Tuan Nguen

Question 1:

```
import scala.swing._
import scala.swing.event._

class Dictionary(fname: String){
  /** A Set object holding the words */
  private val words = new scala.collection.mutable.HashSet[String]

  /** Initialise dictionary from fname */
  private def initDict(fname: String) = {
    val allWords = scala.io.Source.fromFile(fname).getLines
    // Should word w be included?
    def include(w:String) = w.forall(_.isLower)
    for(w <- allWords; if include(w)) words += w
    // println("Found "+words.size+" words")
  }

  // Initialise the dictionary
  initDict(fname)

  /** test if w is in the dictionary */
  def isWord(w: String) : Boolean = words.contains(w)
}

object WordPaths extends SimpleSwingApplication{
  /** The dictionary */
  var dict : Dictionary = null

  /** A type representing paths through the graph of words */
  type Path = List[String]

  /** Print the Path path, separating entries with commas.
    * Pre: path is non-empty. */
  def printPath(path: Path) = {
    print(path.head)
    for(w <- path.tail) print(", "+w)
    println
  }

  /** Find all neighbours of w */
  def neighbours(w: String) : Path = {
    var result = List[String]() // build up the result
    for(i <- 0 until w.length; c <- 'a' to 'z')
      if(c != w(i)){
```

```
        val w1 = w.patch(i,List(c),1) // replace ith character
                                      // of w with c
        if(dict.isWord(w1)) result = w1 :: result
      }
    result
  }

  /** Find a minimum length path from start to target.
    * @return Some(p) for some shortest Path p if one exists;
    * otherwise None. */
  def findPath(start: String, target: String) : Option[Path] = {
    // We'll perform a breadth-first search.  Each node of the search
graph
    // will be a list of words, consecutive words differing in one
letter, and
    // ending with start, thereby representing a path (in reverse
order)
    val queue = scala.collection.mutable.Queue(List(start))
    // Keep track of the words we've already considered
    val seen = new scala.collection.mutable.HashSet[String]
    seen += start

    while(!queue.isEmpty){
      val path = queue.dequeue; val w = path.head
      for(w1 <- neighbours(w)){
        if(w1==target) return Some((target::path).reverse)
        else if(!seen.contains(w1)){seen += w1; queue += w1::path}
      } // end of for
    } // end of while
    None // no solutions found
  } // end of findPath

  def top = new MainFrame {
    var s = ""; var t = ""
    var dictFile = "knuth_words"
    dict = new Dictionary(dictFile)

    title = "Words paths"
    val start = new TextField { columns = 5 }
    object target extends TextField { columns = 5 }
    val tx = new TextArea("A text area") {
      lineWrap = true
    }

    contents = new FlowPanel {
      contents += new Label(" Start ")
```

```
      contents += start
      contents += new Label(" Target ")
      contents += target
      contents += new Label(" Path ")
      contents += tx
      border = Swing.EmptyBorder(10, 10, 10, 10)
    }

    listenTo(start, target)

    reactions += {
      case EditDone(`start`) =>
        s = start.text
        if(t != "") {
          val optPath = findPath(s, t)
          optPath match{
            case Some(path) => tx.text = path mkString ", "
            case None       => tx.text = "No path found"
          }
        }
      case EditDone(`target`) =>
        t = target.text
        if(s != "") {
          val optPath = findPath(s, t)
          optPath match {
            case Some(path) => tx.text = path mkString ", "
            case None       => tx.text = "No path found"
          }
        }
    }
  }
}
```

Question 2:

```
class MySet[T](var elements: Set[T]) extends Set[T] {
  override def empty: MySet[T] = new MySet[T](null)

  def contains(key: T): Boolean = {
    val filtered = elements.filter(_ == key)
    filtered.size != 0
  }

  def iterator: Iterator[T] = {
    elements.toIterator
  }
```

```
  def +(elem: T) = {
    elements += elem
    new MySet(elements)
  }

  def -(elem: T) = {
    elements = elements.filterNot(_ == elem))
    new MySet(elements)
  }
}

object Test{
  def main(args: Array[String]) = {
    val elements = Set(1, 2, 3, 4, 5, 6, 7, 8)
    var mySet = new MySet[Int](elements)
    println(mySet.contains(7)) // true
    println(mySet.contains(10)) // false
    mySet = mySet.+(10)
    println(mySet.contains(10)) // true
    mySet = mySet.+(10)
    mySet = mySet.-(10)
    println(mySet.contains(10)) // false
  }
}
```

