, val elseBranch: Statement) extends Statement{
   override def interpret(env: Map[Symbol, Int]): Statement = cond.interpretCondition(env) match {
      case Some(true) => thenBranch.interpret(env)
      case Some(false) => elseBranch.interpret(env)
      case None => fireInterpretationError(this); this
   }
   override def accept(visitor: Visitor): Unit = visitor.visit(this)
}

trait Visitor {
   def visit(ret: Return): Unit
   def visit(intSmt: InterpretedStatement): Unit
   def visit(assign: Assignment): Unit
   def visit(branch: Branch): Unit
}
```

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

a) "Pattern Recognition" max 14P

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.

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

b) Assess the Design — 9P

1. The design of the previous example is open for which kind of extensions and closed against which kind of modifications? (2P)

2. For two of the **implemented** patterns that make it possible to extend the design, discuss a concrete example of a possible extension. (2P)

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

4. Are there any violations of the Liskov Substitution Principle? If yes, name them. (1P)

5. Do you see any potential to apply the Interface Segregation Principle to the trait `Expr`? Justify your answer! (2P)

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

c) Strategies in Functional Programming Languages — 1P

Which feature makes it possible to implement a new alternative of the Decorator design pattern in Scala? Shortly describe this alternative by discussing the consequences.

d) Filtered Sets — 6P

Sketch two designs for a generic `FilteredSet` trait in Scala that supports filtering a set. Assume that you have a base class `Set[+A]`, which is covariant in `A`. The outcome of the filter method should be a fresh set and this set only contains the elements for which a predicate holds. One of your designs should use the template pattern and the other the strategy pattern to parameterize the filtering process with a predicate operation.

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

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

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

Topic 3: Software Design — 15P

a) Path dependent types — 7P

Look at the following code example. The lines that start with `[ ]` contain definitions that might or might not type-check. Please mark every line that type-checks with a ✓ and every line that does not type-check with an x.

```
case class Board(length: Int, height: Int) {
  case class Coordinate(x: Int, y: Int) {
    require(0 <= x && x < length && 0 <= y && y < height)
  }
  val occupied = scala.collection.mutable.Set[this.Coordinate]()
}

val b1 = Board(20, 20)
val b2 = Board(30, 30)
val b3: Board = b1
val b4: b1.type = b1

val c1 = b1.Coordinate(15, 15)
val c2 = b2.Coordinate(25, 25)
val c3 = b3.Coordinate(10,10)
val c4 = b4.Coordinate(10,10)

[ ]   b1.occupied += c1
[ ]   b2.occupied += c2
[ ]   b1.occupied += c2
[ ]   b1.occupied += c3
[ ]   b2.occupied += c3
[ ]   b1.occupied += c4
[ ]   b2.occupied += c4
```

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

b) Visitor vs. Pattern Matching — 8P

In Scala, pattern matching can be used to more elegantly model simple visitor patterns. In this task you are supposed to transform a given example from a visitor pattern to using pattern matching.

The following code example shows evaluation and printing of simple arithmetic expressions using a visitor pattern. (The task follows on the next page.)

```
////// Class hierarchy
trait Expr {
  def accept[Result](v: Visitor[Result]): Result
}

case class Constant(val value: Int) extends Expr {
  def accept[Result](v: Visitor[Result]) = v.visit(this)
}
case class Var(val name: String) extends Expr {
  def accept[Result](v: Visitor[Result]) = v.visit(this)
}
case class Add(val left: Expr, val right: Expr) extends Expr {
  def accept[Result](v: Visitor[Result]) = v.visit(this)
}
case class Mul(val left: Expr, val right: Expr) extends Expr {
  def accept[Result](v: Visitor[Result]) = v.visit(this)
}

////// Visitors
trait Visitor[Result] {
  def visit(e: Constant): Result
  def visit(e: Add): Result
  def visit(e: Mul): Result
  def visit(e: Var): Result
}

class EvalVisitor(env: Map[String, Int]) extends Visitor[Int] {
  def visit(e: Constant) = e.value
  def visit(e: Add) = e.left.accept(this) + e.right.accept(this)
  def visit(e: Mul) = e.left.accept(this) * e.right.accept(this)
  def visit(e: Var) = env(e.name)
}

class FormatVisitor extends Visitor[String] {
  def visit(e: Constant) = e.value.toString
  def visit(e: Add) = s"(${e.left.accept(this)} + ${e.right.accept(this)})"
  def visit(e: Mul) = s"(${e.left.accept(this)} * ${e.right.accept(this)})"
  def visit(e: Var) = e.name
}


object VisitorExpr extends App {
  val testExpr = Add(Mul(Constant(2),Var("x")), Constant(5))

  println(testExpr.accept(new FormatVisitor()))
  println(testExpr.accept(new EvalVisitor(Map("x" -> 7))))
}
```

Name: _____ Matriculation Number: ⌴⌴⌴⌴⌴⌴⌴

Implement two methods ***eval*** and ***format*** which have the same functionality as the corresponding ***EvalVisitor*** and ***FormatVisitor*** visitors. Implement these methods by using pattern matching and do ***NOT*** use the accept methods on the class hierarchy or any visitor class.

Also write down the modified ***println*** statements that use your implemented methods on ***testExpr***.

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

Topic 5: Reactive Programming — 14P

a) Reactive Programming vs. Observer Design Pattern — 3P

Explain **three** advantages of reactive programming in comparison to the observer design pattern.

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 in which a glitch can occur. Show **two** propagation orders, one with a glitch and one without.

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

c) Event processing 6P

The following listed events and signals describe a small event processing engine. Your task is to implement the following events and signals using REScala syntax. For each value or variable specify the correct type. (e.g. `val i: Int = -1`)

- Event `e1`, which does carry an integer value,
- Event `e2`, which carries no value respectively,
- Signal `s1`, which provides a sequence of the last `10` events in the event stream `e1`,
- Signal `s2`, which provides the sum of the values contained in `s1`,
- Signal `s3`, which contains the value of `s1` updated the last time `e2` fired,
- Signal `s4`, which keeps track of how many `e2` events have been fired since the start of the application.