Question 3:

a)

```
trait PartialOrder[T] {
  def <=(that: T): Boolean
  def lub(that: T): T
}

class MySet[T](var elements: Set[T]) extends Set[T] with
PartialOrder[MySet[T]] {
  override def empty: MySet[T] = new MySet[T](null)

  def contains(key: T): Boolean = {
    val filtered = elements.filter(_ == key)
    filtered.size != 0
  }

  def iterator: Iterator[T] = {
    elements.toIterator
  }

  def +(elem: T) = {
```

```
    elements += elem
    new MySet(elements + elem)
  }

  def -(elem: T) = {
    elements = elements.filterNot(_ == elem)
    new MySet(elements)
  }

  def <=(that: MySet[T]): Boolean = {
    var isSubset = true // flag
    var it = this.iterator
    while(it.hasNext && isSubset) {
      isSubset = that.contains(it.next())
    }
    isSubset
  }

  def lub(that: MySet[T]): MySet[T] = {
    new MySet(this.elements ++ that.elements)
  }

  /** For testing purposes */
  def printSet() = {
    var it = this.iterator
    it.foreach(x => print(x + " "))
    println()
  }
}

class UpSet[T <: PartialOrder[T]](var s: Set[T]) = {
  var it = s.toIterator
  var anyMinElem: T = it.next()
  var minElem: Set[T] = Set(it.next())
  while(it.hasNext){
    if(!(anyMinElem <= it.next()) && !(it.next() <= anyMinElem)){
      minElem + it.next()
    }
    else if(it.next <= anyMinElem){
      anyMinElem = it.next()
      minElem = Set(it.next())
    }
  }

  def contains(x: T): Boolean = {
    var filtered = minElem.filter(_ <= x)
```

```
    filtered.size != 0
  }

  def intersection(that: UpSet[T]): UpSet[T] = {
    val leastUpper = this.minElem.lub(that.minElem)
    new UpSet(leastUpper)
  }
}
```

Question 4:

```
class Bag[T](f: T => Int){
  def add(x: T): Bag[T] = {
    new Bag( y => if (y == x) f(y) + 1 else f(y) )
  }

  def remove(x: T): Bag[T] = {
    new Bag( y => if (y == x && f(y) > 0) f(y) - 1 else f(y) )
  }

  def count(x: T): Int = f(x)

  def union(that: Bag[T]): Bag[T] = {
    new Bag( y => if(f(y) != 0 || that.count(y) != 0)
                  f(y) + that.count(y) else f(y) )
  }
}

object Test{
  def main(args: Array[String]) = {
    val b0: Bag[Any] = new Bag((x) => List(0, 0.0, "zero",
0).count(x==_))
    val b1: Bag[Any] = new Bag((x) => List(1, 1.1, "one",
1).count(x==_))
    val b2: Bag[Int] = new Bag((x) => List(2, 3).count(x==_))
    val b3: Bag[Any] = b0 union b1

    println(b0.add("zero").count("zero") + b1.count(1.1) +
b2.count(2))
    println(b3.remove(0).count(0) + ", " + b3.count(1) + ", " +
b3.count(2))
  }
}
```

Question 5:

Suppose mutable collections are covariant. Then Array[Int] would be a subtype of Array[Any], and we would be able to make a reference from a type Array[Any] variable to Array[Int]. Now suppose we add 3.14 to the array of type Array[Any]. Then we would have to add a float to an array of type Array[Int]. Therefore, mutable collections are not covariant. Similar argument for why mutable collections are not contravariant.

Question 7:

The Façade design pattern provides a unified interface to a set of interfaces in a subsystem. Using this design pattern we minimize the communication and dependence between subsystems. This way we reduce complexity.